GRAPH Network

Foundations of data analysis with R

An Introduction to Modern Data Analysis

This book is a compilation of lesson notes from the 3-month online course offered by The GRAPH Courses.. To access the lesson videos, exercise files, and online quizzes, please visit our website, thegraphcourses.org

Lesson notes | Setting up R and RStudio

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Learning objective

1. You can access R and RStudio, either through RStudio.cloud or by downloading and installing these software to your computer.

Introduction

To start you off on your R journey, we'll need to set you up with the required software, R and RStudio. **R** is the programming language that you'll use write code, while **RStudio** is an integrated development environment (IDE) that makes working with R easier.

Working locally vs. on the cloud

There are two main ways that you can access and work with R and RStudio: download them to your computer, or use a web server to access them on the cloud.

Using R and RStudio on the cloud is the less common option, but it may be the right choice if you are just getting started with programming, and you do not yet want to worry about installing software. You may also prefer the cloud option if your local computer is old, slow, or otherwise unfit for running R.

Below, we go through the setup process for RStudio Cloud, Rstudio on Windows and RStudio on macOS separately. Jump to the section that is relevant for you!

WATCH OUT

WATCH OUT

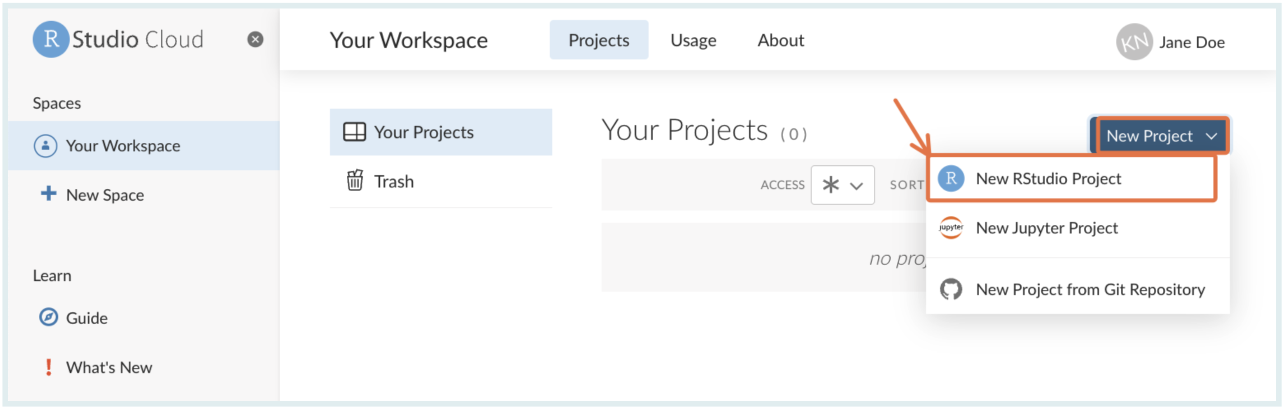
than 25 hours per month, you may want to avoid this option.

RStudio on the cloud

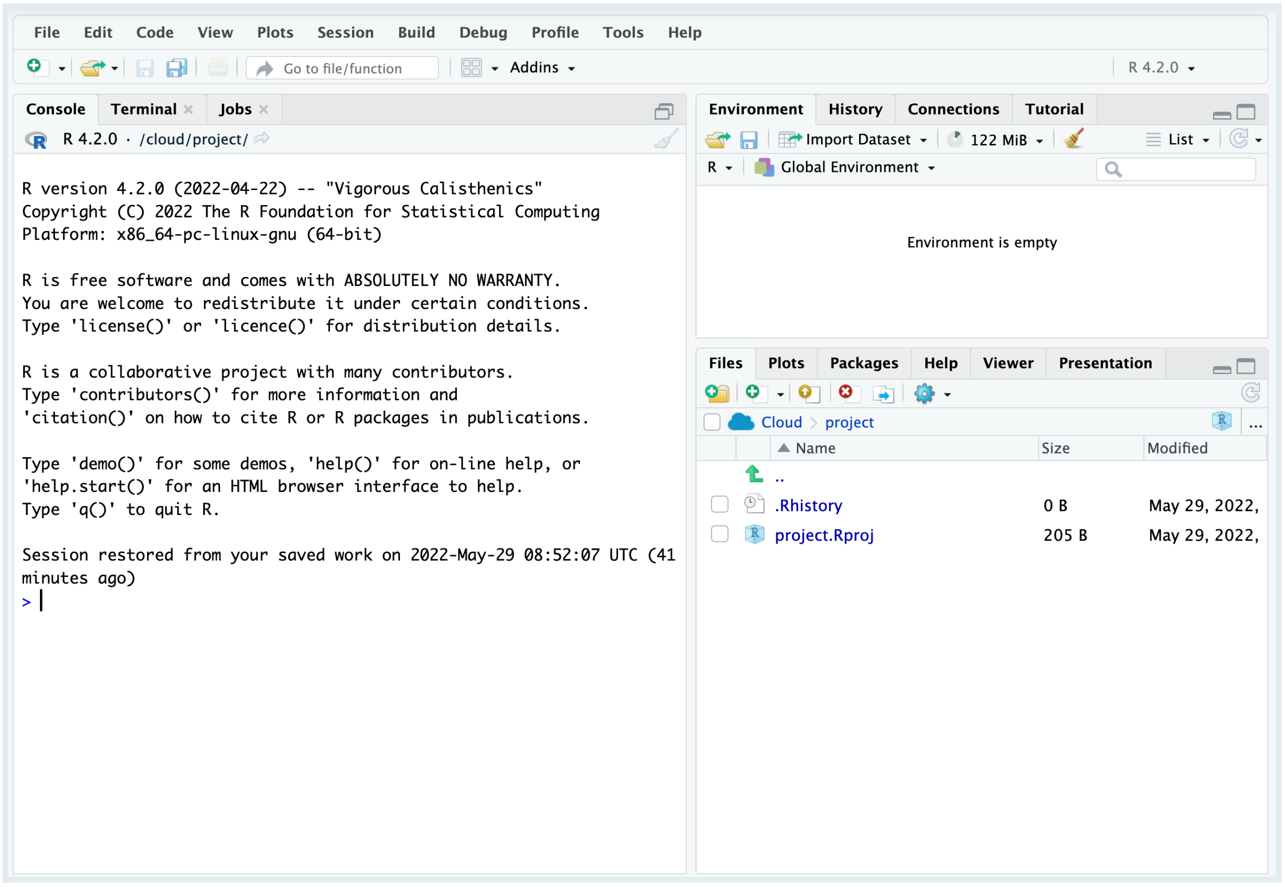
If you'll be working on the cloud, follow the steps below:

1. Go to the website rstudio.cloud and follow the instructions to sign up for a free account. (We recommend signing up with Google if you have a Google account, so you don't need to remember any new passwords).

