Introduction to R

Carolina Sarmiento, Department of Integrative Biology - USF

Spring, 2021

Session 1 - Feb. 24, 2021

What is R?

As defined in the R website, "R is a free software *environment* for statistical computing and graphics." You can download it here by choosing your preferred mirror, usually one that is close to you. Here is a list of some of the main characteristics of R:

- It is free and open-source.
- R is also a language: "to use it well, one must master its grammar and vocabulary." It is an object-oriented programming language.
- It allows users to define new functions (i.e., add additional functionality) – as opposed to very specific and inflexible tools offered by other data analysis software.
- Provides a wide variety of statistical and graphical techniques, and is highly extensible.
- It has strong graphic capabilities.
- R can be used to perform simple and complex mathematical and statistical calculations on a wide variety of data objects. It can also perform such operations on large data sets.
- R can be used throughout the data analysis process: It helps to gather the data (?), to clean it, to explore it, to model it, to present it, and finally, it helps you to compile the results in an eye-catching and easy to understand reports with R Markdown.

What does *open-source* mean?

Open-source refers to software for which the original source code is made freely available and may be redistributed and modified:

- Provides full access to algorithms and their implementation.
- Allows researchers to explore and expand the methods used to analyze data.
- Promotes reproducible research by providing open and accessible tools.
- Most of R is written in… R! This makes it easier to see what functions are actually doing.

Why use R?

Advantages	**Disadvantages**
1. It's free (and it will not change)	1. Steep learning curve
2. Great graphic capabilities	2. Slow (compared with alternatives such as MATLAB and Python)
3. Very active user community (Good online resources)	3. Flexible syntax (no strict guidelines to follow): You need to maintain proper coding standards to avoid messy and complicated code
4. Forces you to think about analyses (You can get it to do exactly what you need it to do)	4. No commercial support: You have to figure out how to use a function or correct methods to implement it (it can be frustrating)
5. R is stable. It will continue to exist, even if programmers/users stop adding to it	5. No extensive quality control (depends on volunteer programmers)

R Base and Packages

Package: A collection of functions, documentation, and data sets developed by the community. Packages increase the power of R by improving existing *base R* functionalities, or by adding new ones.

R Base: The R `base` package contains (many) basic functions for R to work:

- arithmetic
- input/output
- basic programming support
- etc.

Other packages uses R `base` to do many other things. To date (Feb. 22, 2021), there are 17,178 available packages in CRAN. With that number, I think there should be a package available for most of your current needs in data analysis and data visualization! For example, there are packages for:

- Natural Language Processing (~analysis of speech and language) of Korean texts.
- Extracting weather data from the web.
- Estimate actual evapo-transpiration using land surface energy balance models.
- etc., etc., etc.,

Repositories

Repositories are the places where the packages are located (typically online and accessible to everyone). The three most popular are:

A. **CRAN:** Comprehensive R Archive Network

- Network of servers maintained by the R community.
- R foundation coordinates it.
- For a package to be published here, it needs to pass several tests to ensure it follows CRAN policies.

B. **GitHub:**

- Not R specific.
- Probably the most popular repository for open-source projects.
- Easy to share and collaborate with others.
- No review process associated with it.

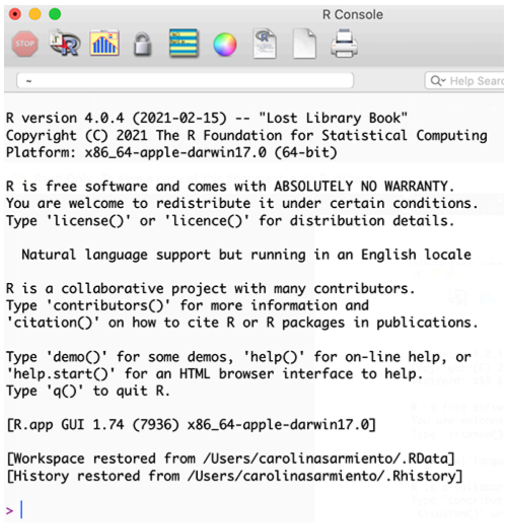
C. **Bioconductor:**

- Topic specific repository, intended for open-source software for bioinformatics.
- It has its own submission and review processes.
- Very active community (several conferences and meetings per year).

Let's start working on R

1. Start by opening R (not RStudio) - this is just for you to know the basics. You can do everything here, just with a text editor and the console, but I encourage you to use RStudio.

When you open R, it opens the **R console**:



```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (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.

[R.app GUI 1.74 (7936) x86_64-apple-darwin17.0]

[Workspace restored from /Users/carolinasarmiento/.RData]
[History restored from /Users/carolinasarmiento/.Rhistory]

> |
```

Figure 1: R Console

When you have your R environment setup, you can start writing your code after the prompt (>)
2. Type a basic operation and hit enter

```
5*5
```

or

```
(50*50)/10
```

While it is possible to work directly on the console, it is highly recommended to use a text editor instead. Even if working in the console is effective and simple, it has its limitations:

- Each time you want to execute a set of commands, you have to re-enter them at the command line.

- Complex commands are potentially subject to errors, and you will need to re-enter the command line to fix them.
- Repeating a set of operations requires re-entering the code stream.
- You can save the console's contents, but it will save everything you have done on a particular session, including the errors, the output, etc. Some things will be useful and some others will not. And finding what is useful can be challenging.

By working in a text editor (i.e., creating an **R script**), you can just keep what you need and save a file that will be more useful and easier to read. A script is simply a text file containing a set of commands and comments. The script can be saved and used later to re-execute the saved commands. The script can also be edited so you can execute a modified version of the commands.

3. Open the R text editor by going to the File menu on the top of the window and open a new document:

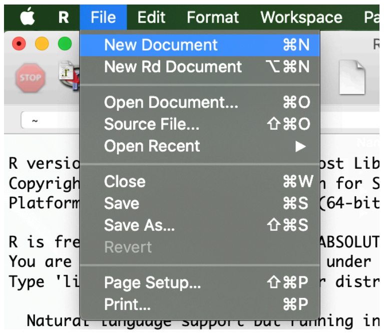
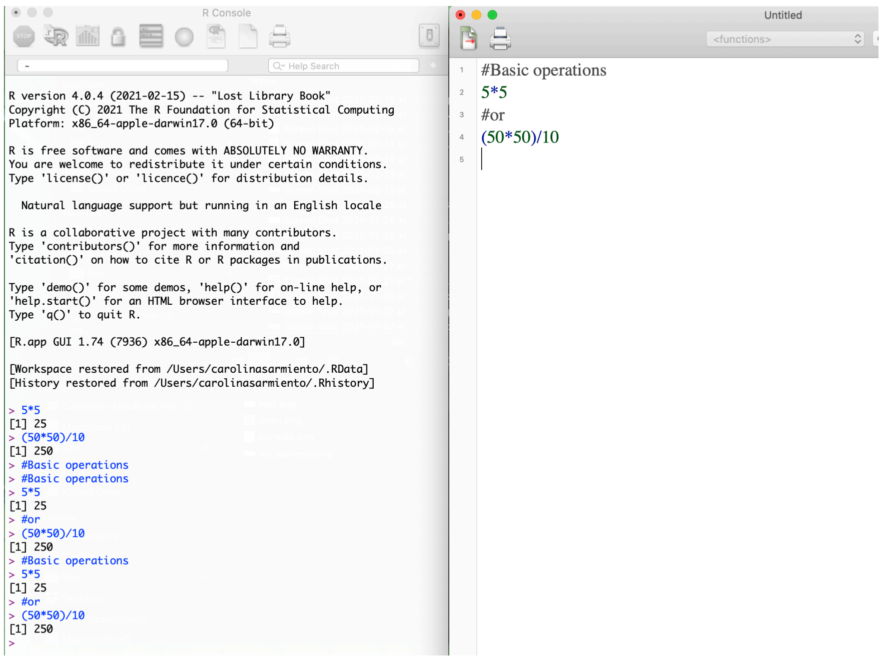


Figure 2: New script file

Now, you have another window, where you will write your **script**. Notice that I used the pound sign(#) to write some comments:



```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (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.

[R.app GUI 1.74 (7936) x86_64-apple-darwin17.0]

[Workspace restored from /Users/carolinasarmiento/.RData]
[History restored from /Users/carolinasarmiento/.Rhistory]

> 5*5
[1] 25
> (50*50)/10
[1] 250
> #Basic operations
> #Basic operations
> 5*5
[1] 25
> #or
> (50*50)/10
[1] 250
> #Basic operations
> 5*5
[1] 25
> #or
> (50*50)/10
[1] 250
>
```

```
#Basic operations
5*5
#or
(50*50)/10
```

Figure 3: Make comments using the # symbol

4. Write a basic operation in the **Text Editor** window and include some comments using the # symbol

5. Execute the line you want by placing your cursor on the line (it can be placed at any point, beginning, middle, or end of the line) and hit Command+Enter (Mac) or Ctrl+Enter (Windows).

Note that you can run a larger piece of your script by highlighting (selecting) all the lines you want to run and then pressing Cmd+Enter (or Ctrl+Enter).

As you see, it is much easier to execute the script when you have the command lines written in the text editor. You can modify them easily and execute them as many times as you need without having to type them again. You can comment and organize your script, which will make things much easier.

Challenge #1: What happens if you remove the # symbol and try to run a 'comment' line?

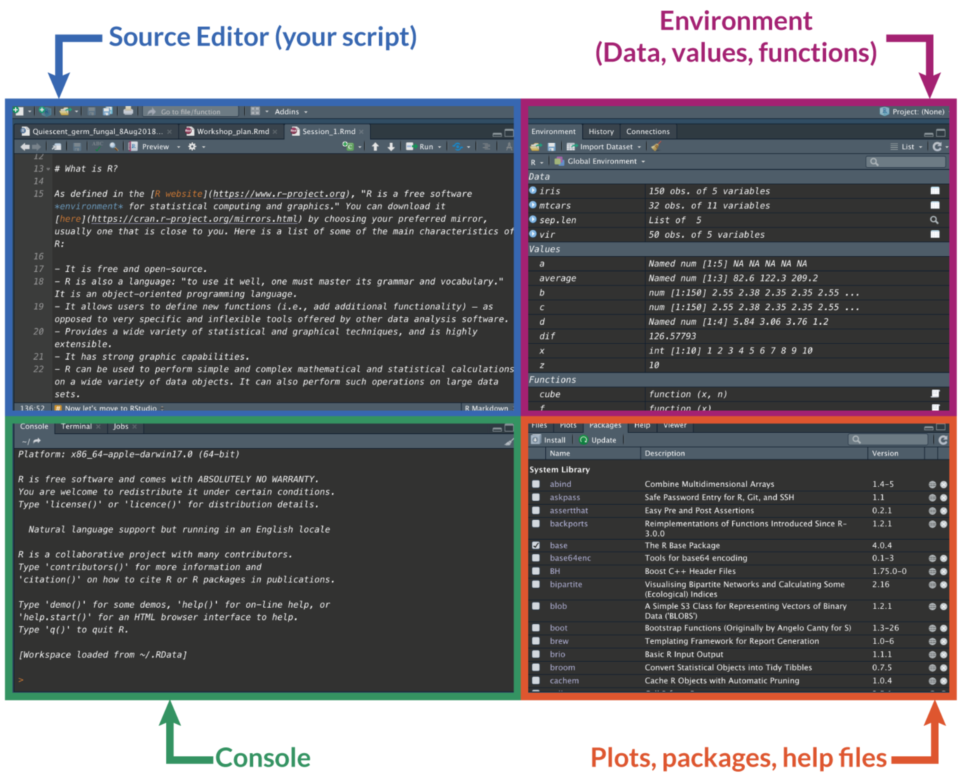
Now, let's move to RStudio

RStudio is an open-source environment for working with R: it integrates the tools you use with R into a single environment.

