Introduction to R

Claudia A Engel

Last updated: October 20, 2017

Contents

Prerequisites

- Geared specifically towards users who are **new** to R.
- Have R and RStudio installed (see setup instructions below).

Next workshop:
Thu, October 19, 2017, 2-4pm (if you would like help setting up please arrive at 1:30) Room 433A in CESTA (Building 160 - Wallenberg 4th floor)

Setup Instructions

R and **RStudio** are separate downloads and installations. 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 R before you install RStudio.

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
- 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.

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
- 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.

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 3.3.1.
- 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.

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.

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 interdisciplinary and extensible

With 10,000+ packages 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 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.

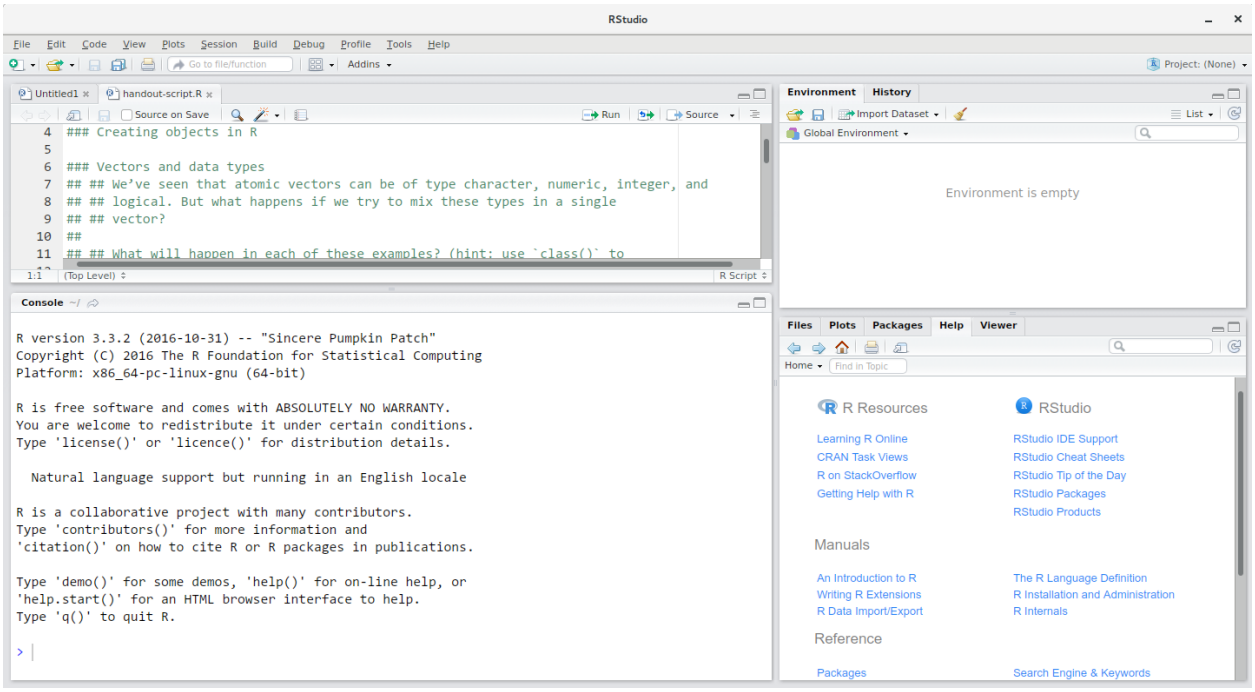


Figure 1.1: The RStudio Interface

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.

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.