2. Once you're done, click on the "New Project" icon at the top right, and select "New RStudio Project".



You should see a screen like this:

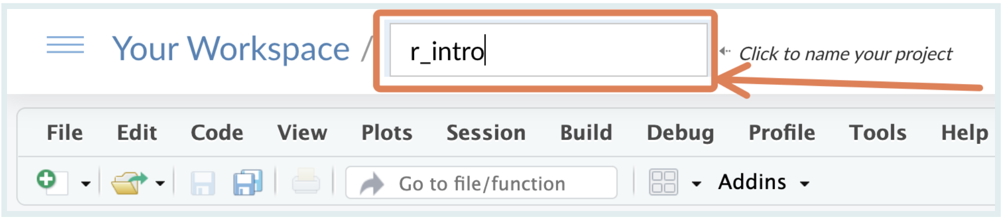


```
R version 4.2.0 (2022-04-22) -- "Vigorous Calisthenics"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

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.

Session restored from your saved work on 2022-May-29 08:52:07 UTC (41
minutes ago)
>
```

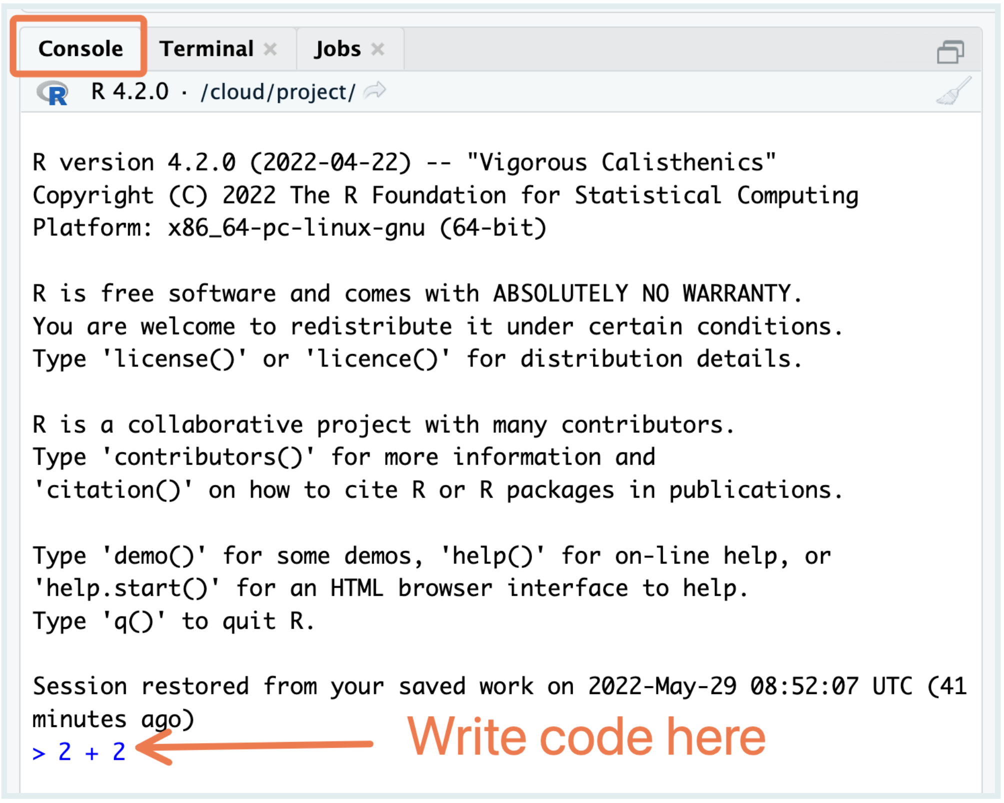
Name	Size	Modified
..		
.Rhistory	0 B	May 29, 2022,
project.Rproj	205 B	May 29, 2022,

This is RStudio, your new home for a long time to come!

At the top of the screen, rename the project from “Untitled Project” to something like “r_intro”.


Your Workspace /
r_intro
Click to name your project
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins

You can start using R by typing code into the “console” pane on the left:



```
R version 4.2.0 (2022-04-22) -- "Vigorous Calisthenics"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

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.

Session restored from your saved work on 2022-May-29 08:52:07 UTC (41
minutes ago)
> 2 + 2
```

Try using R as a calculator here; type `2 + 2` and press Enter.

That's it; you're ready to roll. Whenever you want to reopen RStudio, navigate to rstudio.cloud,

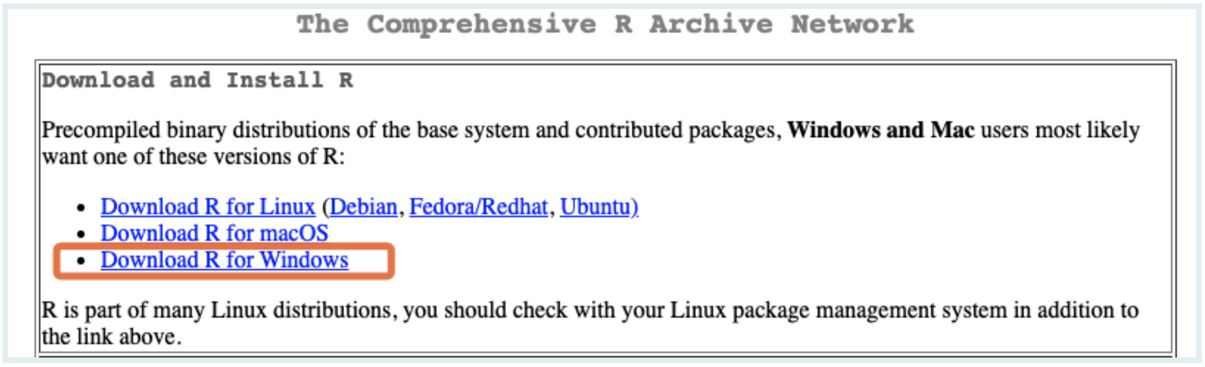
Proceed to the "wrapping up" section of the lesson.