It includes:

- The console
- Syntax-highlighting editor
- Tools for plotting
- R help
- history
- Tools for debugging
- Workspace management

1. Open RStudio: your desktop should look more or less like this:



```
# What is R?

As defined in the [R website](https://www.r-project.org), "R is a free software
*environment* for statistical computing and graphics." You can download it
[here](https://cran.r-project.org/mirrors.html) by choosing your preferred mirror,
usually one that is close to you. Here is a list of some of the main characteristics of
R:

- It is free and open-source.
- R is also a language: "to use it well, one must master its grammar and vocabulary."
It is an object-oriented programming language.
- It allows users to define new functions (i.e., add additional functionality) — as
opposed to very specific and inflexible tools offered by other data analysis software.
- Provides a wide variety of statistical and graphical techniques, and is highly
extensible.
- It has strong graphic capabilities.
- R can be used to perform simple and complex mathematical and statistical calculations
on a wide variety of data objects. It can also perform such operations on large data
sets.
```

```
Data
iris          150 obs. of 5 variables
mtcars        32 obs. of 11 variables
sep.len       List of  5
vir           50 obs. of 5 variables
Values
a             Named num [1:5] NA NA NA NA NA
average       Named num [1:3] 82.6 122.3 209.2
b             num [1:150] 2.55 2.38 2.35 2.35 2.55 ...
c             num [1:150] 2.55 2.38 2.35 2.35 2.55 ...
d             Named num [1:4] 5.84 3.06 3.76 1.2
dif           126.57793
x             int [1:10] 1 2 3 4 5 6 7 8 9 10
z             10
Functions
cube          function (x, n)
f             function (x)
```

```
Platform: x86_64-apple-darwin17.0 (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.

[Workspace loaded from ~/.RData]

>
```

	Name	Description	Version
	System Library		
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.2.1
✓	base	The R Base Package	4.0.4
☐	base64enc	Tools for base64 encoding	0.1-3
☐	BH	Boost C++ Header Files	1.75.0-0
☐	bipartite	Visualising Bipartite Networks and Calculating Some (Ecological) Indices	2.16
☐	blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.1
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-26
☐	brew	Templating Framework for Report Generation	1.0-6
☐	brio	Basic R Input Output	1.1.1
☐	broom	Convert Statistical Objects into Tidy Tibbles	0.7.5
☐	cachem	Cache R Objects with Automatic Pruning	1.0.4

*Note: if it is the first time you are working on RStudio, or if you closed all the files before quitting RStudio the last time you used it, it is possible that you don't have a Source Editor window opened. In that case, go to **File > New File > R Script** to open a new source editor (new script).*

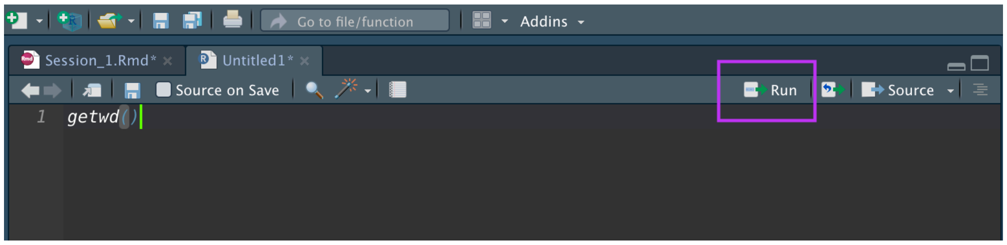
Before starting, let's set the Working Directory (WD):

R is always pointed at a directory (or folder) on your computer (**working directory**). R will use this folder to read and save files. For example, if you ask R to import a dataset from a .csv file, it will assume that the file is inside of your working directory.

You can find out which directory R is currently working on by running the getwd (get working directory) function.

2. Find out the current working directory by typing `getwd()` in your text (or source) editor window and then press Cmd (or Ctrl) + Enter to execute the line

Tip: you can also run the line (or selected lines) by clicking on the 'Run button' at the top of your text editor window:



```
getwd()
```

Figure 4: Run current line or selection

Before setting a new working directory, I strongly suggest creating a dedicated folder to R for each of the projects you work on, to keep all the files organized. Here is an example of how I work:

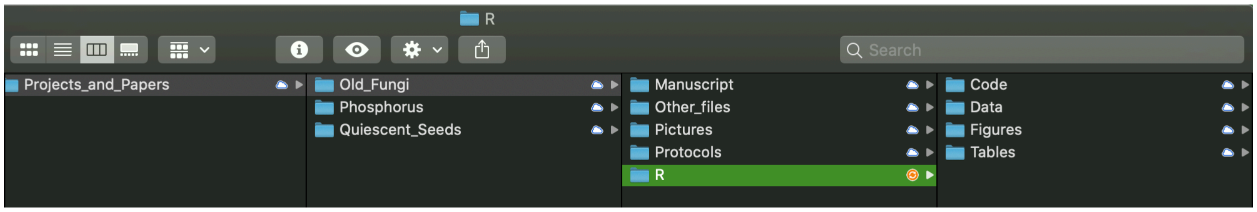
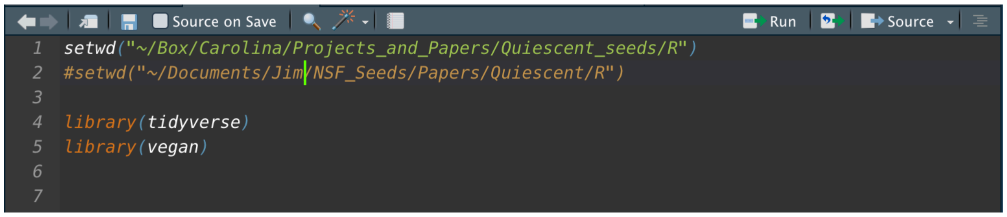


Figure 5: View of my folders

I always have sub-folders to keep my scripts, data, figures, etc. You can, of course, use a different approach but it is a good idea to be careful when choosing a working directory, so it is easy for you to remember where your R files are stored.

3. Now change your working directory. For that go to the top menu **Session > Set Working Directory > Choose Directory** and choose the R folder you just created.

Tip: Copy the line that appeared on your console - without the prompt: `setwd(...)` and paste it at the beginning of your script. Next time you open your script, you will only have to execute that line. Often my scripts look like this:



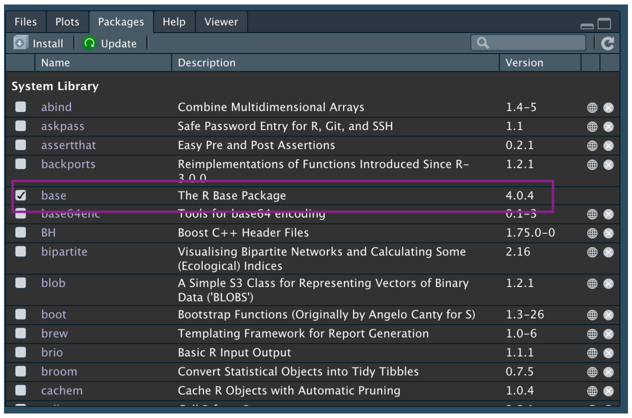
```r
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

library(tidyverse)
library(vegan)


```

Note that when I share the code with collaborators, we include a line with the path to their WD, so it is easier for everyone to work on the same file.

Now, let's learn how to install and load a package

Tip: From now on, always work on your text editor file, either if that is setting your working directory, loading a package, making a comment, or anything else, I encourage you to avoid working directly on the console - it is usually not helpful and it is not faster! For the remainder of this guide, I will show how examples will look on the console, but please work on your script and execute the lines from there. We

will install the package **dslabs** but before that, please take a moment to check your Packages tab on the bottom-right window of RStudio:



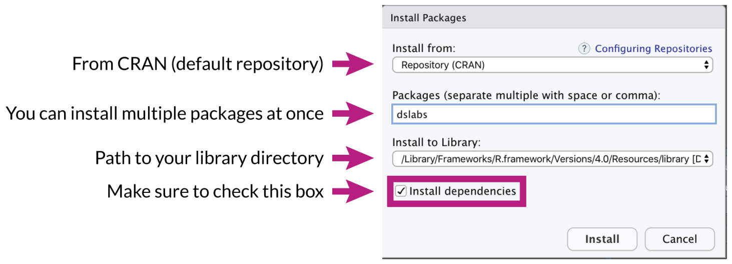
System Library

	Name	Description	Version
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.2.1
✓	base	The R Base Package	4.0.4
☐	base64enc	Tools for base64 encoding	0.1-3
☐	BH	Boost C++ Header Files	1.75.0-0
☐	bipartite	Visualising Bipartite Networks and Calculating Some (Ecological) Indices	2.16
☐	blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.1
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-26
☐	brew	Templating Framework for Report Generation	1.0-6
☐	brio	Basic R Input Output	1.1.1
☐	broom	Convert Statistical Objects into Tidy Tibbles	0.7.5
☐	cachem	Cache R Objects with Automatic Pruning	1.0.4

Figure 6: Package list on your System Library

This tab shows a list of the packages you have already installed in your **System Library.** The library is the directory (folder) where the packages are installed. Note that the R `base` package is already loaded, but in my case, I have several packages that I have installed, but they are not loaded.

4. Click on the *Install* button on the top-left corner of the *Packages* tab and enter `dslabs` and click *install*:


From CRAN (default repository)
You can install multiple packages at once
Path to your library directory
Make sure to check this box
Configuring Repositories
Install
Cancel

Install Packages

Install from: Repository (CRAN)

Packages (separate multiple with space or comma): dslabs

Install to Library: /Library/Frameworks/R.framework/Versions/4.0/Resources/library [D

✓ Install dependencies

Figure 7: Installing the package `dslabs`

What happened on your console? Yes! you can do everything by command line - RStudio only makes our life a little easier.

After a while, you can end up with a collection of many packages. If R loaded all of them at the beginning of each session, that would take a lot of memory and time. So, before you can use a package, you have to load it into R by using the `library()` function. Or you can also check the box next to the package, on your package list.

5. Load the package `dslabs`:

```
library(dslabs)
```

Please **note** that the input of `install.packages()` is a character vector (the package name) and requires the name to be in quotes, while `library()` accepts either characters or names and makes it possible for you to write the name of the package without quotes.

Challenge #2: What happens if you try to install a package by typing the function `install.packages()` but you forget to use the quotation marks?

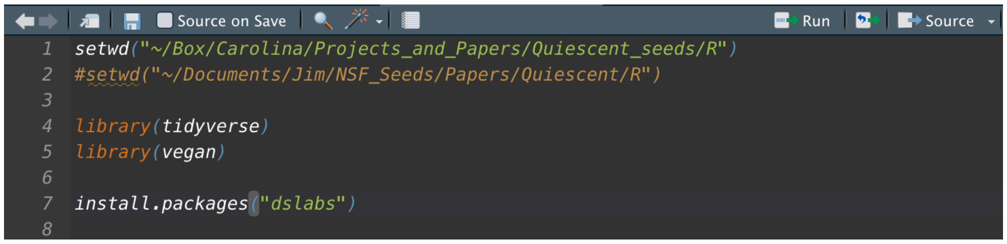
Challenge #3: What happens if you forget to close the parenthesis at the end of a function (e.g. `library()`)?