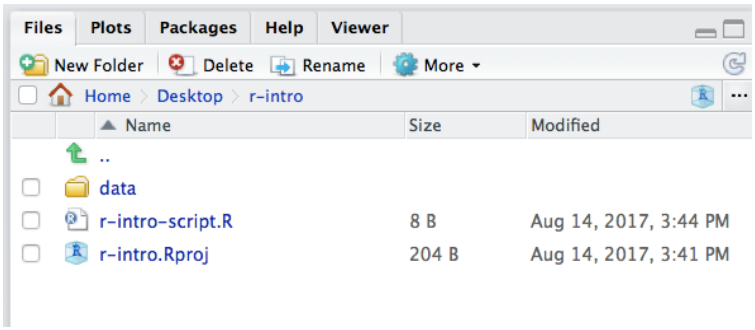


Figure 1.2: What it should look like at the beginning of this lesson

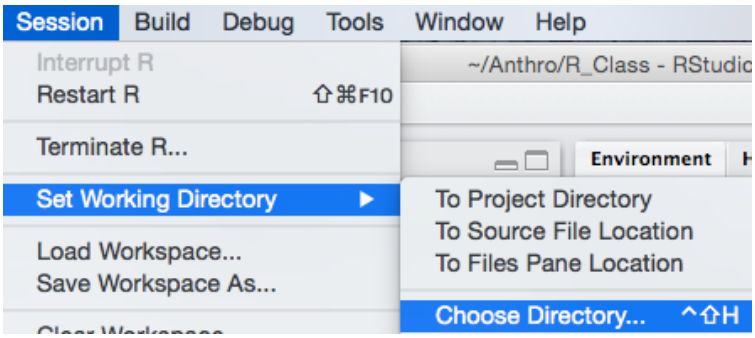


Figure 1.3: How to set a working directory with the RStudio interface

- 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 working directory.

Your working directory should now look like in Figure 1.2.

If you ever need to set a different working directory you can use the RStudio interface like seen in Figure 1.3.

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

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

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

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

```
getwd()
```

1.4.1 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 **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.

One of the first things you will notice is 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 and hit the `Tab` key, it will make suggestions. More on code completion in RStudio is here: https://support.rstudio.com/hc/en-us/articles/205273297-Code-Completion

All in all, RStudio will make typing 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:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `weight_kg <- 55`, the value of `weight_kg` is `55`. The arrow can be read as `55` **goes into** `weight_kg`. For historical reasons, 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:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
weight_kg          # and so does typing the name of the object
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

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

```
weight_kg <- 57.5
2.2 * weight_kg
```

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

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

> Challenge
>
> What do you think is the current content of the object `weight_lb`? 126.5 or 220?

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
>
> What are the values after each statement in the following?

```
mass <- 47.5            # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass/age  # mass_index?
```

2.1.2 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc.

They all have in common that they are executed by typing their name followed by round brackets, in which we provide one or more parameters (or arguments) for the function to do something, separated by commas.

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*. A typical example would be the function `sqrt()`. 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:

```
a <- 64
b <- sqrt(a)
```

Here, we assign the value `64` to he variable `a`, which is then given to the `sqrt()` function. The `sqrt()` function calculates the square root, and returns the value which is then assigned to variable `b`. This function is very simple, because it takes just one argument.

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()`.

```
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`.

```
args(round)
```

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

```
?round
```

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

```
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:

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

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

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

```
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 `weight_g`:

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

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(weight_g)
```

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(weight_g)
```

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(weight_g)
```

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

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

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

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:

```
animals <- c("mouse", "rat", "dog", "bear")
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).

```
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

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):

```
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

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"
```

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance. We can use this to select a sequence, like this:

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

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

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

```
animals[-2]
```

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

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

```
#> [1] "bear"
```

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" "bear"
```

R indices start at 1. Programming languages like Fortran, 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
```

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

```
animals[has_tail]
```

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

Typically, these logical vectors are not typed out by hand, but are the output of other functions or 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, 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[animals == "bear" | animals == "rat"] # returns both rat and cat
```

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

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

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

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

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

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

```
weight_g > 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
weight_g[weight_g > 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):

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

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

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
#> numeric(0)
```

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.

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

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

```
sum(heights)
```

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

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

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

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

```
#> [1] 16
```

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.
heights[!is.na(heights)]
```

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

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

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

```
# Extract elements which are complete cases.
heights[complete.cases(heights)]
```

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

Challenge

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

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

2. Use the function `median()` to calculate the median of the `lengths` vector.

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

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

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(lubridate)
```

or

```
require(lubridate)
```

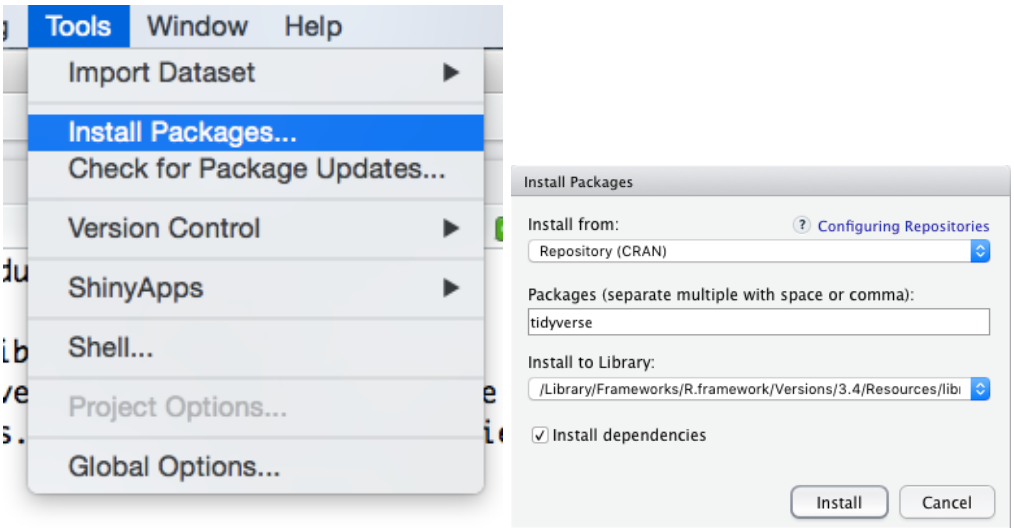


Figure 2.1: How to install an R package with the RStudio interface

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

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:

```
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

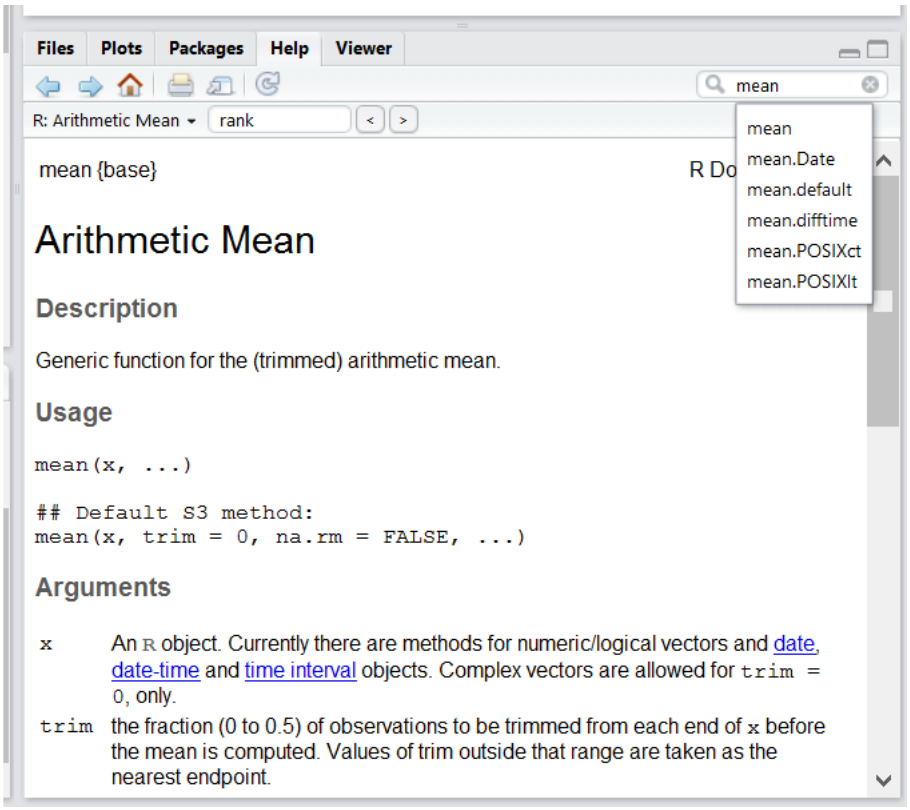


Figure 2.2: The RStudio help interface

```
??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:

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

```
#> 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), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), .Names = c("Sepal.Length",
#> "Sepal.Width", "Petal.Length", "Petal.Width", "Species"), row.names = c(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:

```
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):

```
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.

```
sessionInfo()
```

```
#> R version 3.4.2 (2017-09-28)
#> Platform: x86_64-apple-darwin15.6.0 (64-bit)
#> Running under: macOS Sierra 10.12.6
#>
#> Matrix products: default
#> BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib
#>
#> locale:
#> [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> loaded via a namespace (and not attached):
#>  [1] compiler_3.4.2  backports_1.0.5 bookdown_0.4.2  magrittr_1.5
#>  [5] rprojroot_1.2   tools_3.4.2     htmltools_0.3.6 rstudioapi_0.6
#>  [9] yaml_2.1.14     Rcpp_0.12.12    stringi_1.1.5   rmarkdown_1.6
#> [13] knitr_1.17      stringr_1.2.0   digest_0.6.12   evaluate_0.10.1
```

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.

3.1 Loading tabular data

One the most common ways of getting data into R is to read in a table. And – you guessed it – we read it into a data frame! We will take a simple CSV file as example. What is a CSV file?