Set up on Windows

Download and install R

If you're working on Windows, follow the steps below to download and install R:

1. Go to cran.rstudio.com to access the R installation page. Then click the download link for Windows:


The Comprehensive R Archive Network

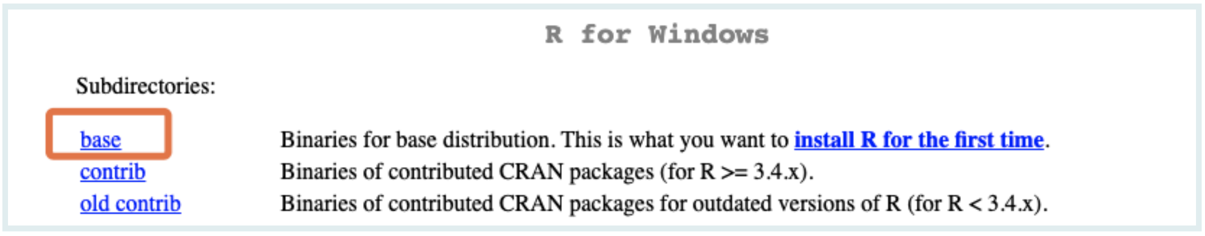
Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

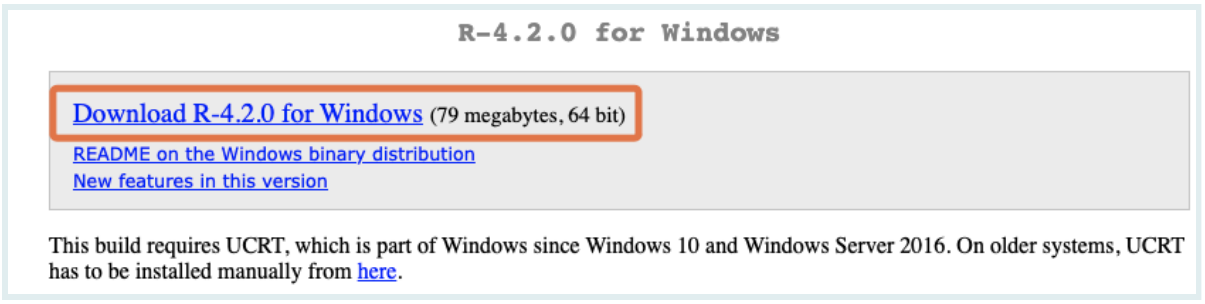
2. Choose the "base" sub-directory.


R for Windows

Subdirectories:

base	Binaries for base distribution. This is what you want to **install R for the first time**.
contrib	Binaries of contributed CRAN packages (for R >= 3.4.x).
old contrib	Binaries of contributed CRAN packages for outdated versions of R (for R < 3.4.x).

3. Then click on the download link at the top of the page to download the latest version of R:


R-4.2.0 for Windows

Download R-4.2.0 for Windows (79 megabytes, 64 bit)
README on the Windows binary distribution
New features in this version

This build requires UCRT, which is part of Windows since Windows 10 and Windows Server 2016. On older systems, UCRT has to be installed manually from here.

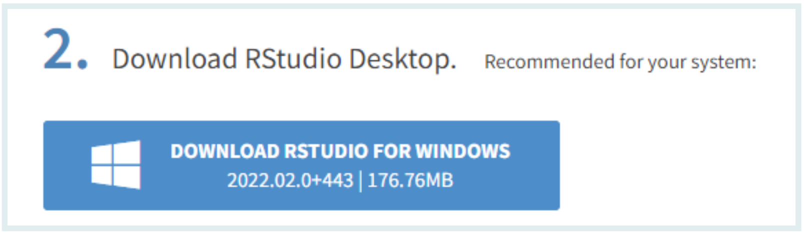
Note that the screenshot above may not show the latest version.

4. After the download is finished, click on the downloaded file, then follow the instructions on the installation pop-up window. During installation, you should not have to change any of the defaults; just keep clicking "Next" until the installation is done.

   Well done! You should now have R on your computer. But you likely won't ever need to interact with R directly. Instead you'll use the RStudio IDE to work with R. Follow the instructions in the next section to get RStudio.

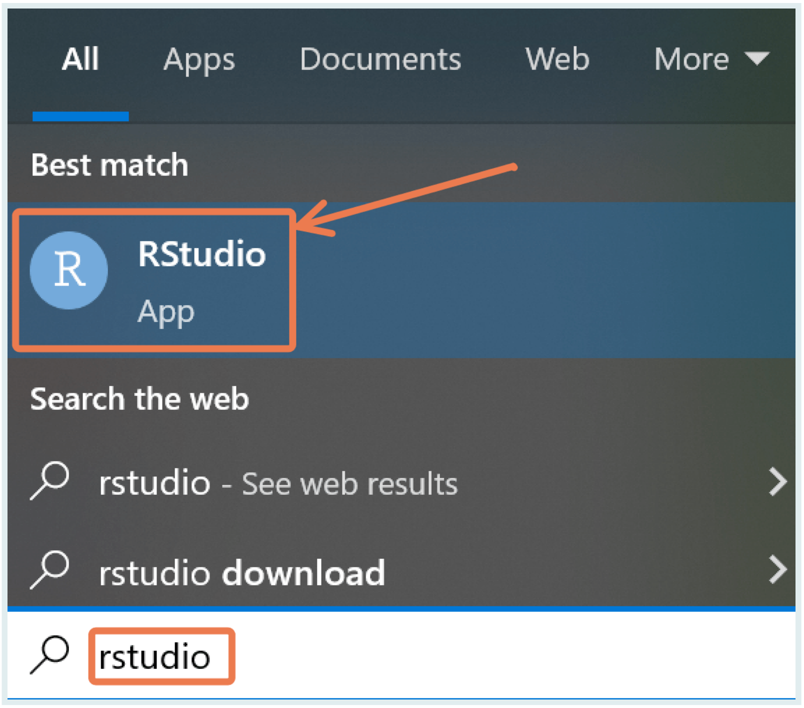
Download, install & run RStudio

To download RStudio, go to rstudio.com/products/rstudio/download/#download and download the Windows version.