(Some) RStudio Tips and Tools:

As you have seen, RStudio has many features and coding tools to make our work in R a little easier. Here I will show you some others that are really useful - I know that some of them won't make much sense now, but I hope that by knowing they exist, you will start using them as you go:

Syntax highlighting

R Studio's text editor recognizes some elements of your script and colors them differently, which is helpful to code effectively. It also highlights brackets and parenthesis, and signals when something is incomplete. Here is an example:



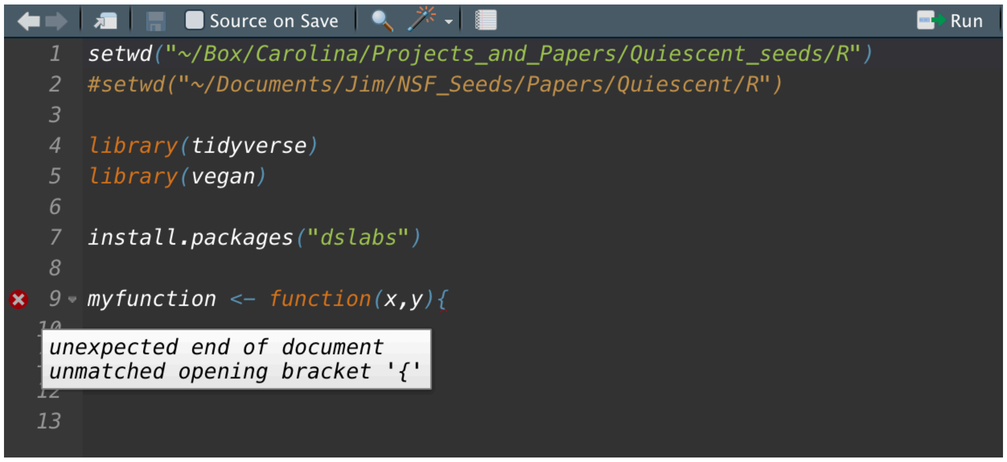
```
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

library(tidyverse)
library(vegan)

install.packages("dslabs")
```

Figure 8: RStudio syntax highlighting

Note that in my case, some functions are in white (e.g. `setwd()`, `install.packages()`), some others are in orange (e.g. `library()` but don't ask me why!), comments are in yellow, character strings are in green, parenthesis are in blue. You can change this by clicking on the menu **RStudio** > **Preferences...** > **Appearance** (Mac) or **Tools** > **Global Options** > **Appearance** (Windows).

The editor also highlights the closing parenthesis if you place the cursor on the opening one, and vice versa.

The editor also signals when there is an error in the coding, which is really useful:



```
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

library(tidyverse)
library(vegan)

install.packages("dslabs")

myfunction <- function(x,y){
```

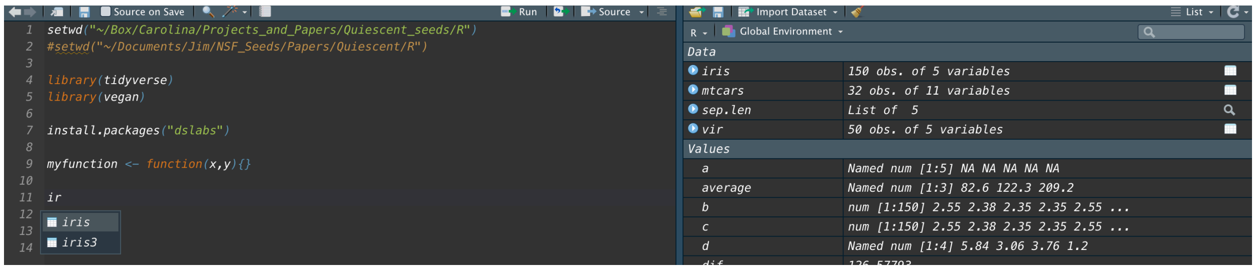
```
unexpected end of document
unmatched opening bracket '{'
```

Figure 9: Coding error

If you place your mouse cursor over the “x” on the left, it shows you the error, in this case, it’s missing the closing bracket.

Code completion

- RStudio supports the automatic completion of code using the **Tab** key. For example, if you have an object named iris in your workspace you can type ir and then Tab and RStudio will automatically complete the full name of the object (you will be able to practice this later).

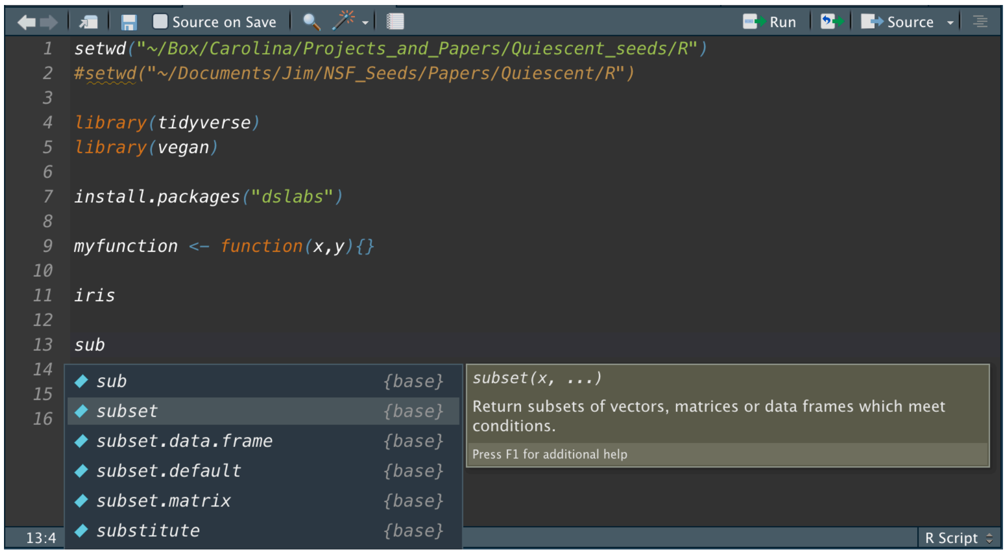


```
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

library(tidyverse)
library(vegan)

install.packages("dslabs")

myfunction <- function(x,y){}

ir
iris
iris3
```

```
Data
iris      150 obs. of 5 variables
mtcars    32 obs. of 11 variables
sep.len   List of  5
vir       50 obs. of 5 variables
Values
a         Named num [1:5] NA NA NA NA NA
average   Named num [1:3] 82.6 122.3 209.2
b         num [1:150] 2.55 2.38 2.35 2.35 2.55 ...
c         num [1:150] 2.55 2.38 2.35 2.35 2.55 ...
d         Named num [1:4] 5.84 3.06 3.76 1.2
dif       126.57793
```

Figure 10: Code completion using Tab to select the object iris

- The code completion feature also provides inline help for functions whenever possible. For example, if you typed sub then pressed Tab you would see:



```
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

library(tidyverse)
library(vegan)

install.packages("dslabs")

myfunction <- function(x,y){}

iris

sub
```

subset(x, ...)

Return subsets of vectors, matrices or data frames which meet conditions.

Press F1 for additional help

Figure 11: Code completion using Tab to select a function

Let's practice this:

- Type sub
- Press *Tab*
- Select subset and press *Enter*
- Press *Tab* again

Smart indentation:

Indenting your code makes it a lot easier to read.

Tip: Command+I (Mac) or Ctrl+I (Windows) will indent a selected chunk of code.

Challenge #4: Paste the following code in your RStudio source editor

```
# When pasting, make sure you see four (4) lines as shown here:
for(i in 1:10) {
if(i %% 2 == 0)
print(paste(i, "is even"))
}
```

Is it indented? If not, select it and hit Command+I (Mac) or Ctrl+I (Windows)

Comments:

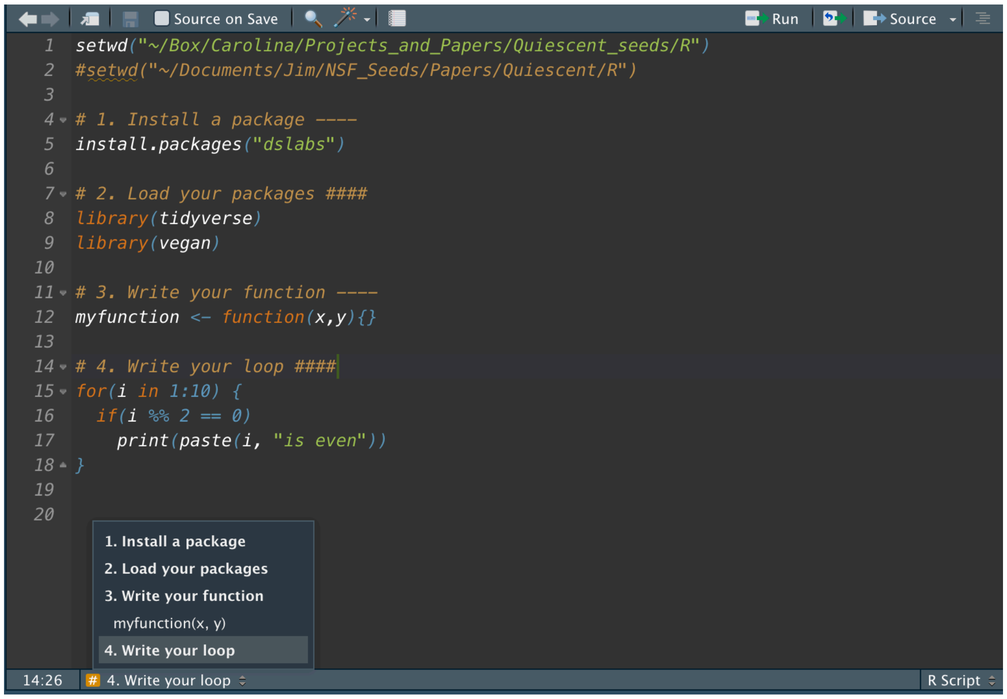
- Comment your code. Always. Your collaborators and future-you will be very grateful. Comments start with # followed by space and text of the comment.

- To comment/un-comment a selected chunk, use Command+Shift+C (or Ctrl+Shift+C).

Challenge #5: Select the previous chunk of code and make it a comment

Code sections:

This is a very useful tool - you can add sections to your script by typing four dashes (- - - - ; no spaces) or pound symbols (####) after the Section title. This is really useful when you have long scripts, making a lot easier to navigate them (if you use this tool in a good way - if you add too many sections it can become useless.) You can navigate the sections by clicking on the 'two-arrows' symbol at the bottom-left of your text editor window. The following is just an example with a very short code (so here, making sections is unnecessary/useless)



```r
setwd("~/Box/Carolina/Projects_and_Papers/Quiescent_seeds/R")
#setwd("~/Documents/Jim/NSF_Seeds/Papers/Quiescent/R")

# 1. Install a package ----
install.packages("dslabs")

# 2. Load your packages ####
library(tidyverse)
library(vegan)

# 3. Write your function ----
myfunction <- function(x,y){}

# 4. Write your loop ####
for(i in 1:10) {
  if(i %% 2 == 0)
    print(paste(i, "is even"))
}
```

Figure 12: Code Sections

Cheat sheets:

RStudio has a good collection of Cheatsheets that you can find here.

Challenge #6: go to the Cheatsheet page (above) and find the "Contributed Cheatsheets", download the "Base R" cheatseet and give it a good read!

Some take-home messages thus far:

- Using a script (as opposed to typing your instructions directly on the console) is highly recommended
- RStudio makes easier some common tasks.
- The pound symbol (#) is useful to comment on your code - R "ignores" the text preceded by #. And I strongly suggest using it (as much as you can!)

Getting help

There are several ways you can find help in R:

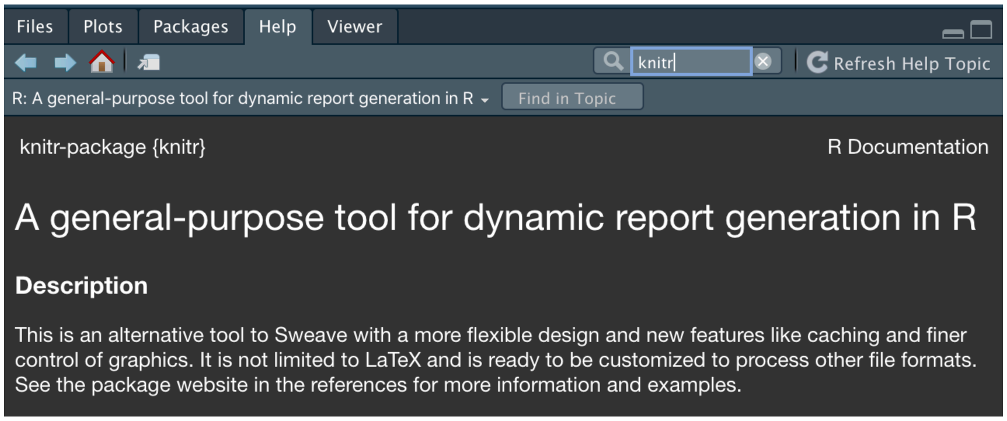
- If you want to explore a package that is installed, click on its name (in the Packages tab) - that shows you the Description file and the list of elements in the package (usually a collection of functions and datasets), you can click on those too if you want to access the documentation file for that particular element.