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

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:

```
download.file("https://github.com/cengel/R-intro/raw/master/data/MS_trafficstops_bw.csv",
              "data/MS_trafficstops_bw.csv")
```

You are now ready to load the data:

```
trafficstops <- read.csv('data/MS_trafficstops_bw.csv')
```

This statement doesn't produce any output because, as you might recall, assignments don't display anything. If we want to check that our data has been loaded, we can print the variable's value: `trafficstops`.

Wow… that was a lot of output. At least it means the data loaded properly. Let's check the top (the first 6 lines) of this data frame using the function `head()`:

```
head(trafficstops)
```

```
#>              id state  stop_date       county_name county_fips
#> 1 MS-2013-00001    MS 2013-01-01      Jones County       28067
#> 2 MS-2013-00002    MS 2013-01-01 Lauderdale County       28075
#> 3 MS-2013-00003    MS 2013-01-01       Pike County       28113
#> 4 MS-2013-00004    MS 2013-01-01    Hancock County       28045
#> 5 MS-2013-00005    MS 2013-01-01     Holmes County       28051
#> 6 MS-2013-00006    MS 2013-01-01    Jackson County       28059
#>            police_department driver_gender driver_birthdate driver_race
#> 1 Mississippi Highway Patrol             M       1950-06-14       Black
#> 2 Mississippi Highway Patrol             M       1967-04-06       Black
#> 3 Mississippi Highway Patrol             M       1974-04-15       Black
#> 4 Mississippi Highway Patrol             M       1981-03-23       White
#> 5 Mississippi Highway Patrol             M       1992-08-03       White
#> 6 Mississippi Highway Patrol             F       1960-05-02       White
#>                                                violation_raw officer_id
#> 1                     Seat belt not used properly as required       J042
#> 2                                          Careless driving       B026
#> 3 Speeding - Regulated or posted speed limit and actual speed       M009
#> 4 Speeding - Regulated or posted speed limit and actual speed       K035
#> 5 Speeding - Regulated or posted speed limit and actual speed       D028
#> 6 Speeding - Regulated or posted speed limit and actual speed       K023
```

3.2 Inspecting `data.frame` Objects

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.).

We can see this when inspecting the structure of a data frame with the function `str()`:

```
str(trafficstops)
```

```
#> 'data.frame':    211211 obs. of  11 variables:
#>  $ id              : Factor w/ 211211 levels "MS-2013-00001",..: 1 2 3 4 5 6 7 8 9 10 ...
#>  $ state           : Factor w/ 1 level "MS": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ stop_date       : Factor w/ 1288 levels "2013-01-01","2013-01-02",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ county_name     : Factor w/ 82 levels "Adams County",..: 34 38 57 23 26 30 30 22 26 26 ...
#>  $ county_fips     : int  28067 28075 28113 28045 28051 28059 28059 28043 28051 28051 ...
#>  $ police_department: Factor w/ 1 level "Mississippi Highway Patrol": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ driver_gender   : Factor w/ 3 levels "","F","M": 3 3 3 3 3 2 2 2 3 3 ...
#>  $ driver_birthdate : Factor w/ 21423 levels "","1930-01-11",..: 3558 9575 12137 14670 18820 7061 4504 191
#>  $ driver_race     : Factor w/ 3 levels "","Black","White": 2 2 2 3 3 3 3 3 3 3 ...
#>  $ violation_raw   : Factor w/ 19 levels "??","Careless driving",..: 17 2 19 19 19 19 19 19 19 19 ...
#>  $ officer_id      : Factor w/ 897 levels "","A003","A004",..: 519 52 635 560 212 550 559 205 723 723 ...
```

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!

- Size:
  - `dim(trafficstops)` - 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(trafficstops)` - returns the number of rows
  - `ncol(trafficstops)` - returns the number of columns
  - `length(trafficstops)` - returns number of columns
- Content:
  - `head(trafficstops)` - shows the first 6 rows
  - `tail(trafficstops)` - shows the last 6 rows
- Names:
  - `names(trafficstops)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
  - `rownames(trafficstops)` - returns the row names
- Summary:
  - `str(trafficstops)` - structure of the object and information about the class, length and content of each column
  - `summary(trafficstops)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides `data.frame`.