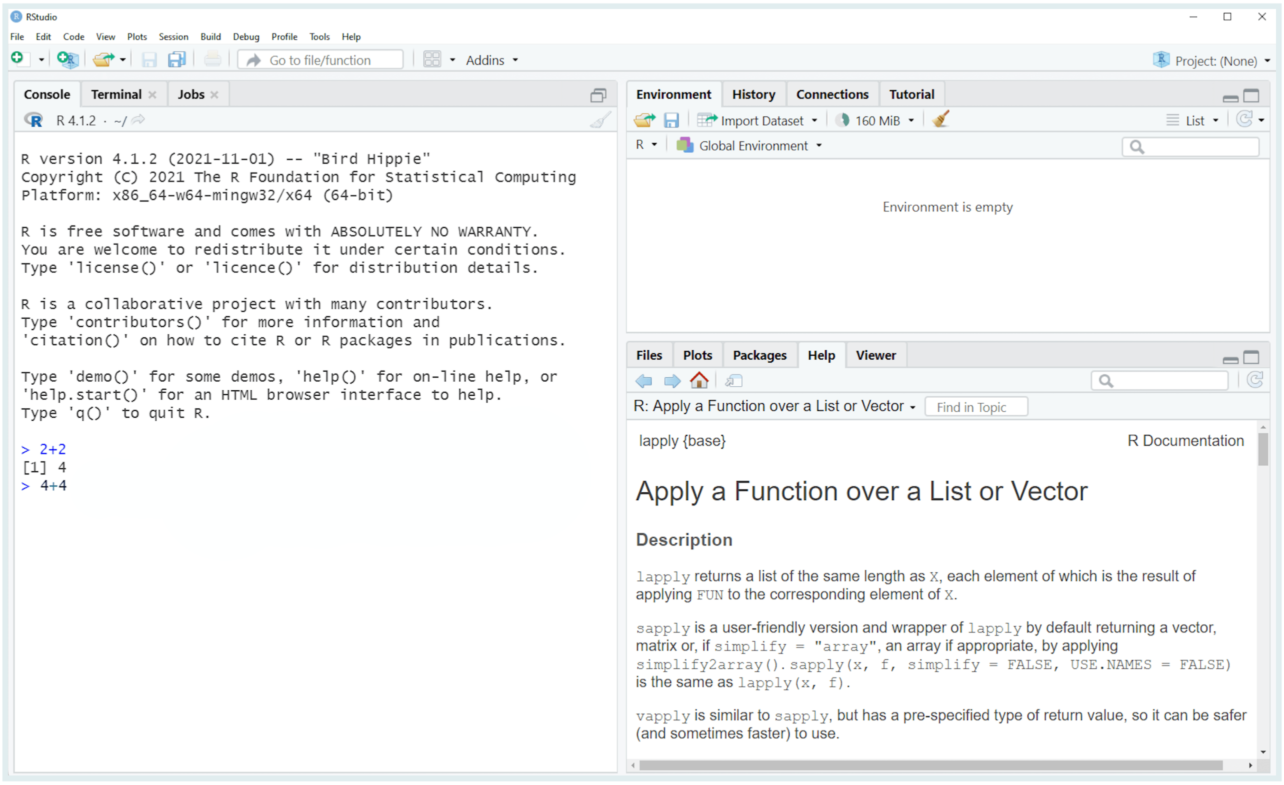

2. Download RStudio Desktop.
Recommended for your system:
DOWNLOAD RSTUDIO FOR WINDOWS
2022.02.0+443 | 176.76MB

After the download is finished, click on the downloaded file and follow the installation instructions.

Once installed, RStudio can be opened like any application on your computer: press the Windows key to bring up the Start menu, and search for "rstudio". Click to to open the app:



You should see a window like this:



```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

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

lapply {base} R Documentation

Apply a Function over a List or Vector

Description

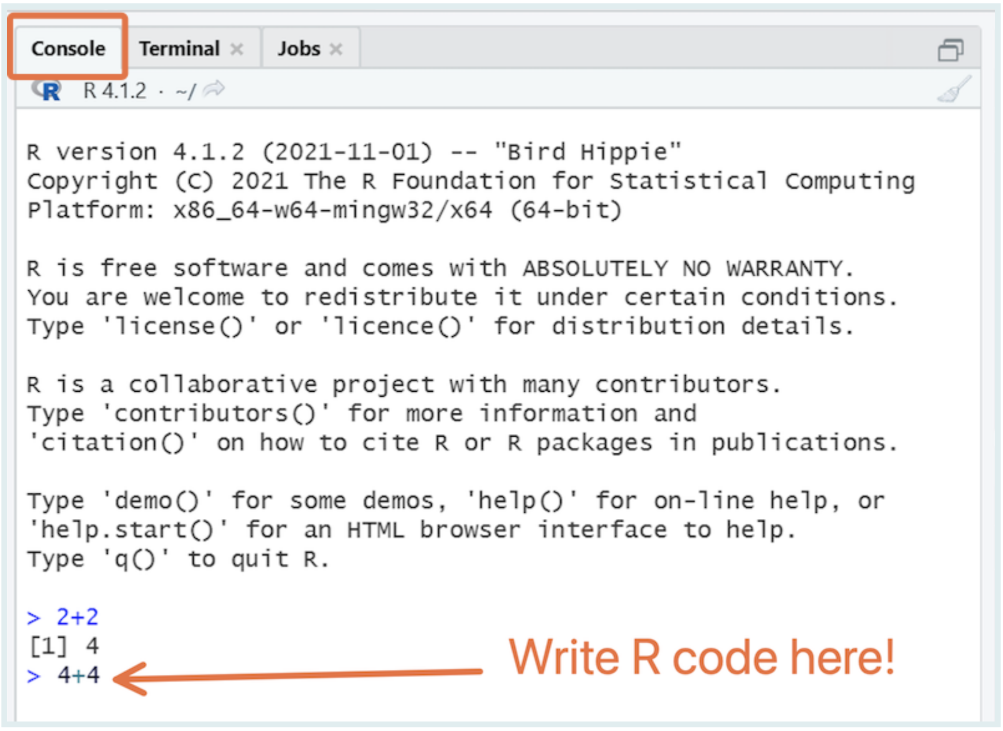
`lapply` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply` is a user-friendly version and wrapper of `lapply` by default returning a vector, matrix or, if `simplify = "array"`, an array if appropriate, by applying `simplify2array()`. `sapply(x, f, simplify = FALSE, USE.NAMES = FALSE)` is the same as `lapply(x, f)`.

`vapply` is similar to `sapply`, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

This is RStudio, your new home for a long time to come!

You can start using R by typing code into the "console" pane on the left:



```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

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

Try using R as a calculator here; type `2 + 2` and press Enter.