Challenge #7: What is the purpose of the `dslabs` package and who is the author of the package?

- To access the documentation file for a function, try:

```
help(anova)
# or
help("anova")
# or
?anova
```

- In RStudio: go to the Help tab in the bottom-right window of RStudio and type the function name, dataset name, or package in the search pane (magnifying lens symbol) to get help. Note that this is not always easy to follow.



knitr-package {knitr} R Documentation

A general-purpose tool for dynamic report generation in R

Description

This is an alternative tool to Sweave with a more flexible design and new features like caching and finer control of graphics. It is not limited to LaTeX and is ready to be customized to process other file formats. See the package website in the references for more information and examples.

Figure 13: Getting help in RStudio

- Online: There is a ton of online help, that sometimes is challenging to find good help. Below are my favorite ways to find online help:

  a. Try searching on Rseek.
  b. Try Stackoverflow using the [r] tag.
  c. Ask Google by starting with R: before your question (e.g. try R: how to subset). Usually, the first hits are the best ones (I tend to favor questions solved by/on Stackoverflow).

R is an object-oriented language

That means that variables, data, functions, results, etc., are stored in the active memory of the computer in the form of **objects**, which have a **name**.

You create the objects and you assign the names!

Objects are created with the "assignment" operator to assign values to an object, which is the arrow with a minus sign "<-".

Tip: Useful keyboard shortcut Option + - (Mac) ; Alt + - (Windows)

The way you assign a name to an object is usually:

name <- Object

- Names of objects MUST start with a letter and can include letters, digits, dots (.) and underscores(_).
- R is case sensitive, so x and X are different objects. Remember, spelling always counts and spelling mistakes are among the most frequent "bugs" in R code.
- Try to assign short and meaningful names.
- If you assign the same name to a different object, R will overwrite it. Be careful, it is very easy to replace existing objects.

1. Let's create some objects:

A few things to have in mind:

- If your object is a string of characters (a sentence, for example) you usually have to use quotes. Quotes are not needed when the elements on your objects are numbers.
- If your object is a vector (i.e., a group of data elements) you have to use the function `c()` and use a comma to separate the *components* of your vector

Challenge #8:

Remember to work on your text editor, always.

- Create an object called 'name' that stores your name on it. Can you see it on your 'Environment now?
- Create an object (named a) with a sequence of numbers from 1 to 5.
- Create an object (named b) with 5 other numbers.
- Create an object (named c) that contains the first 5 letters of the alphabet.
- Look at the help file of the function `c ()` What does the 'c' means?
- Can you combine letters and numbers in the same object?

2. Let's make some basic operations with your objects (e.g. a+b) and assign that sum another name:

- Can you sum a + c?

3. Now, let's assign the data set 'iris' to an object named 'mydata'.

- Check out the documentation file (i.e., help) for iris.

Challenge #9: What package is it part of?

Read and Save data

Importing/reading data

There are several ways to import data on R:

A. The one that I use the most and recommend: Importing a .csv file (created in Excel):

```
mydata <- read.csv("path/to/file.csv", header = TRUE)
```

- This imports a .csv file and recognizes the first row as column names.

B. Importing a text (.txt) file:

```
mydata2 <- read.table("path/to/file.txt", sep= "\t", h = T)
```

Imports a tab-delimited text file and recognizes the first row as the column names

C. Importing a file from anywhere in your computer:

```
mydata3 <- read.csv(file.choose( ))
```

Challenge #10: Import the spiders dataset (uploaded in Teams) as a .csv file

Exporting/saving data

To save data, you will have to export your objects (where the data are stored):

```
write.table(object, "path/to/object", sep=",")

write.csv(object,"path/to/object.csv")

# Example: write.csv(mydata2, "Data/New_iris.csv") - will save the object 'mydata2'
# to a sub-folder called 'Data' inside my WD.
```

Session 2 - March 3, 2021

Objects can be modifyed with `Operators` and `Functions`

As we already saw, R is an **object-oriented language:** Objects allow the user to create symbolic representations of simple or complex data sets or other information. Objects are stored in the active memory of the computer.

Operators

R is a language rich in built-in operators. An operator is a symbol that, as its name implies, is used to perform operations on objects (values and variables). There are several kinds of operators:

1. *Arithmetic operators:* used to perform simple mathematical operations on numeric values and vectors.

Arithmetic Operators	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponent

```r
# Try these operations on numeric values

x <- 6
y <- 2

x^y
x/3
x*(x+y)
```

Note that all the basic arithmetic operators can be performed on pairs of vectors. Each operation is performed in an element-by-element manner:

```r
v1 <- c(11,12,13,14,15)
v2 <- c(1,2,3,4,5)

# Try:
v1 + v2
v1 ^ v2
```

2. *Comparison Operators:* Used to compare two objects (values or vectors).

Operator	Description
$<$	less than
$>$	greater than

Operator	Description
<=	less than or equal to
>=	greater than or equal to
==	exactly equal to
!=	not equal to

```r
# Try:

x > y
x != y
v1 < v2
```

3. *Logical Operators:* Used to carry out Boolean operations (combine o exclude elements e.g. AND, OR, NOT)

Operator	Description
&	Returns True if both statements are true (AND)
|	Returns True if one of the statements is true (OR)
!	Reverses the result, returns False if the result is true (NOT)

```r
x <- 2
y <- -2

# and
x > 0 & y < 0

# or
x > 0 | y > 0

# not
!(x > 0 & y < 0)

v1 < x & y < v2

v1 > x | y > v2
```

4. Others

Operator	Description
%in%	Returns True if a value is present in the vector
<-	Assignment
:	Generates a number sequence from a to b

```r
v <- c("red", "green", "blue")

"red" %in% v
"black" %in% v

x <- 3
x

n <- c(50:100)
```

Functions

Almost everything in R is done through functions. A function is a set of statements organized together to perform a specific task. R has a large number of built-in functions and the user can create their own functions.

The different parts of a function are:

Function Name – This is the actual name of the function. It is stored in R environment as an **object** with this name.

Arguments – When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

function.name (arg1, arg2, ...)

Some common **built-in functions** (from the R `base` package):

Function	Description
abs(x)	absolute value
sqr(x)	square root
round(x, digits=n)	round x to *n* decimal places
log(x)	natural logarithm
exp(x)	e^x
mean(x, na.rm=TRUE)	mean of object x, remove NAs
median(x)	median
c(x,y,z)	combine values into a vector
function(x)	function definition

User-written functions: one of the great strengths of R is the user's ability to add functions. We will learn how to create basic functions on Session 4 (March 17).

Some other useful Functions: It is impossible to try and provide a comprehensive list of the functions you will use the most. Here is a brief list of my favorites and/or most used:

Functions for manipulating data frames or other objects:

- str() - tells you the structure of an object.
- ncol() - tells you the number of columns.
- nrow() - tells you the number of rows.
- head() - shows you the top 6 rows.
- tail() - shows you the last 6 rows.
- names() - shows you the column names and allows you to change them.

- colSums() - calculate the sum of a column or columns in a data frame.
- colMeans() - calculate the mean of a column or columns in a data frame.

General functions for creating or manipulating objects:

- seq() - Create a sequence of numbers.
- rep() - Repeat something a set number of times.
- rbind() - Bind two vectors together as rows.
- cbind() - Bind two vectors togethr as columns
- paste() - Stick two bits of text together.

Mathematical functions

- mean() - Calculate the mean of a vector of numbers.
- sum() - Calculate the sum of a vector of numbers.
- max() - Find the max value in a vector.
- min() - Find the min value in a vector.
- range() - Gives you the min and max value (range) of a vector.
- sd() - Calculate the standard deviation of a vector.

Challenge #1: use the functions `rnorm()`, `hist()`, and `abline()` to:

1. Make a vector with a 100 observations, normal distribution, mean equal to 0 and standard deviation equal to 2.
2. Make a histogram of that vector - with 25 bars.
3. Change the color of the bars.
4. Change the main title and the x and y axis labels.
5. Add a green vertical line to denote the mean value.
6. Save the histogram as a PDF.

Hint: use the help tab in RStudio to find out how to do the task.

Object properties

1. Mode: How objects are stored in R, it refers to their basic structure (character, numeric, logical, complex, function). You can find out the mode of an object using the function `mode()`

```
# Try
z <- c("a","b","c")
mode(z)

mode(iris$Sepal.Length)
```

2. Class: How objects are treated by functions (character, factor, numeric, matrix, data.frame, etc.)

```
#Try
class(iris$Species)
class(rnorm)
class(iris)
```

3. Length: Number of elements of an object

```
#Try
length(z)
length(iris$Sepal.Width)
```

Data structure

Data objects in R can be of different types:

Object	Modes	Several modes possible?
vector	numeric, character, logical or complex	No
array	numeric, character, logical or complex	No
matrix	numeric, character, logical or complex	No
data frame	numeric, character, logical or complex	Yes
list	numeric, character, logical, complex, function, expression…	Yes

A **vector** has only a single dimension, the length of the vector is defined by the number of elements in the vector. All of these must be the same mode.

We can ask R what kind of object it is by using the structure or the `str()` function:

```
v <- rnorm(100,0,1)
str(v)
```

which tells us it is a numeric vector with 100 elements and it gives us the first five elements. We can also ask R directly if v is a vector:

```
is.vector(v)
```

A **matrix** is a collection of elements of the same data type or mode (numeric, character, or logical) arranged into a fixed number of rows and columns (2-dimensional).

```
a <- c(1:10)
b <- c(11:20)
m <- cbind(a,b) # the function cbind() combines two objects by columns

is.matrix(m)
```

An **array** is similar to a matrix but it can hold multi-dimensional data.

A **data frame** is perhaps the most useful (or used) of the objects. Data frames are a table composed of several vectors all of the same length, but the data in those vectors can be of different modes. In R, we convert a matrix into a data frame with the function `as.data.frame()` and the opposite is possible too (`as.matrix()`).

```
c <- rep(TRUE,10)
d <- rep(c("a","b"),5)
my.df <- data.frame(a,b,c,d)
str(my.df)
```

Finally, **lists** are objects that can contain any type of object (vectors, matrices, data frames, even other lists). Lists are often objects that are generated (output) by some analyses/functions. Although most of the time you will be dealing with vectors, matrices and data frames, it is important to know lists exist!

Lets say we want to create a list that has both my.df and the vector v on it:

```
mylist <- list(my.df,v)
str(mylist)
mylist
```

Accessing the values of an object

The indexing system:

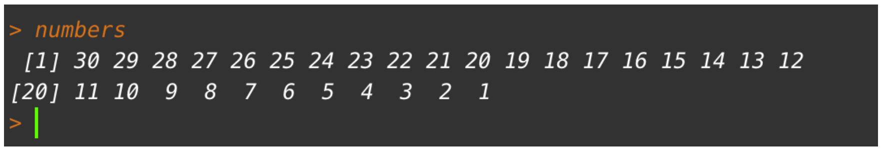
There are multiple ways to access or replace values in vectors or other data structures. The most common approach is to use "indexing". The square brackets ([ ]) illustrate another strong point of R: they can be used to extract a value from a vector, a matrix, a data frame or a list.