Challenge

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

- What is the class of the object `trafficstops`?
- 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 trafficstops 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" we want from it. Row numbers come first, followed by column numbers. However, note that different ways of specifying these coordinates lead to results with different classes.

```
trafficstops[1, 1]   # first element in the first column of the data frame (as a vector)
trafficstops[1, 6]   # first element in the 6th column (as a vector)
trafficstops[, 1]    # first column in the data frame (as a vector)
trafficstops[1]      # first column in the data frame (as a data.frame)
trafficstops[1:3, 7] # first three elements in the 7th column (as a vector)
trafficstops[3, ]    # the 3rd element for all columns (as a data.frame)
trafficstops[1:6, ]  # equivalent to head(trafficstops)
trafficstops[,-1]           # the whole data frame, excluding the first column
trafficstops[-c(7:211211),] # equivalent to head(trafficstops)
```

As well as using numeric values to subset a `data.frame` (or `matrix`), columns can be called by name, using one of the four following notations:

```
trafficstops["violation_raw"]       # Result is a data.frame
trafficstops[, "violation_raw"]     # Result is a vector
trafficstops[["violation_raw"]]     # Result is a vector
trafficstops$violation_raw          # Result is a vector
```

For our purposes, the last three notations are equivalent. 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 `data.frame` (`trafficstops_200`) containing only the observations from row 200 of the `trafficstops` dataset.
2. Notice how `nrow()` gave you the number of rows in a `data.frame`?
   - 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 (`trafficstops_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 `trafficstops_middle`.
4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(trafficstops)` keeping just the first through 6th rows of the trafficstops dataset.

3.4 Conditional subsetting

Often times we need to extract a subset of a data frame based on certain conditions. For example, if we wanted to look at traffic stops in Tallahatchie County only we could say:

```
# the condition:
trafficstops$county_name == "Tallahatchie County" # returns a logical vector of the length of the colum
# use this vector to extract rows and all columns
trafficstops[trafficstops$county_name == "Tallahatchie County", ] # note the comma: we want all columns
# assign it to a new data frame
Tallahatchie_trafficstops <- trafficstops[trafficstops$county_name == "Tallahatchie County", ]
```

This is also a possibility (but slower):

```
Tallahatchie_trafficstops <- subset(trafficstops, county_name == "Tallahatchie County")
nrow(Tallahatchie_trafficstops) # 393 stops in Tallahatchie County!
```

```
#> [1] 393
```

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 trafficstops in Hancock, Harrison, and Jackson Counties into a separate data frame `coastal_counties`.
- Using `coastal_counties`, count the number of Black and White drivers in the three counties.
- Bonus: 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.

```
new_col <- row.names(trafficstops)
trafficstops_withnewcol <- cbind(trafficstops, new_col)
head(trafficstops_withnewcol)
```

```
#>              id state  stop_date       county_name county_fips
#> 1 MS-2013-00001    MS 2013-01-01      Jones County       28067
#> 2 MS-2013-00002    MS 2013-01-01 Lauderdale County       28075
#> 3 MS-2013-00003    MS 2013-01-01       Pike County       28113
#> 4 MS-2013-00004    MS 2013-01-01    Hancock County       28045
#> 5 MS-2013-00005    MS 2013-01-01     Holmes County       28051
#> 6 MS-2013-00006    MS 2013-01-01    Jackson County       28059
#>            police_department driver_gender driver_birthdate driver_race
#> 1 Mississippi Highway Patrol             M       1950-06-14       Black
#> 2 Mississippi Highway Patrol             M       1967-04-06       Black
#> 3 Mississippi Highway Patrol             M       1974-04-15       Black
#> 4 Mississippi Highway Patrol             M       1981-03-23       White
#> 5 Mississippi Highway Patrol             M       1992-08-03       White
#> 6 Mississippi Highway Patrol             F       1960-05-02       White
#>                                                  violation_raw officer_id
#> 1                     Seat belt not used properly as required       J042
#> 2                                             Careless driving       B026
#> 3 Speeding - Regulated or posted speed limit and actual speed       M009
#> 4 Speeding - Regulated or posted speed limit and actual speed       K035
#> 5 Speeding - Regulated or posted speed limit and actual speed       D028
#> 6 Speeding - Regulated or posted speed limit and actual speed       K023
#>   new_col
#> 1       1
#> 2       2
#> 3       3
#> 4       4
#> 5       5
#> 6       6
```