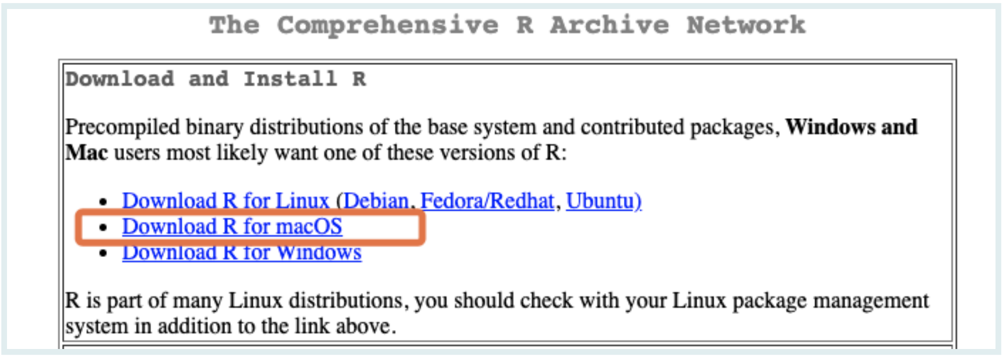
That's it; you're ready to roll. Proceed to the "wrapping up" section of the lesson.

Set up on macOS

Download and install R

If you're working on macOS, follow the steps below to download and install R:

1. Go to cran.rstudio.com to access the R installation page. Then click the link for macOS:



The Comprehensive R Archive Network

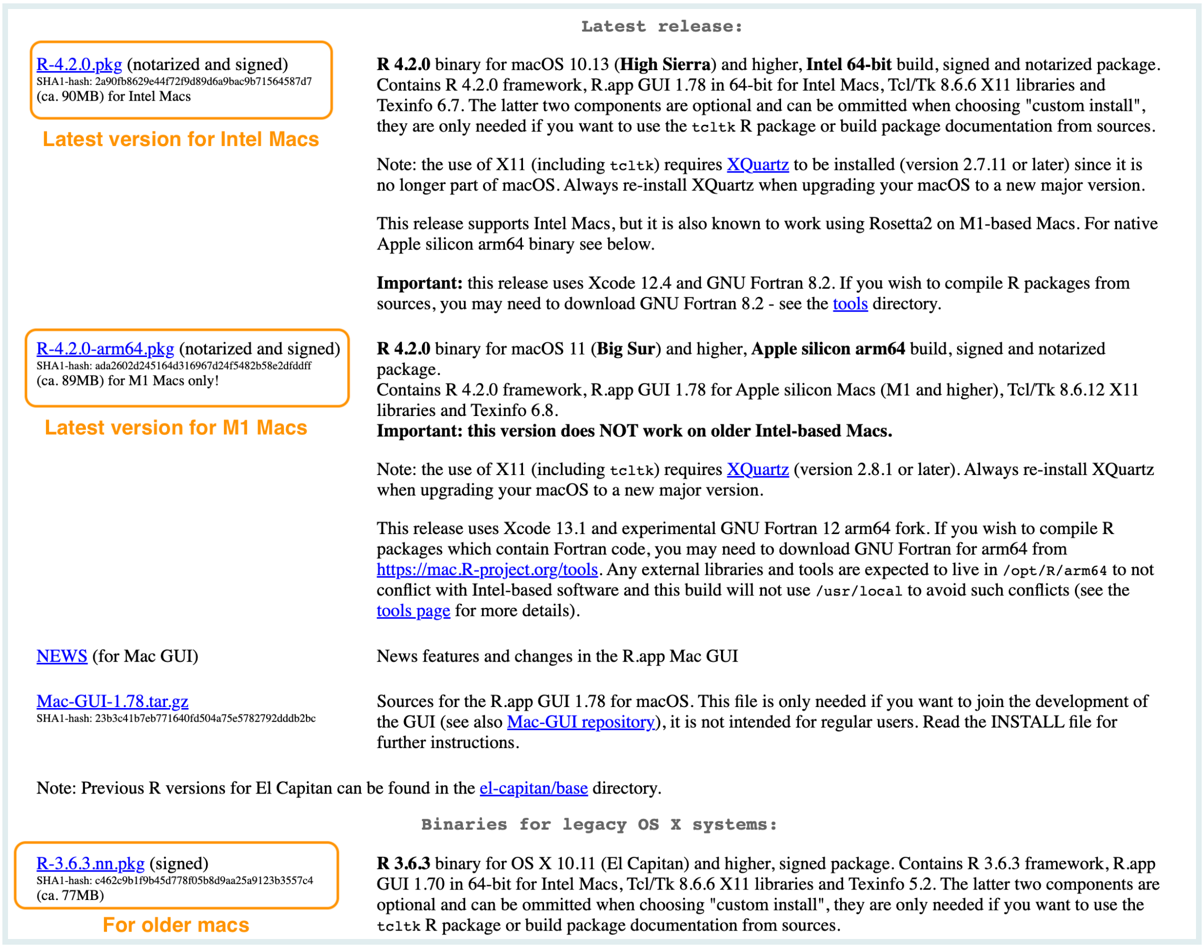
Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

2. Download and install the relevant R version for your Mac. For most people, the first option under "Latest release" will be the one to get.



Latest release:

R-4.2.0.pkg (notarized and signed)
SHA1-hash: 2a90fb8629e44f72f9d89d6a9bac9b71564587d7
(ca. 90MB) for Intel Macs

R 4.2.0 binary for macOS 10.13 (**High Sierra**) and higher, **Intel 64-bit** build, signed and notarized package. Contains R 4.2.0 framework, R.app GUI 1.78 in 64-bit for Intel Macs, Tcl/Tk 8.6.6 X11 libraries and Texinfo 6.7. The latter two components are optional and can be ommitted when choosing "custom install", they are only needed if you want to use the `tcltk` R package or build package documentation from sources.

Note: the use of X11 (including `tcltk`) requires XQuartz to be installed (version 2.7.11 or later) since it is no longer part of macOS. Always re-install XQuartz when upgrading your macOS to a new major version.