In vectors:

Each element of a vector can be extracted by placing its position in brackets

```
#Let's consider our vector a, from above:
a <- c(1:10)
a[5] # Extract the fifth element of vector a

# Now, let's try the same with our vector d
d <- rep(c("a","b"),5)
d[5]

# Note that we'll get a missing value if we try to extract an element by indexing
# outside of the length of the vector:
numbers <- c(30:1)
numbers[40]
```

Note that every time R shows you a vector, it displays a number such as [1] in front of the output (it actually shows a number in brackets in every line, but it depends on the width of your console). That number is called the index of that value - in my case, the first value of vector *numbers* is 30 and the twentieth value in that vector is 11.



```
> numbers
 [1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12
[20] 11 10  9  8  7  6  5  4  3  2  1
> 
```

The bracket function can also take vectors as arguments. If you want to select more than one number, you can simply provide a vector of indices as an argument inside the brackets, like so:

```
numbers[c(5,11,3)] # Extracts the 5th, 11th and 3rd elements of the vector numbers
```

R returns a vector with the numbers in the order you asked for. So, you can use the indices to order the values the way you want.

You also can store the indices you want to retrieve in another vector and give that vector as an argument, as in the following example:

```r
indices <- c(5,11,3)
numbers[indices]
```

You can use indices to drop values from a vector as well. If you want all the numbers except for the third value, you can do that with the following code:

```r
numbers[-3]
```

The indexing system can also be used to replace values:

```r
numbers[indices] <- 0 #this will replace the 5th, 11th and 3rd elements of the vector
# numbers by 0

# Let's check it out
numbers

# or try
newd <- d #let's first create a new object, so we can keep d as is.
newd[6] <- NA # and now replace the 6th element by NA
newd
```

We can also use some of the operators we know to index vectors

```r
# Let's use the vector a, from above and extract the elements that are greater than
# or equal to 5
a[a >= 5]

# This works by first constructing a logical vector
a >= 5
# and then using that to return elements where the logical is TRUE:
a[a >= 5]

# We can use an exclamation point (!) to negate the logical and thus return an opposite
# set of vector elements:
a[!a == 5]
```

In matrices and data frames:

Matrices and data frames have similar structure - and you can access its elements as you do with vectors, but taking into account the position in the rows and in the columns: [**rows, columns**]

Let's use the BCI dataset: it represents the counts of 225 plant species (stored in the first row) in 50 1-hectare plots on Barro Colorado Island (BCI) in Panama. For this, you have to install and load the package `vegan`

```r
A <- BCI # to make it easier (for me, less typing), I've just stored this data set in an
# object named "A"
str(A)

# Pay particular attention to the placement of the comma - remember that it separates
# rows and columns
```

```
A[3,] # shows 3rd row - all columns
A[,4] # shows 4th column - all rows
A[3,4] # shows element in row 3, column 4
A[c(1,9),] # shows row 1 and 9 - all columns
A[,-1] # shows all of A, except column 1
A[-c(1,9),] # shows all of A, except rows 1 and 9
A[1:10,5:8] # shows first 10 rows and columns 5-8
A[,4:1] # shows first 4 columns of A, in opposite order
```

Note that you can also index with names:

```
# Let's create a small data frame
subject <- c(1,2,3,4)
sex <- c("M", "F", "F", "M")
size <- c(7,6,9,11)

mydf2 <- data.frame(subject, sex, size)

# There are several ways to get what you want:
# Get the elements on rows 1 and 2 of column "size"
mydf2[1:2, "size"]
# or
mydf2[c(1,2), 3]

#Get rows 3 and 4, and only the columns named "sex" and "size"
mydf2[3:4, c("sex","size")]
# or
mydf2[c(3,4), c(2,3)]
```

You can also access the columns of a data frame using the $ symbol and the name of the column:

```
mydf2$size
# Note that RStudio recognizes the name of your data frame when you type the dollar
# symbol, it shows you the 'available columns' so you can choose them by clicking on the
# name
```

We can also use some of the operators we already know, to select some elements in our data set:

```
females <- mydf2[mydf2$sex=="F",] # extracts the rows that meet the condition 'sex equal
# female' and all columns and stores them in an object named 'females'

mysub <- mydf2[mydf2$subject < 3, ]

mydata <- iris # Here, let's use the iris dataset
myflowers <- mydata[mydata$Sepal.Length > 6 & mydata$Sepal.Width < 4, ] #selects all the
# rows with sepal lengths > 6cm and sepal widths < 4cm and stores them in an object named
# 'myflowers'
```

In lists:

Objects in a list can also be extracted with the indexing system. You have to use double brackets for this ([[ ]])

```r
# Let's use the list we created above. Remember that we have two elements on this list: a
# data frame with 10 rows and 4 columns and a vector with a 100 numbers:
str(mylist)

mylist[[2]] # Extracts the 2nd component of the list

# We can name the objects in the list
names(mylist) <- c("mydf3","myvector")

# and also use the names to access the elements of the list
mylist[["myvector"]] # shows the component named myvector in list
```

You can also use the dollar sign to access the elements of a list:

```r
mylist$myvector

# or

mylist$mydf3$b
```

Subsetting data

The powerful indexing features of R can be used to select and exclude variables and observations. The following code demonstrate ways to keep or delete variables and observations from a dataset.

Select (keep) variables

```r
# select variables v1, v2, v3
myvars <- c("v1", "v2", "v3")
newdata <- mydata[myvars]

# select 1st and 5th thru 10th variables
newdata <- mydata[c(1,5:10)]
```

Challenge #2: Use the BCI dataset to select the counts that correspond to the following species: *Cecropia insignis*, *Colubrina glandulosa*, and *Ceiba pentandra*. Store those in an object called 'my.species"

Exclude (drop) variables

```r
# exclude variables v1, v2, v3
myvars <- names(mydata) %in% c("v1", "v2", "v3")
newdata <- mydata[!myvars]

# exclude 3rd and 5th variable
newdata <- mydata[c(-3,-5)]
```

Challenge #3: Use the BCI dataset and make a new object where you exclude the following species: *Vochysia ferruginea*, *Trema micrantha*, *Tabebuia rosea*, *Simarouba amara* and *Pourouma bicolor*

Selecting observations

We already saw how to select observations (rows) based on conditions and using the indexing system.

Challenge #4: Let's install and load the package `dslabs` and use the dataset "gapminder" to:

- Create a new object that shows the life expectancy and fertility data for African countries in 1960

Selection using the Subset Function

The `subset( )` function is an easy way to select variables and observations. In the following example, using the gapminder data set, we select the gdp and fertility rates of all the countries in the Americas after the year 2000:

```
myd <- gapminder
americas <- subset(myd, continent == "Americas" & year >=2000, select=c(country,year, fertility, gdp))
```

Sorting data

To sort a data frame in R, use the `order( )` function. By default, sorting is ASCENDING. If you add a minus sign to the sorting variable, it indicates DESCENDING order.

Challenge #5: Let's use the "mtcars" dataset and the function `order ( )` to sort it by Miles per gallon (ascending) and number of cylinders (descending).

Hint: this involves the indexing system too.

Merging data

Adding Columns:

To merge two data frames (datasets) horizontally, use the `merge ( )` function. In most cases, you join two data frames by one or more common key variables (i.e., an inner join).

```
# Let's use our previous object mydf2 and create a new data frame to join:
mydf4 <- data.frame(subject=c(4,3,2,1), weight = c(200,150,110,130))
mydf4

# To merge both, use the function merge ():
mydf5 <- merge(mydf2, mydf4, by="subject")
```

Adding Rows:

To join two data frames (datasets) vertically, we can use the `rbind ()` function. The two data frames must have the same variables, but they do not have to be in the same order.

```
#Let's create a new data frame to join with mydf4
mydf6 <- data.frame(weight = c(180,150,120,140), subject=c(5,6,7,8))

mydf7 <- rbind(mydf4,mydf6)
```

Challenge #6: sort mydf7 by subject.

Adding new variables

You can use the assignment operator and the dollar symbol to create a new variable in the data set

```r
# Using the mtcars data set, let's add a new variable that explores the relationship
# between the miles per gallon and the gross horsepower

cars <- mtcars
cars$mpg.hp <- cars$mpg/cars$hp
```

Recoding variables

You can create a new variable, based on conditions from another existing variable. Here's one way to do this:

```r
people <- data.frame(age=c(15,65,78,30,22,14,89,44,56,60), sex=rep(c("M","F"),5))
people

people$agecat[people$age > 75] <- "Elder"
people$agecat[people$age > 45 & people$age <= 75] <- "Middle Aged"
people$agecat[people$age <= 45] <- "Young"
people
```

Session 3 - March 10, 2021

Introduction to the `Tidyverse` - Data manipulation with `dplyr`

The Tidyverse is a collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

`Tidyverse` has very popular packages such as `ggplot2` and `dplyr`

To install the complete `Tidyverse` (recommended):

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

Other packages in the `Tidyverse` (I have never used some of them!):

- `tidyr`: set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable.
- `readr`: a fast and friendly way to read rectangular data (like csv, tsv, and fwf).
- `purrr`: provides a complete and consistent set of tools for working with functions and vectors (enhances functional programming).
- `forcats`: provides a suite of useful tools that solve common problems with factors. R uses factors to handle categorical variables, variables that have a fixed and known set of possible values.
- `reprex`: helps creating **reproducible examples** or reprex. The goal of a reprex is to package your problematic code in such a way that other people can run it. It is useful if you are looking for help and *need* to post a question online (e.g. in a public forum)

dplyr is a powerful R package designed to transform and manipulate tabular data (i.e. data frames). The package contains a set of functions (or "verbs") that perform common data manipulation operations such as filtering for rows, selecting specific columns, re-ordering rows, adding new columns and summarizing data.

Here, you can find a cheat sheet for Data Transformation with dplyr.

There are two things worth knowing before you start to work on dplyr and tidyverse in general:

- It uses with tibbles. **Tibbles** are data frames, but slightly tweaked to work better in the tidyverse. You can coerce a data frame into a tibble using the function as_tibble( ) or you can create a tibble from individual vectors with tibble( )

Note: to work with a data frame, you don't have to convert it to a tibble, but if you work with dplyr your output will be a tibble - it has no impact on whatever you do with it, it's just worth noticing they exist:

```r
mydata <- iris
str(mydata)

mydata.t <- as_tibble(iris)
str(mydata.t)
```

- **Pipes (%>%)** take the output from one function and feed it to the first argument of the next function (i.e., the result from one step is then "piped" into the next step). You can use the pipe to rewrite multiple operations that you can read left-to-right, top-to-bottom (reading the pipe operator as "then"). It is useful to make your code shorter and more clear - we'll use it soon!

Tip: I suggest to create a keyboard shortcut for the pipe, it will be really useful: go to the Tools menu (tab) on RStudio and select 'Modify keyboard shortcuts...' Look for 'Insert pipe operators'. In my case, I assigned Cmd + p

The pipe operator is actually a function of the magrittr package and it works with nearly any functions. This is a non-dplyr example:

```r
x <- rnorm(10)
x %>% max # with the pipe, you apply the function max to the object x - it is equivalent
# to max(x) - here it doesn't make much sense to use it, but with longer pieces of code,
# it make things much easier.
```

Here, a simple dplyr example:

```r
ssl <- mydata.t %>%
  select(Species, Sepal.Length)
# This would be equivalent to:
# sp <-  mydata[,c("Species", "Sepal.Length")]
ssl #check how is this different for 'sp'
str(ssl)
```

Important dplyr verbs to remember

Verbs	Description
select()	select columns
filter()	filter rows
arrange()	re-order or arrange rows
mutate()	create new columns
summarise()	summarise values

These all combine naturally with `group_by()` which allows you to perform any operation "by group".