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.

```
trafficstops$row_numbers <- c(1:nrow(trafficstops))
trafficstops$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 `trafficstops` 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")

trafficstops_withnewrow <- rbind(trafficstops, new_row)
tail(trafficstops_withnewrow)
```

```
#>                    id state  stop_date         county_name county_fips
#> 211207 MS-2016-24293    MS 2016-07-09       George County       28039
```

```
#> 211208 MS-2016-24294    MS 2016-07-10       Copiah County       28029
#> 211209 MS-2016-24295    MS 2016-07-11      Grenada County       28043
#> 211210 MS-2016-24296    MS 2016-07-14       Copiah County       28029
#> 211211 MS-2016-24297    MS 2016-07-14       Copiah County       28029
#> 211212 MS-2017-12345    MS 2017-08-24 Tallahatchie County       12345
#>                police_department driver_gender driver_birthdate
#> 211207 Mississippi Highway Patrol             M       1992-07-14
#> 211208 Mississippi Highway Patrol             M       1975-12-23
#> 211209 Mississippi Highway Patrol             M       1998-02-02
#> 211210 Mississippi Highway Patrol             F       1970-06-14
#> 211211 Mississippi Highway Patrol             M       1948-03-11
#> 211212                       MSHP             F       1999-06-14
#>        driver_race
#> 211207       White
#> 211208       Black
#> 211209       White
#> 211210       White
#> 211211       White
#> 211212    Hispanic
#>                                                    violation_raw
#> 211207 Speeding - Regulated or posted speed limit and actual speed
#> 211208 Speeding - Regulated or posted speed limit and actual speed
#> 211209                     Seat belt not used properly as required
#> 211210       Expired or no non-commercial driver license or permit
#> 211211                     Seat belt not used properly as required
#> 211212                                                    Speeding
#>        officer_id
#> 211207       K025
#> 211208       C033
#> 211209       D014
#> 211210       C015
#> 211211       C015
#> 211212       ABCD
```

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.

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

When we did `str(trafficstops)` we saw that only one of the columns are numeric (`county_fips`), all the others are of a special class called a `factor`. 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 2 levels:

```
party <- factor(c("republican", "democrat", "democrat", "republican"))
```

R will assign `1` to the level `"democrat"` and `2` to the level `"republican"` (because `d` comes before `r`, even though the first element in this vector is `"republican"`). You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

```
levels(party)
nlevels(party)
```

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 `party` vector would be:

```
party # current order
```

```
#> [1] republican democrat   democrat   republican
#> Levels: democrat republican
```

```
party <- factor(party, levels = c("republican", "democrat"))
party # after re-ordering
```

```
#> [1] republican democrat   democrat   republican
#> Levels: republican democrat
```

In R's memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self describing: `"democrat"`, `"republican"` is more descriptive than `1`, `2`. Which one is "republican"? You wouldn't be able to tell just from the integer data. Factors, on the other hand, have this information built in. It is particularly helpful when there are many levels (like the county names in our example dataset).

3.6.1 Converting factors

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

```
as.character(party)
```

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

```
f <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(f)               # wrong! and there is no warning...
as.numeric(as.character(f)) # works...
as.numeric(levels(f))[f]    # The recommended way.
```

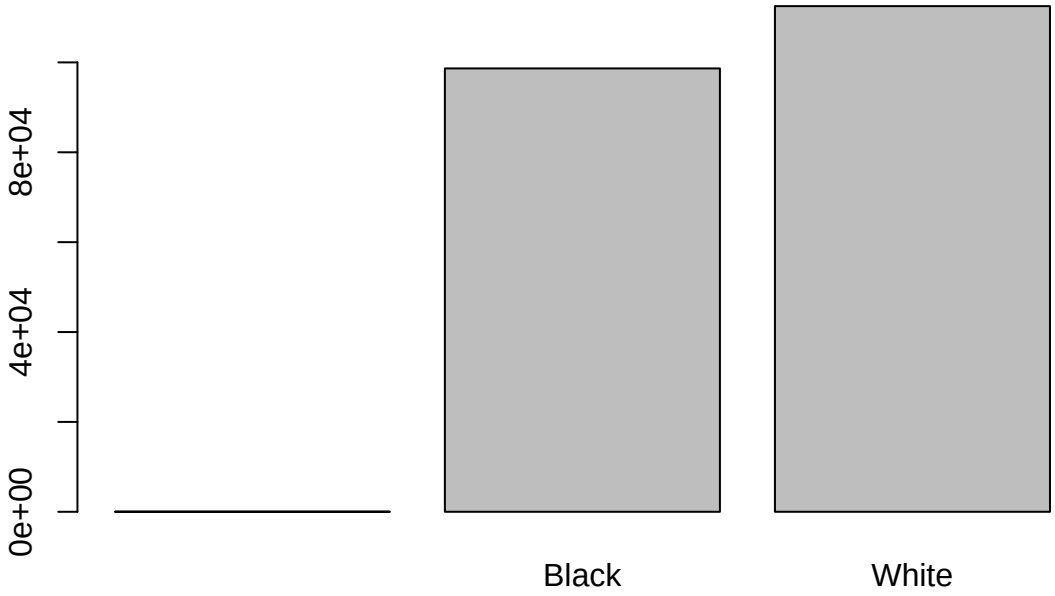
Notice that in the `levels()` approach, three important steps occur:

- We obtain all the factor levels using `levels(f)`
- We convert these levels to numeric values using `as.numeric(levels(f))`
- We then access these numeric values using the underlying integers of the vector `f` 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 blacks and whites in the dataset:

```
# bar plot of the number of black and white drivers stopped:
plot(trafficstops$driver_race)
```



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. Before doing that, we're going to pull out the data on race and work with that data, so we're not modifying the working copy of the data frame:

```
race <- trafficstops$driver_race
head(race)
```

```
#> [1] Black Black Black White White White
#> Levels:  Black White
```

```
levels(race)
```

```
#> [1] ""      "Black" "White"
```

```
levels(race)[1] <- "Missing"
levels(race)
```

```
#> [1] "Missing" "Black"   "White"
```

```
head(race)
```

```
#> [1] Black Black Black White White White
#> Levels: Missing Black White
```

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)?

—>

3.6.3 Using `stringsAsFactors=FALSE`

By default, when building or importing a data frame with `read.csv()`, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`.

In most cases, it's preferable to set `stringsAsFactors = FALSE` when importing your data, and converting as a factor only the columns that require this data type.

Compare the output of `str(trafficstops)` when setting `stringsAsFactors = TRUE` (default) and `stringsAsFactors = FALSE`:

```
# Compare the difference between when the data are being read as
# `factor`, and when they are being read as `character`.
trafficstops <- read.csv("data/MS_policing_bw.csv", stringsAsFactors = TRUE)
str(trafficstops)
trafficstops <- read.csv("data/MS_policing_bw.csv", stringsAsFactors = FALSE)
str(trafficstops)
# Convert the column "driver_race" into a factor
trafficstops$driver_race <- factor(trafficstops$driver_race)
```

> Challenge
>
> Can you predict the class for each of the columns in the following example? Check your guesses using `str(country_climate)`: * Are they what you expected? Why? Why not? * What would have been different if we had added `stringsAsFactors = FALSE` to this call? * What would you need to change to ensure that each column had the accurate data type?
>
> ```
> country_climate <- data.frame(
>        country=c("Canada", "Panama", "South Africa", "Australia"),
>        climate=c("cold", "hot", "temperate", "hot/temperate"),
>        temperature=c(10, 30, 18, "15"),
>        northern_hemisphere=c(TRUE, TRUE, FALSE, "FALSE"),
>        has_kangaroo=c(FALSE, FALSE, FALSE, 1)
>        )
> ```

The automatic conversion of data type is sometimes a blessing, sometimes an annoyance. Be aware that it exists, learn the rules, and double check that data you import in R are of the correct type within your data frame. If not, use it to your advantage to detect mistakes that might have been introduced during data entry (a letter in a column that should only contain numbers for instance).

3.7 Dates

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(trafficstops)
```

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

We're going to be using 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".

Start by loading the required package:

```
library(lubridate)
```

```
stop_date <- ymd(trafficstops$stop_date)
str(stop_date) # notice the 'date' class
```

The `ymd` function also has nicely taken care of the fact that the original format of the date column is a factor!

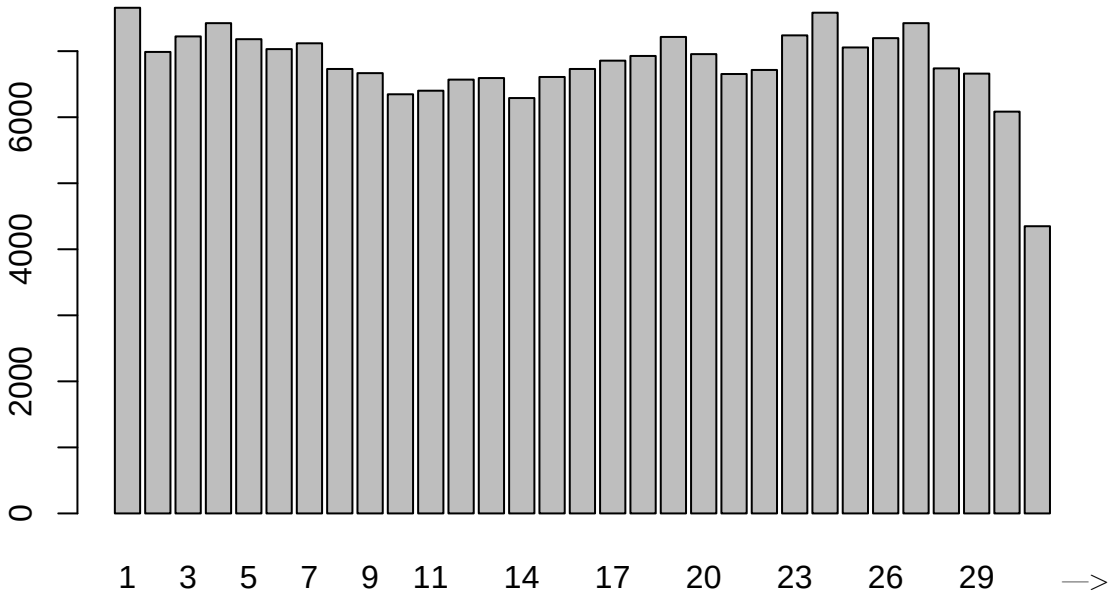
We can now easily extract year, month, and date using the respective functions: `year()`, `month()`, and `day()` like so:

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

Challenge

- Are there more stops in certain months of the year or certain days of the month?

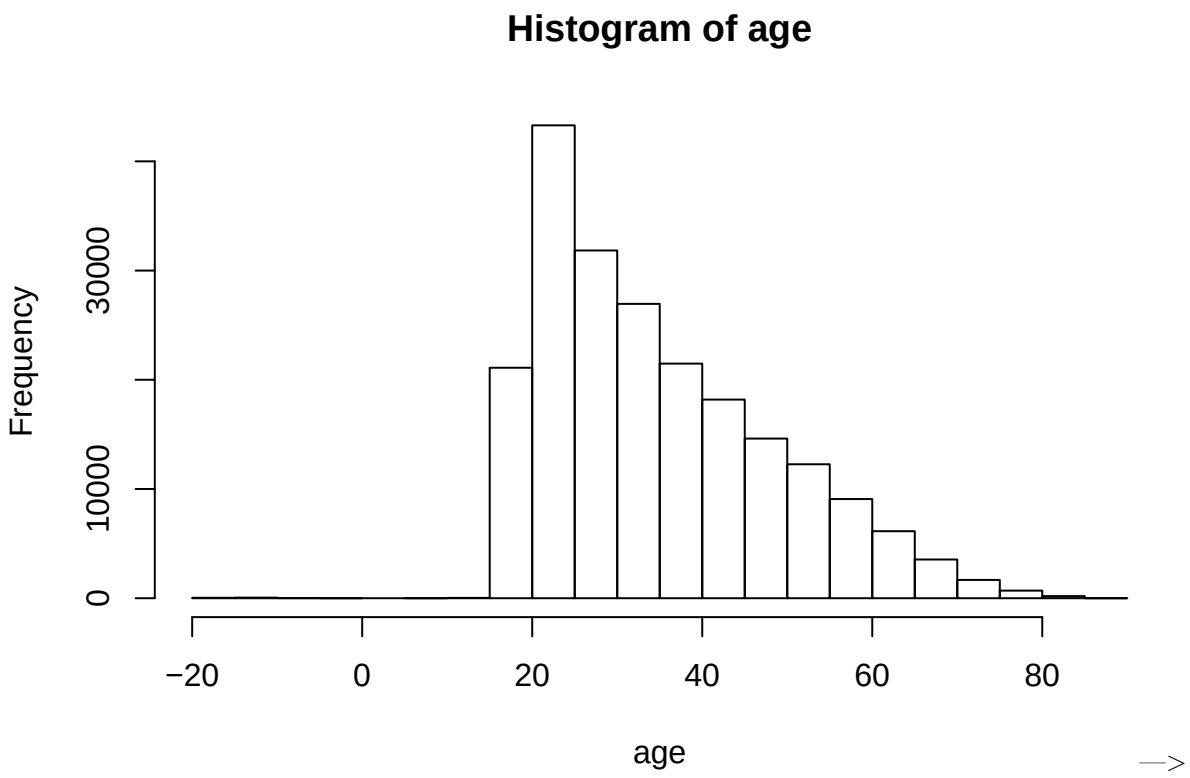
```
plot(factor(day(stop_date)))
```



Challenge

- 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?

```
#or
hist(age)
```



Bibliography