This release supports Intel Macs, but it is also known to work using Rosetta2 on M1-based Macs. For native Apple silicon arm64 binary see below.

Important: this release uses Xcode 12.4 and GNU Fortran 8.2. If you wish to compile R packages from sources, you may need to download GNU Fortran 8.2 - see the tools directory.

R-4.2.0-arm64.pkg (notarized and signed)
SHA1-hash: ada2602d245164d316967d24f5482b58e2dfddff
(ca. 89MB) for M1 Macs only!

R 4.2.0 binary for macOS 11 (**Big Sur**) and higher, **Apple silicon arm64** build, signed and notarized package.
Contains R 4.2.0 framework, R.app GUI 1.78 for Apple silicon Macs (M1 and higher), Tcl/Tk 8.6.12 X11 libraries and Texinfo 6.8.
Important: this version does NOT work on older Intel-based Macs.

Note: the use of X11 (including `tcltk`) requires XQuartz (version 2.8.1 or later). Always re-install XQuartz when upgrading your macOS to a new major version.

This release uses Xcode 13.1 and experimental GNU Fortran 12 arm64 fork. If you wish to compile R packages which contain Fortran code, you may need to download GNU Fortran for arm64 from https://mac.R-project.org/tools. Any external libraries and tools are expected to live in `/opt/R/arm64` to not conflict with Intel-based software and this build will not use `/usr/local` to avoid such conflicts (see the tools page for more details).

NEWS (for Mac GUI)

News features and changes in the R.app Mac GUI

Mac-GUI-1.78.tar.gz
SHA1-hash: 23b3c41b7eb771640fd504a75e5782792dddb2bc

Sources for the R.app GUI 1.78 for macOS. This file is only needed if you want to join the development of the GUI (see also Mac-GUI repository), it is not intended for regular users. Read the INSTALL file for further instructions.

Note: Previous R versions for El Capitan can be found in the el-capitan/base directory.

Binaries for legacy OS X systems:

R-3.6.3.nn.pkg (signed)
SHA1-hash: c462c9b1f9b45d778f05b8d9aa25a9123b3557c4
(ca. 77MB)

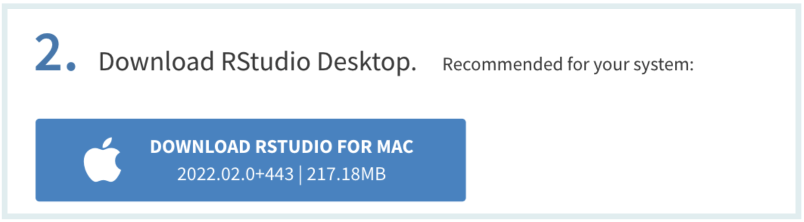
R 3.6.3 binary for OS X 10.11 (El Capitan) and higher, signed package. Contains R 3.6.3 framework, R.app GUI 1.70 in 64-bit for Intel Macs, Tcl/Tk 8.6.6 X11 libraries and Texinfo 5.2. The latter two components are optional and can be ommitted when choosing "custom install", they are only needed if you want to use the `tcltk` R package or build package documentation from sources.

3. After the download is finished, click on the downloaded file, then follow the instructions on the installation pop-up window.

Well done! You should now have R on your computer. But you likely won't ever need to interact with R directly. Instead you'll use the RStudio IDE to work with R. Follow the instructions in the next section to get RStudio.

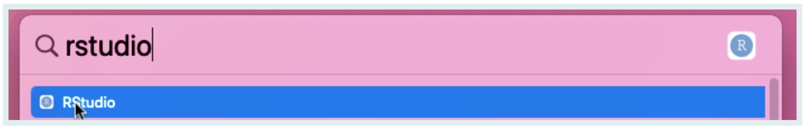
Download, install & run RStudio

To download RStudio, go to rstudio.com/products/rstudio/download/#download and download the version for macOS.