Commonalities

You may have noticed that the syntax and function of all these verbs are very similar:

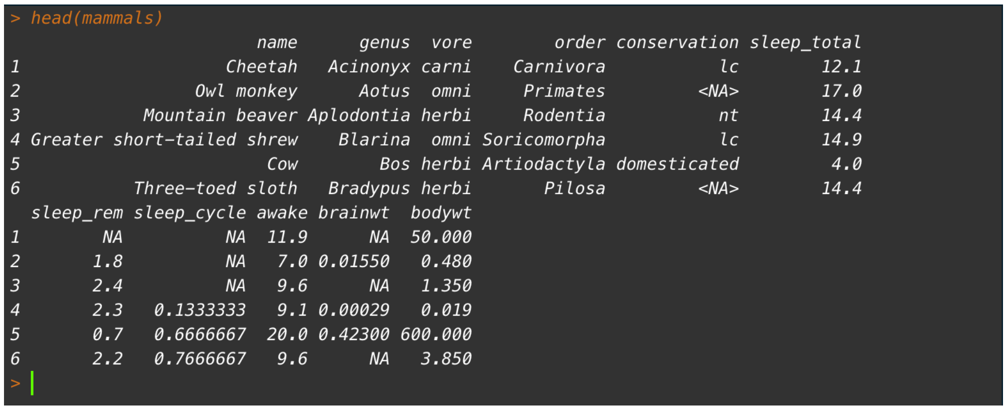
- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using $.
- The result is a new data frame

To explore how the main verbs of `dplyr` work for data manipulation, we will use the mammals data set (provided). The data set contains the sleep times and weights for a set of mammals:

#	Column name	Description
1	name	common name
2	genus	taxonomic rank
3	vore	carnivore, omnivore or herbivore?
4	order	taxonomic rank
5	conservation	conservation status of the mammal
6	sleep_total	total amount of sleep, in hours
7	sleep_rem	rem sleep, in hours
8	sleep_cycle	length of sleep cycle, in hours
9	awake	amount of time spent awake, in hours
10	brainwt	brain weight in kilograms
11	bodywt	body weight in kilograms

Challenge #1: import the provided data set (note that it is a .txt file) and assign it to the object 'mammals'. How many rows and columns does it have?

When you import it, you should have something like this



```
> head(mammals)
                        name       genus  vore        order conservation sleep_total
1                    Cheetah   Acinonyx carni    Carnivora           lc        12.1
2                 Owl monkey       Aotus  omni     Primates         <NA>        17.0
3            Mountain beaver Aplodontia herbi     Rodentia           nt        14.4
4 Greater short-tailed shrew     Blarina  omni Soricomorpha           lc        14.9
5                        Cow         Bos herbi Artiodactyla domesticated         4.0
6           Three-toed sloth   Bradypus herbi       Pilosa         <NA>        14.4
  sleep_rem sleep_cycle awake brainwt  bodywt
1        NA          NA  11.9      NA  50.000
2       1.8          NA   7.0 0.01550   0.480
3       2.4          NA   9.6      NA   1.350
4       2.3   0.1333333   9.1 0.00029   0.019
5       0.7   0.6666667  20.0 0.42300 600.000
6       2.2   0.7666667   9.6      NA   3.850
>
```

Figure 14: head(mammals)

dplyr verbs in action

The most common functions are `select()` and `filter()` which selects columns and filters rows, respectively.

select( ) - Columns

- Select a set of columns (e.g., name and sleep_total):

```
new.m <- mammals %>%
  select(name, sleep_total)

head(new.m)
```

- Select all columns except a specific column (e.g., vore:

```
new.m <- mammals %>%
  select(-vore)

new.m
```

- Select a range of columns:

```
mam <- mammals %>%
  select(sleep_total:bodywt)

mam
```

- Select all columns that start with a character string (e.g., "sl"):

```
sl <- mammals %>%
  select(starts_with("sl"))

head(sl)
```

Additional options to select columns based on a specific criteria:

- `ends_with( )` - Select columns that end with a character string
- `contains( )` - Select columns that contain a character string

filter( ) - Rows

- Filter the rows for mammals that sleep more than 16 hours

```
m.sleep <- mammals %>%
  filter(sleep_total >= 16)

m.sleep
```

- Filter cases for mammals that sleep more than 16 hours and have a body weight of 1 kilogram or more.

```
m.sleep.w <- mammals %>%
  filter(sleep_total >= 16, bodywt >= 1)

m.sleep.w
```

- Filter cases for mammals belonging to orders Perissodactyla and Primates:

```
m.order <- mammals %>%
  filter(order %in% c("Perissodactyla", "Primates"))

m.order
```

Challenge #2:

- Use the `filter( )` function to select only the primates. How many animals in the table are primates? Hint: the `nrow( )` function gives you the number of rows of a data frame or matrix.
- What is the class of the object you obtain after subsetting the table to only include primates?
- Use the `select( )` function to extract the sleeping time (total) for the primates. What class is this object? Hint: use %>% to pipe the results of the `filter()` function to `select()`.
- Now we want to calculate the average amount of sleeping time for primates (the average of the numbers extracted above). One challenge is that the `mean( )` function requires a vector so, if we simply apply it to the output above, we get an error. Look at the R help file for the function `unlist( )` and use it to compute the desired average.

arrange( ) - re-order

Use this function to re-order rows

- Arrange rows by a particular column (e.g., genus) - for this, only list the name of the column you want to arrange the rows by:

```
m.genus <- mammals %>%
  arrange(genus)

m.genus
```

- Now, we will select three columns, arrange the rows by the taxonomic order and then arrange the rows by sleep_total. Finally show the mammals that sleep 16 hours or more:

```
m.sleep.new <- mammals %>%
    select(name, order, sleep_total) %>%
    arrange(order, sleep_total) %>%
    filter(sleep_total >= 16)
```

Note: to arrange the rows in a descending order, use the function desc()

```
m.sleep.new <- mammals %>%
    select(name, order, sleep_total) %>%
    arrange(order, desc(sleep_total)) %>%
    filter(sleep_total >= 16)
```

mutate( ) - new columns

The mutate() function will add new columns to the data frame. You can create a single new column or several at the same time: Create a new column called rem_proportion which is the ratio of rem sleep to total amount of sleep.

```
m.new <- mammals %>%
    mutate(rem_proportion = sleep_rem / sleep_total,
           bodywt_grams = bodywt * 1000)

head(m.new)
```

summarise( ) - summary tables

`summarise( )` creates a new data frame. *Note:* `summarise()` and `summarize()` are synonyms.

```
summ <- mammals %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n()) # n( ) returns the current group size
```

group_by( ) - group operations

Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

grouping doesn't change how the data looks (apart from listing how it's grouped):

```
str(mammals)

by.order <- mammals %>%
  group_by(order)

by.order
```

but it changes how it acts with the other dplyr verbs:

```r
by.order <- mammals %>%
  group_by(order) %>%
  summarise(avg_sleep = mean(sleep_total),
            min_sleep = min(sleep_total),
            max_sleep = max(sleep_total),
            total = n()
  )

# or

max.or <- mammals %>%
  group_by(order) %>%
  select(name,order,genus,sleep_total,bodywt) %>%
   filter(bodywt == max(bodywt))
```

Other useful verbs

- `rename( )` - Rename columns
- `count( )` - Count number of rows in each group defined by a given variable
- `n( )` - Number of values/rows (current group size)
- `relocate( )` - Change column order

Challenge #3: Repeat the last exercise of 'Challenge #2', but this time using just `filter( )` and `summarize( )` to get the answer.

Pivoting with `tidyr( )`

There is a consistent way to organize your data in R, an organization called **tidy data** which will make much easier to work inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

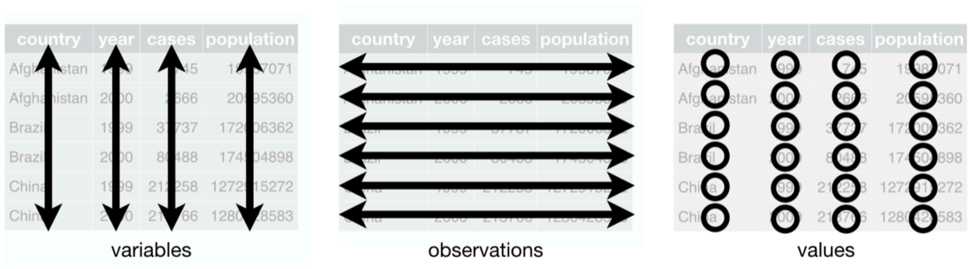


Figure 15: Tidy dataset: variables are in columns, observations are in rows, and values are in cells.

Session 4 - March 17, 2021

(Brief) Intro to loops

As many other things in R, it is good to know that *loops* exist and how to use them. Loops are not functions, but control statements - that allow you to to repeat a sequence of instructions under certain conditions. They allow you to automate parts of your code that are in need of repetition.

There is one important thing you should know: you'll often read that one should avoid making use of loops in R. Why? Well, that's because R supports vectorization, which allows for much faster calculations. For example, solutions that make use of loops are less efficient than vectorized solutions that make use of apply functions, such as `lapply( )` and `sapply( )` (we'll see an example later). **It's often better to use the latter.**

Nevertheless, as a beginner in R, it is good to have a basic understanding of loops and how to write them. There are three types of loops:

- for loop
- while loop
- repeat loop

for loops

It is a type of control statement that enables one to run statements or a set of statements multiple times. For loop is commonly used to iterate over items of a sequence. In this loop the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

Syntax:

```
for (value in sequence)
{
  statement
}
```

Here is an easy example: suppose you want to do several printouts of the following sentence: "The year is [year]" where [year] is equal to 2010, 2011, up to 2020. You can do this as follows:

```r
print(paste("The year is", 2010))
```

```
## [1] "The year is 2010"
```

```r
print(paste("The year is", 2011))
```

```
## [1] "The year is 2011"
```

```r
print(paste("The year is", 2012))
```

```
## [1] "The year is 2012"
```

```r
print(paste("The year is", 2013))
```

```
## [1] "The year is 2013"
```

```r
print(paste("The year is", 2014))
```

```
## [1] "The year is 2014"
```

```r
print(paste("The year is", 2015))
```

```
## [1] "The year is 2015"
```

```r
print(paste("The year is", 2016))
```

```
## [1] "The year is 2016"
```

```r
print(paste("The year is", 2017))
```

```
## [1] "The year is 2017"
```

```r
print(paste("The year is", 2018))
```

```
## [1] "The year is 2018"
```

```r
print(paste("The year is", 2019))
```

```
## [1] "The year is 2019"
```

```r
print(paste("The year is", 2020))
```

```
## [1] "The year is 2020"
```

You immediately see this is rather tedious: you repeat the same code chunk over and over. In this case, by making use of a for loop in R, you can automate the repetitive part:

```r
years <- c(2010:2020) # first, let's define our sequence

for (x in years){  # for each value in our sequence
  print(paste("The year is", x)) # join "The year is" and each value, and print it
  }
```

Let's have a look at another example: Let's consider the following matrix:

```r
x <- matrix(c(28, 35, 13, 13, 1.62, 1.53, 1.83, 1.71, 65, 59, 72, 83),
             nrow = 4, dimnames = list(c("Veronica", "Karl", "John", "Peter"),
                                       c("Age", "Size", "Weight")))
```

We would like to get the average values for all three columns. This can be done with a single loop:

a. Loop over all columns by name.
b. Extract the current column.
c. Calculate the average (arithmetic mean).

```r
for (var in colnames(x)) {        # a (for each variable in x)
    m <- mean(x[, var])           # b and c (calculate the mean of each variable)
    print(paste("Average", var, "is", m))
}
```

The `for` loop is by far the most popular and its construct implies that the number of iterations is fixed and known in advance. But if you do not know or control the number of iterations or you can't predict the conditions, the `while` and `repeat` loops can help you.

While loops

The `while` loop is made of an initialization block as before, followed by a logical condition. This condition is typically a comparison (e.g., greater than, less than or equal to) or logical (i.e., TRUE/FALSE) operator. If the result is False (F), the loop is never executed, if it is True (T), the instruction or block of instructions will be executed next. In contrast to a for-loop which runs for a fixed number of iterations, a while-loop runs while a condition is true: Suppose we want to print all numbers `x` in `1,2,...,∞` as long as `x^2` is lower than 20, starting with `x=0`:

```r
# Start with 0
x <- 0
# Loop until condition is FALSE
while (x^2 < 20) {
  print(x)      # Print x
  x <- x + 1    # Increase x by 1
}
```

if x is 5, 5^2 = 25; x^2 < 20 is FALSE, so the loop stops.

repeat loops

The last one is a repeat-loop. In contrast to the other two the repeat loop runs forever – except we explicitly stop it by calling `break`.

For example, we could use a repeat loop to solve the same task as shown above where we would like to get all numbers `x [0,1,...,∞]` where `x^2 < 20` like this:

```r
# Initialization
x <- 0
# Repeat loop
repeat {
    if (x^2 > 20) break      # Break condition (important)
    print(x)                 # print(x)
    x <- x + 1               # Increase x by 1
}
```

Challenge #1: Write a for loop that iterates over the numbers 1 to 7 and prints the cube of each number using `print( )`.

Loop replacements

Instead of the three basic repetitive control structures (for, while, and repeat) R comes with a series of functions which can be used as replacements. These 'loop replacements' are real functions (no longer control statements). The `apply` family of functions can often do the tasks of most "home-brewed" loops, sometimes faster (though that won't really be an issue for most people) but more importantly with a much lower risk of error.

A strategy to have in the back of your mind which may be useful is: for every loop you make, try to remake it using an `apply` function (often `lapply` or `sapply` will work). If you can, use the `apply` version. There's nothing worse than realizing there was a small, tiny, seemingly meaningless mistake in a loop which weeks, months or years down the line has propagated into a huge mess. It is strongly recommended trying to use the `apply` functions whenever possible.

Function	**Description**
apply()	Apply a function over margins of an array (e.g., over rows or columns of a matrix).
lapply()	Apply a function over a vector or list, returns a list.
sapply()	Like lapply() but tries to simplify the result to a vector or matrix.
vapply()	Like sapply() but with pre-specified return value.
tapply()	Apply a function over a ragged array (e.g., within groups) and return a table.

You can take a look at the `apply` functions by entering `?apply` on the console, or searching in the help tab.

Let's use a matrix generated with random values

```r
x <- matrix(rnorm(20), nrow = 4,
            dimnames = list(NULL, LETTERS[1:5]))
```

Suppose that for each column of the matrix we would like to calculate the means and standard deviation:

To calculate the mean over all columns, we need to call:

```r
my.mean <- apply(x, 2, mean) # the arguments are: x=our matrix, MARGIN=2, FUN=mean
```

The `MARGIN = 2` indicates that we would like to apply the function column-by-column. `mean` is the function to be applied. As we have used a named matrix (try `colnames(x)`) , we will get a named vector as a result. The very same can be used to calculate the standard deviation.

Challenge #2:

- Calculate the standard deviation of each column of matrix `x`
- Calculate the sum of the values for each row of matrix `x` - Hint: check the argument "MARGIN" on the description file of the `apply( )` function.

Useful control statements: if/if…else

Decision making is an important part of programming. This can be achieved in R programming using the conditional `if...else` statement.

if **statement:**

The syntax of the `if` statement is:

```
if (test_expression) {
statement
}
```

If the `test_expression` is `TRUE`, the statement gets executed. But if it's `FALSE`, nothing happens.

For example:

```
x <- 5

if(x > 0){
print("Positive number")
}
```

if…else **statement:**

The syntax of the `if...else` statement is:

```
if (test_expression) {
statement1
} else {
statement2
}
```

The `else` part is optional and is only evaluated if `test_expression` is `FALSE`. It is important to note that else must be in the same line as the closing braces of the if statement.

For example:

```
x <- -5
if(x > 0){
print("Non-negative number")
} else {
print("Negative number")
}
```

if…else **ladder (if…else…if):**

It allows you execute a block of code among more than 2 alternatives.

The syntax of if…else statement is:

```
if ( test_expression1) {
statement1
} else if ( test_expression2) {
statement2
} else if ( test_expression3) {
statement3
} else {
statement4
}
```

Only one statement will get executed depending upon the test_expressions.

For example:

```
x <- 0

if (x < 0) {
print("Negative number")
} else if (x > 0) {
print("Positive number")
} else
print("Zero")
```

There is an easier way to use the `if…else` statement specifically for vectors: You can use the `ifelse()` function instead. Check `?ifelse`

Challenge #3: These statements can be useful to create new variables, based on existing variables. Let's consider the following data set that recorded the string length (in cms.) and color, N represents the number of strings I measured:

```
my.strings <- data.frame(N = c(1:20),
                         len=runif(20, min=1, max=20),
                         col=sample(c("red","blue","green","black"),20, replace=T)
                         )
```

Using the function `ifelse( )`, create a new variable "group" where "short" is assigned to strings measuring 10 cm or less and "long" is assigned to strings measuring more than 10cm.

(Brief) Intro to user-defined functions

A function is a set of statements organized together to perform a specific task. R has a large number of built-in functions and the user can create their own functions.

When Should I Use Functions?

- *Avoid repetitions:* Try to avoid copying & pasting chunks of code. Whenever you use copy & paste, it is a good indication that you should think about writing a function.

- *Facilitate reuse:* Whenever similar code chunks should be used in different parts of the code, or even different scripts/projects.
- *Impose structure:* Functions help you to structure your code and to avoid long and/or complex scripts.

An R function is created by using `function( )`. The basic syntax of an R function is:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
   return(val)
}
```

Function Components

The different parts of a function are:

- *Function Name:* This is the name of the function. It is stored in R environment as an object with this name.
- *Arguments:* An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- *Function Body:* The function body contains a collection of statements that defines what the function does.
- *Return Value:* The return value of a function is the last expression in the function body to be evaluated.

Let's start to write some functions, so we learn about the different parts:

Function A

- Name: `say.hello`
- Arguments: None
- Instructions: Outputs `"Hello world!"` on the console
- Return value: No explicit return (output)

```
say.hello <- function() {
    print("Hello World!")
}
```

Take a look at your upper-right panel, under the "Functions" list. After running the previous chunk of code, you should have now a function called "say.hello". Now, let's call (use) the function:

```
# Call function
say.hello()
```

As the function has no input arguments, nothing has to be declared between the round brackets `()` when calling the function. We are not even allowed to do so (try say.hello("test")).

Function B

- Name: `say.hi` (new name)
- Arguments: One argument: `x`
- Instructions: Paste and output the result using `paste()`
- Return value: No explicit return (output)

As shown in the instructions we have to adjust our function to have one input argument named `x`. We will use the content of `x` and combine it with `"Good morning"` to say hello to a specific person.

```
say.hi <- function(x) {
    paste("Good morning", x, sep=", ") # paste the string "Good morning, with our value
                                       # of x, and use a comma and a space to separate
                                       # them
}
```

The difference to "Function A": we now have one input argument to control the behavior of the function. As there is no default (we'll come back to that later) this is a *mandatory* argument. If we do not specify it, we will run into an error as the function expects that we do hand over this argument:

```
# Try
say.hi("Alex")
# and
say.hi()
# and
say.hi(1)
# and
say.hi("Rob")
```

Function C

- Name: say.hello (same name; re-declare function)
- Arguments: One argument: `x`
- Instructions: Combine "Hi" and argument `x` and store the resulting character string on `res`. Do not print/show the result.
- Return value: Explicit return of the result

Let's declare a new function `say.hello`. We are using `paste()` to create the welcome message, and then return that character to be used outside the function:

```
 # re-declare the function (overwrites the previous one!)
say.hello <- function(x) {
    res <- paste("Hi", x)
    return(res)
}

# and lets assign it to an object:
result <- say.hello("Matt")
```

Function D

So far, our functions always returned a character string. A function can, of course also print one thing, and return something else. Let's create one last function:

- Name: hello
- Arguments: One argument: `x`
- Instructions: Use paste() to create a character string, show it on the console using cat(). Then, calculate (count) the number of characters in this string
- Return value: Explicit return; number of characters of the newly created string

```
hello <- function(x) {
    # First, create the new string using paste
    x <- paste("Hi", x) # As we no longer need 'x' later on, we simply overwrite it here
    cat(x, "\n") #  immediately print x
    # Count the number of characters in 'x'
    res <- nchar(x)
    # Return the object 'res'
    return(res)
}

hello("Max")
```

When calling the function, we can immediately see "Hello Max". This is caused by calling `cat()` inside the function (`\n` is a regular expression used to denote 'end-of-line'). But let's see what the function returns:

```
result <- hello("Max")
result
class(result)
```

What we get in return is an integer vector which contains 6. This is the number of characters in "Hi Max" ("Hi" has two characters, "Max" another three, plus 1 for the space in between, thus 6).

Default and multiple arguments

Multiple arguments: So far, our function(s) always only had one single input argument. More often than not, functions come with multiple arguments:

```
my.fun <- function(a,b){
  return(a+b)
}
```

In this case, both arguments are required arguments. When calling the function, we must specify both. As in the function declaration, the arguments are separated by a comma `(,)`.

```
my.res <- my.fun(2,2)
```

Default arguments: A default argument definition allows to define 'optional' arguments. The user can always specify them if needed, if not explicitly specified in the function call, the default value will be used.

Let's extend the function from above and add a second argument which will take the value 2 by default:

```r
my.fun <- function(a,b=2){
  return(a+b)
}
```

To call the function, you can 'let' the function use the argument b as defined, or assign a new value:

```r
result <- my.fun(a=2)
result

# or
result2 <- my.fun(2,5)
result2

# When calling a function, Arguments can be specified or not (named or unnamed).
# Named arguments are always matched first, the remaining ones are matched by
# position:
result3 <- my.fun(a=3, b=4)
# or
result4 <- my.fun(b=7, a=3)
#or
result5 <- my.fun(b=5, 5)
```

Session 5 - March 31, 2021

Data Visualization: `ggplot2`

"The goal of [a good plot] is to help you understand your data by visualizing it, and to help you convey that understanding to others."

'ggplot2' is a system for creating graphics. It is a package developed by Hadley Wickham (initially released for R in 2007) that implements the idea of *grammar of graphics* – a concept created by Leland Wilkinson in his 2005 book.

"With `ggplot2` you provide the data, tell it how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details." `ggplot2` has an underlying grammar that allows you to make graphs by combining independent components: it's designed to work iteratively. You start with a layer that shows the raw data, then you add layers of annotations and statistical summaries. This allows you to produce graphics using the same structured thinking that you would use to design an analysis. This reduces the distance between the plot in your head and the one on the page.

What is the grammar of graphics?

Wilkinson created the *grammar of graphics* to describe the fundamental features that underlie all statistical graphics. `ggplot2` builds on Wilkinson's grammar by focusing on the use of layers and adapting it for use in R. Briefly, the "grammar" tells us that a plot maps the data to the aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars). The components of the grammar are described below - this will introduce us to some of the terminology that will be useful when using `ggplot2`. Don't worry if it doesn't make sense right away: you'll have more opportunities to learn about the components and how they work together:

All plots are composed of the **data**, the information you want to visualize, and a **mapping,** the description of how the data are mapped to aesthetic attributes. There are five mapping components:

1. A **layer** is a collection of geometric elements and statistical transformations. Geometric elements, **geom**s for short, represent what you actually see in the plot: points, lines, polygons, etc. Statistical transformations, **stat**s for short, summarize the data: for example, binning and counting observations to create a histogram, or fitting a linear model.

2. **Scales** map values in the data space to values in the aesthetic space. This includes the use of color, shape or size. Scales also draw the legend and axes, which make it possible to read the original data values from the plot.

3. A **coord**, or coordinate system, describes how data coordinates are mapped to the plane of the graphic. It also provides axes and grid lines to help read the graph. We normally use the Cartesian coordinate system, but a number of others are available, including polar coordinates and map projections.

4. A **facet** specifies how to break up and display subsets of data as small multiple panels.

5. A **theme** controls the finer details of display, like the font size and background color. While the defaults in `ggplot2` have been chosen with care, you may need to consult other references to create an attractive plot.

As you know, `ggplot2` is a package that belongs to the `tidyverse`. So let's start by making sure you have the latter loaded on your session:

```
library(tidyverse)
```

- Here, you can download the `ggplot2` cheat sheet, really useful!
- And here you can find the reference card for `ggplot2`.

The goal is to teach you how to produce useful graphics with ggplot2 as quickly as possible. You'll learn the basics of `ggplot()`. `ggplot2` allows you to make complex plots with just a few extra lines of code.

We'll mostly use one data set that's bundled with ggplot2: `mpg`. It includes information about the fuel economy of popular car models between 1999 and 2008, collected by the US Environmental Protection Agency:

```
head(mpg)
```

Challenge #1:

1. Find out what each variable in the `mpg` dataset means. Hint: have to look for the dataset description file.
2. How can you find out what other datasets are included with `ggplot2`?
3. Which manufacturer has the most models in this dataset? Which model has the most variations?

The components of a graph

To illustrate how `ggplot2` works, let's make a basic scatterplot, just as an example. For now, just run the code to see the differences: Suppose you want to plot the effect of the engine size (x) on the fuel economy (y). You can do it on R base as follows:

```r
plot(mpg$displ, mpg$hwy)
```

Or in ggplot2 by entering the following:

```r
ggplot(mpg) +
  geom_point(aes(x = displ, y = hwy))
```

Note that the previous code produces the same plot as this one:

```r
ggplot(mpg, aes(displ, hwy)) +
  geom_point()
```

The first step in learning ggplot2 is to be able to break a graph apart into components. Let's break down the plot above and introduce some of the ggplot2 terminology. The main three components to note are:

- **Data:** The fuel economy data in popular car models: mpg. Also referred to as the **data** component.
- **Aesthetic mapping:** The plot uses several visual cues to represent the information provided by the dataset. The two most important cues in this plot are the point positions on the x-axis and y-axis: engine size mapped to x position, fuel economy to y position. Each point represents a different observation, and we *map* data about these observations to visual cues like x- and y-scale. We refer to this as the **aesthetic mapping** component.
- **Geometry:** The plot above is a scatterplot. This is referred to as the **geometry** component. Other possible geometries are barplot, histogram, smooth densities, qqplot, and boxplot.

The first part, `ggplot()`, tells it to create a plot object, and the second part, `geom_point()`, tells it to add a layer of points to the plot. The first two unnamed arguments to `aes( )` will always be mapped to x and y, so you can omit the `'x='` and `'y='` from the first ggplot example.

ggplot objects

The first step in creating a ggplot2 graph is to define a ggplot object. We do this with the function `ggplot()`, which initializes the graph. If we read the help file for this function, we see that the first argument is used to specify what data is associated with this object:

```r
ggplot(data=mpg)
```

We can also pipe the data in as the first argument. So this line of code is equivalent to the one above:

```r
mpg %>% ggplot()
```

It renders a plot, in this case a blank slate since no geometry has been defined. The only style choice we see is a gray background.

What has happened above is that the object was created and, because it was not assigned to a named object, it was automatically evaluated. But we can assign our plot to an object, for example like this:

```r
p <- ggplot(data = mpg)
class(p)
```

Geometries

In `ggplot2` we create graphs by adding **layers**. Layers can define geometries, compute summary statistics, define what scales to use, or even change styles. To add layers, we use the symbol `+`. In general, a line of code will look like this:

DATA %>% `ggplot()` + LAYER 1 + LAYER 2 + … + LAYER N

Usually, the first added layer defines the geometry. Geometry function names follow the pattern: geom_X where X is the name of the geometry. Some examples include `geom_point()`, `geom_bar()`, and `geom_histogram()`:

- `geom_smooth()` fits a smoother to the data and displays the smooth line and its standard error.
- `geom_boxplot()` produces a box-and-whisker plot to summarize the distribution of a set of points.
- `geom_histogram()` and `geom_freqpoly()` show the distribution of continuous variables.
- `geom_bar()` shows the distribution of categorical variables. `geom_bar(stat = "identity")` makes a bar chart. We need stat = "identity" because the default stat automatically counts values (so is essentially a 1d geom). The identity stat leaves the data unchanged. Multiple bars in the same location will be stacked on top of one another.
- `geom_path()` and `geom_line()` draw lines between the data points. A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction. Lines are typically used to explore how things change over time.
- `geom_area()` draws an area plot, which is a line plot filled to the y-axis (filled lines). Multiple groups will be stacked on top of each other.
- `geom_point()` produces a scatterplot.
- `geom_polygon()` draws polygons, which are filled paths. Each vertex of the polygon requires a separate row in the data. It is often useful to merge a data frame of polygon coordinates with the data just prior to plotting.
- `geom_rect()` and `geom_tile()` draw rectangles. `geom_rect()` is parameterised by the four corners of the rectangle: `xmin`, `ymin`, `xmax` and `ymax`. `geom_tile()` is exactly the same, but parameterised by the center of the rectangle and its size: `x`, `y`, `width` and `height`.

Aesthetic mappings

Aesthetic mappings describe how properties of the data connect with features of the graph, such as distance along an axis, size, or color. The `aes()` function connects data with what we see on the graph by defining aesthetic mappings and will be one of the functions you use most often when plotting. The outcome of the `aes()` function is often used as the argument of a geometry function.

In our first example, we saw how this geometry layer works. This time, instead of defining our plot from scratch, we can also add a layer to the **p object** that was defined above:

```
p + geom_point(aes(displ, hwy))
```

The scale and labels are defined by default when adding this layer. Like `dplyr` functions, `aes()` also uses the variable names from the object component: we can use *displ* and *hwy* without having to call them as mpg*displandmpg*hwy. The behavior of recognizing the variables from the data component is quite specific to `aes()`. With most functions, if you try to access the values of displ or hwy outside of `aes()` you receive an error.

Layers

To understand better the concept of layers, we'll add a second layer in the plot by adding a label to each point to identify the number of cylinders. The `geom_label()` and `geom_text()` functions permit us to add text to the plot with and without a rectangle behind the text, respectively.

Because each point (each car in this case) has a label, we need an aesthetic mapping to make the connection between points and labels. By reading the help file, we learn that we supply the mapping between point and label through the label argument of `aes()`. So the code looks like this:

```
p + geom_point(aes(displ, hwy)) +
    geom_text(aes(displ, hwy, label = cyl))
```

We have successfully added a second layer to the plot - note, however, that the resulting plot is not necessarily a nice one, it just added the number of cylinders on top of each dot in the scatterplot. Let's try plotting the numbers without the points:

```
p.test <- p + geom_text(aes(displ, hwy, label = cyl))
p.test
```

Arguments on the geometry functions

Each geometry function has many arguments other than aes and data. They tend to be specific to the function. For example, if we want to draw larger points, we see in the help file (?geom_point) that size is an aesthetic and we can change it like this:

```
p + geom_point(aes(displ, hwy), size = 3)
```

Now because the points are larger it is hard to see the labels. If we read the help file for geom_text, we see the nudge_x argument, which moves the text slightly to the right or to the left:

```
p + geom_point(aes(displ, hwy), size = 3) +
    geom_text(aes(displ, hwy, label = cyl), nudge_x = 0.15)
```

Global versus local aesthetic mappings

In the previous line of code, we define the mapping `aes(displ, hwy)` twice, once in each geometry. We can avoid this by using a global aesthetic mapping. We can do this when we define the ggplot object. Remember that the function `ggplot()` contains an argument that permits us to define aesthetic mappings. If we define a mapping in `ggplot()`, all the geometries that are added as layers will default to this mapping. We redefine p:

```
p <- mpg %>% ggplot(aes(displ, hwy, label = cyl))
```

and then we can simply write the following code to produce the previous plot:

```
p + geom_point(size = 3) +
    geom_text(nudge_x = 0.15)
```

If necessary, we can override the global mapping by defining a new mapping within each layer. These local definitions override the global. Here is an example:

```r
p + geom_point(size = 3) +
    geom_text(aes(x = 4, y = 40, label = "Hello there!"))
```

Clearly, the second call to `geom_text()` does not use displ and hwy - we re-defined the position of the text we want to add.

Scales

Let's assume our desired axes should be in log-scale. This is not the default, so this change needs to be added through a scales layer. A quick look at the cheat sheet reveals the scale_x_continuous function lets us control the behavior of scales. We use them like this:

```r
p + geom_point(size = 3) +
    geom_text(nudge_x = 0.02) + #Note we had to adjust this too.
    scale_x_continuous(trans = "log10") +
    scale_y_continuous(trans = "log10")
```

This particular transformation is so common that `ggplot2` provides the specialized functions `scale_x_log10()` and `scale_y_log10()`, which we can use to rewrite the code like this:

```r
p + geom_point(size = 3) +
    geom_text(nudge_x = 0.02) +
    scale_x_log10() +
    scale_y_log10()
```

Labels and titles

Similarly, to change labels and add a title, we use the following layers:

```r
p + geom_point(size = 3) +
  scale_x_log10() +
  scale_y_log10() +
  xlab("Log of Engine displacement (L)") +
  ylab("Log of Miles per gallon") +
  ggtitle("Fuel Economy (1999 - 2008)")
```

Categories as colors

We can change the color of the points using the `col` argument in the `geom_point()` function. To facilitate demonstration of new features, we will redefine `p` to be everything except the points layer:

```r
p <-  mpg %>% ggplot(aes(displ, hwy, label = cyl)) +
  scale_x_log10() +
  scale_y_log10() +
  xlab("Log of Engine displacement (L)") +
  ylab("Log of Miles per gallon") +
  ggtitle("Fuel Economy (1999 - 2008)")
```

and then test out what happens by adding different calls to `geom_point()`. We can make all the points pink by adding the `color` argument:

```r
p + geom_point(size = 3, color ="orchid")
```

A nice default behavior of `ggplot2` is that if we assign a categorical variable to color, it automatically assigns a different color to each category and also adds a legend.

Since the choice of color is determined by a feature of each observation, this is an aesthetic mapping. To map each point to a color, we need to use `aes()`. We use the following code:

```r
p + geom_point(aes(col=class), size = 3)
```

To add additional *variables* to a plot, we can use other aesthetics like shape and size.

Challenge #2: Try mapping each point to a different size or shape by adjusting the `aes()` arguments using the previous plot. Map the type of *drive train* (drv) to the argument `shape` and the *number of cylinders* (cyl) to the argument `size`.

Annotation, shapes, and adjustments

We often want to add shapes or annotation to figures that are not derived directly from the aesthetic mapping; examples include labels, boxes, shaded areas, and lines.

```r
# We could add a horizontal line:
p + geom_hline(yintercept = 30, col="cadetblue", lty=6, lwd = 1.1) +
  geom_point(aes(col=class), size = 3)

# Or we can add a smooth line - we draw it first, so it doesn't go over our points
p <- p +  geom_smooth(method = "lm", col="darkgrey", lty=2) +
  geom_point(aes(col=class), size = 3)

# Note that the arguments lty and lwd belong to the graphical parameters
# function (par) we saw in Session 4
# We have also defined here a new object p.
```

Themes

The style of a ggplot2 graph can be changed using the theme functions. Several themes are included as part of the ggplot2 package.

You can check out the existing themes included in the package, here

For example:

```r
p + theme_light()
```

Add-on packages

The power of ggplot2 is augmented further due to the availability of add-on packages. The remaining changes needed to put the finishing touches on our plot require the `ggthemes` and `ggrepel` packages. Go ahead and install and load the packages to try them:

```
library(ggthemes)
library(ggrepel)
p + geom_text_repel() +
  theme_economist_white()
```