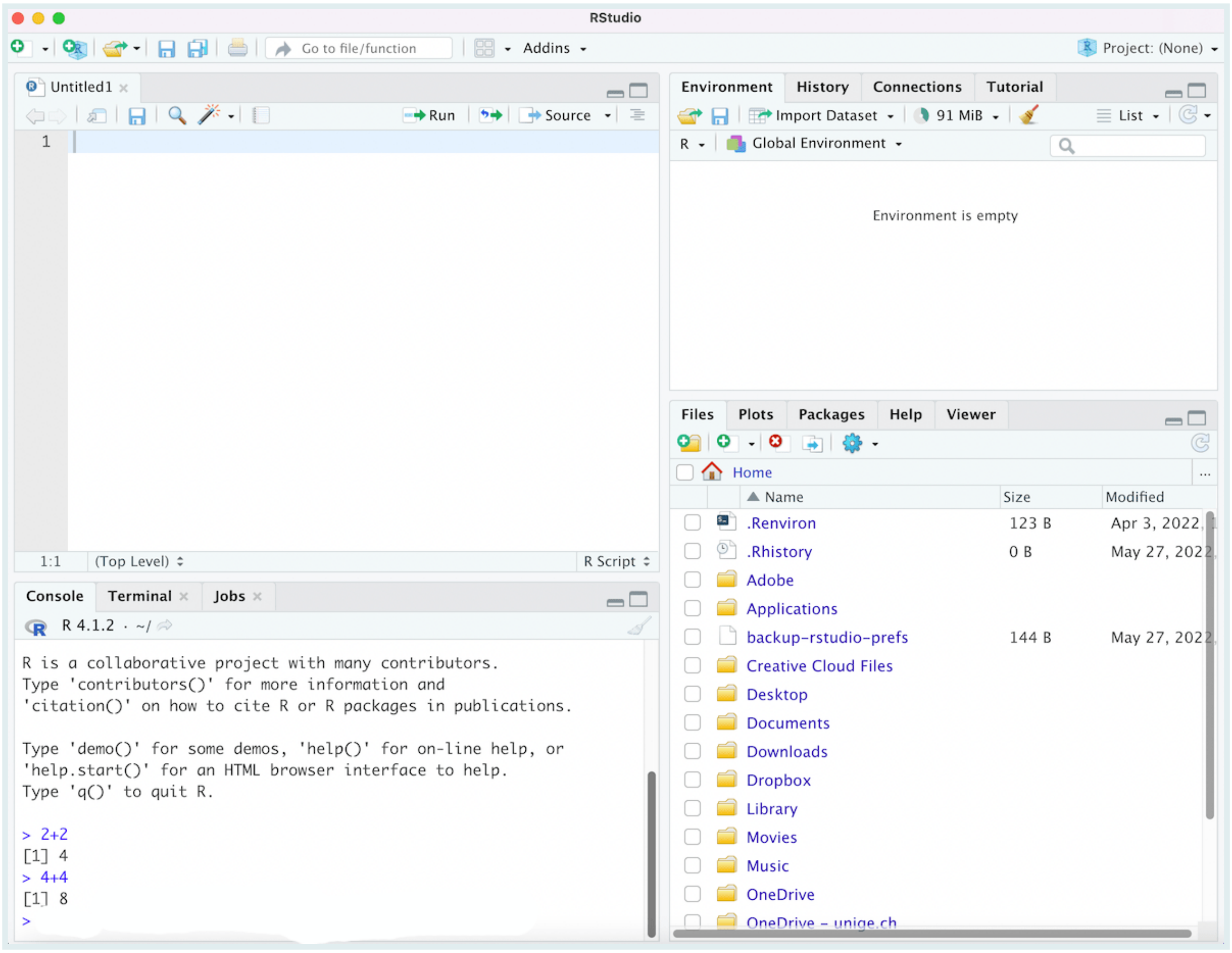


After the download is finished, click on the downloaded file and follow the installation instructions.

Once installed, RStudio can be opened like any application on your computer: Press `Command` + `Space` to open Spotlight, then search for “rstudio”. Click to open the app.



You should see a window like this:

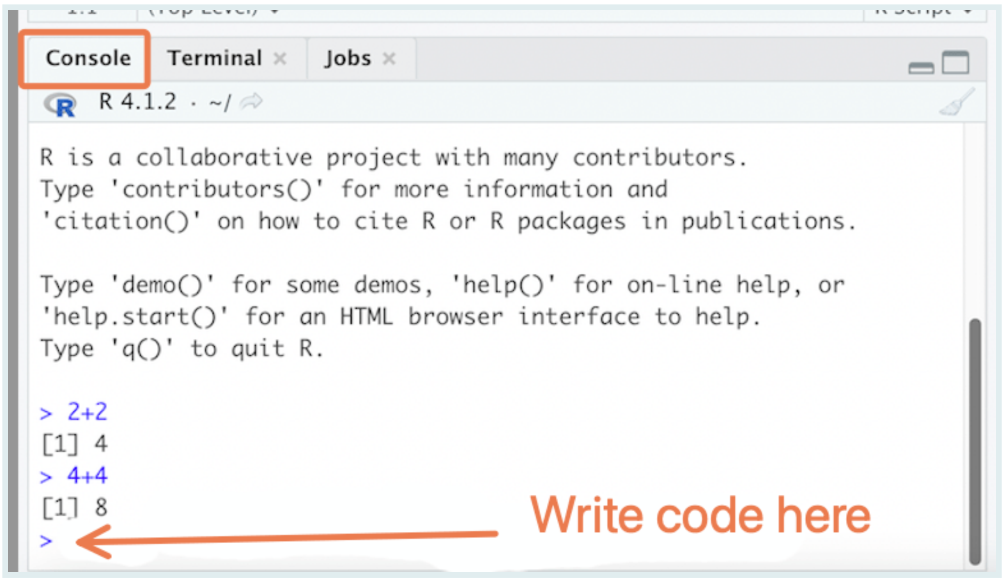


```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

> 2+2
[1] 4
> 4+4
[1] 8
>
```

Name	Size	Modified
.Renviron	123 B	Apr 3, 2022,
.Rhistory	0 B	May 27, 202
Adobe		
Applications		
backup-rstudio-prefs	144 B	May 27, 202
Creative Cloud Files		
Desktop		
Documents		
Downloads		
Dropbox		
Library		
Movies		
Music		
OneDrive		
OneDrive – unige.ch		

This is RStudio, your new home for a long time to come!

You can start using R by typing code into the “console” pane on the left:



```
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.

> 2+2
[1] 4
> 4+4
[1] 8
>
```

Try using R as a calculator here; type `2 + 2` and press Enter.

Wrap up

You should now have access to R and RStudio, so you're all set to begin the journey of learning to use these immensely powerful tools. See you in the next session!

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAMECK AGASA

Statistician/Data Scientist

MICHAL SHRESTHA

Global Health Researcher, the GRAPH Network
An advocate of health equity & justice through equal access to health data

ELTON MUKONDA

Data analyst, the GRAPH Network
A data enthusiast with a passion for population health research

OLIVIA KEISER

Head of division of Infectious Diseases and Mathematical Modelling, University of Geneva

References

Some material in this lesson was adapted from the following sources:

- Nordmann, Emily, and Heather Cleland-Woods. *Chapter 2 Programming Basics | Data Skills*. *psyteachr.github.io*, https://psyteachr.github.io/data-skills-v1/programming-basics.html Accessed 23 Feb. 2022.

Lesson notes | Using RStudio

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

---

Learning objectives

1. You can identify and use the following tabs in RStudio: Source, Console, Environment, History, Files, Plots, Packages, Help and Viewer.

2. You can modify RStudio's interface options to suit your needs.

Introduction

Now that you have access to R & RStudio, let's go on a quick tour of the RStudio interface, your digital home for a long time to come.

We will cover a lot of territory quickly. Do not panic. You are not expected to remember it all this. Rather, you will see these topics again and again throughout the course, and you will naturally assimilate them that way.

You can also refer back to this lesson as you progress.

The goal here is simply to make you aware of the tools at your disposal within RStudio.

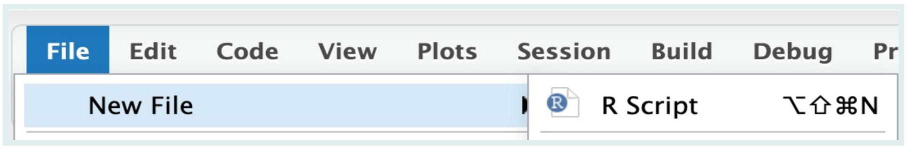
To get started, you need to open the RStudio application:

- If you are working with RStudio Cloud, go to rstudio.cloud, log in, then click on the "r_intro" project that you created in the last lesson. (If you do not see this, simply create a new R project using the "New Project" icon at the top right).
- If you are working on your local computer, go to your applications folder and double click on the RStudio icon. Or you search for this application from your Start Menu (Windows), or through Spotlight (Mac).

The RStudio panes

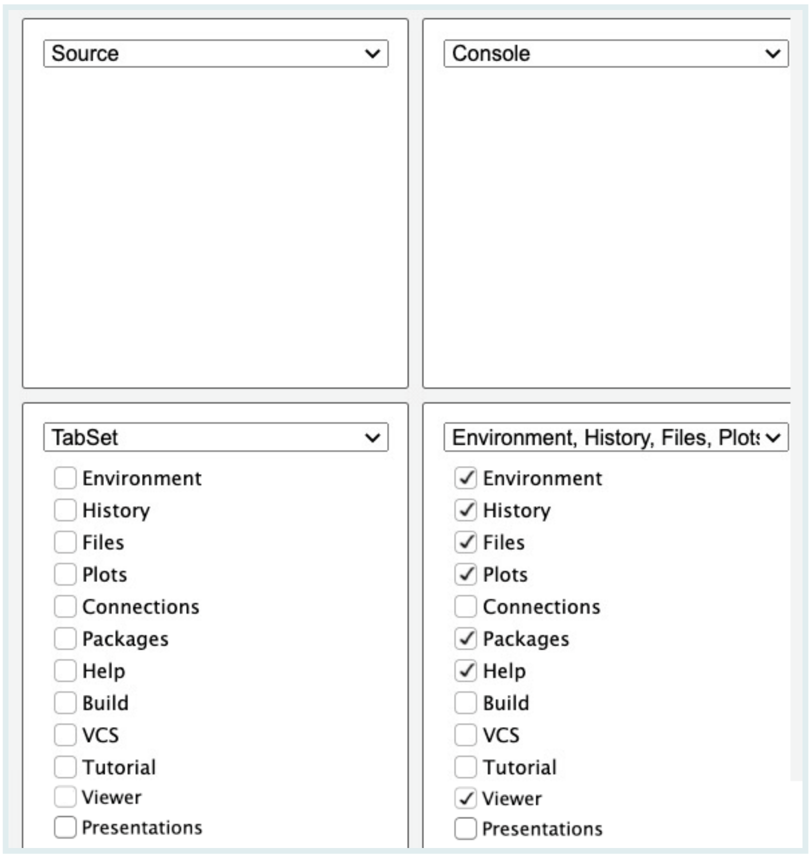
By default, RStudio is arranged into four window panes.

If you only see three panes, open a new script with `File > New File > R Script` . This should reveal one more pane.



Before we go any further, we will rearrange these panes to improve the usability of the interface.

To do this, in the RStudio menu at the top of the screen, select `Tools > Global Options` to bring up RStudio's options. Then under `Pane Layout`, adjust the pane arrangement. The arrangement we recommend is shown below.



Source

Console

TabSet
- [ ] Environment
- [ ] History
- [ ] Files
- [ ] Plots
- [ ] Connections
- [ ] Packages
- [ ] Help
- [ ] Build
- [ ] VCS
- [ ] Tutorial
- [ ] Viewer
- [ ] Presentations

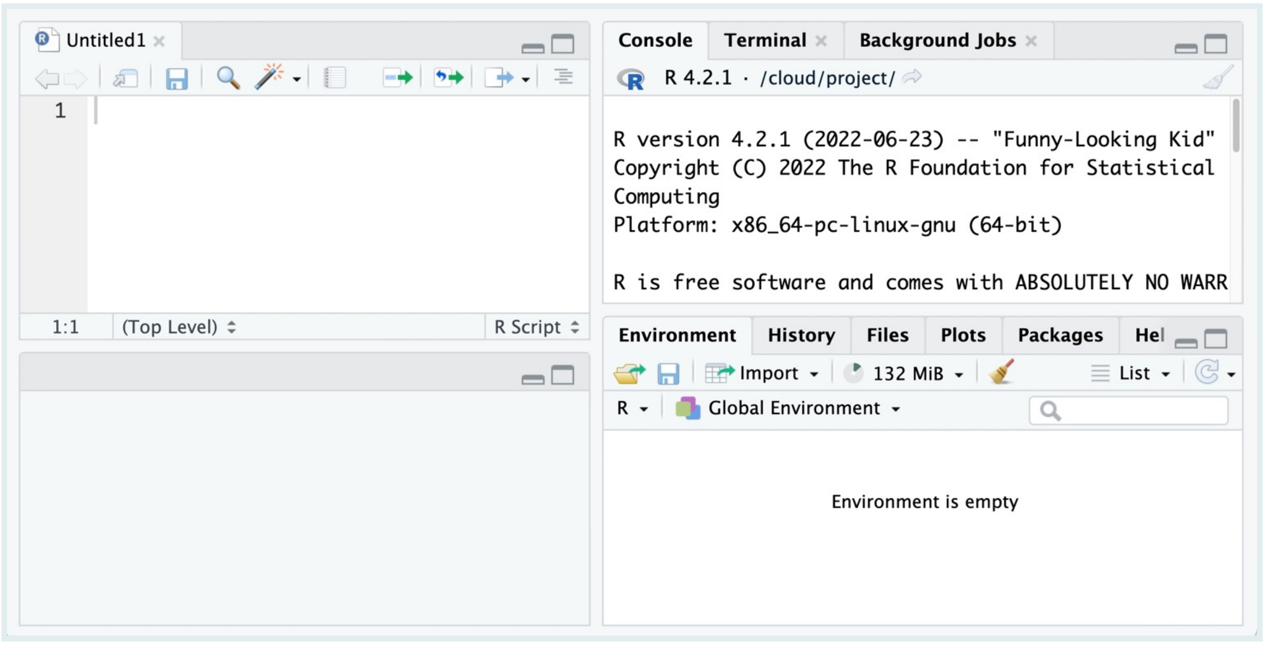
Environment, History, Files, Plot
- [✓] Environment
- [✓] History
- [✓] Files
- [✓] Plots
- [ ] Connections
- [✓] Packages
- [✓] Help
- [ ] Build
- [ ] VCS
- [ ] Tutorial
- [✓] Viewer
- [ ] Presentations

At the top left pane is the Source tab, and at the top right pane, you should have the Console tab.

Then at the bottom left pane, no tab options should checked—this section should be left empty, with the drop-down saying just "TabSet".

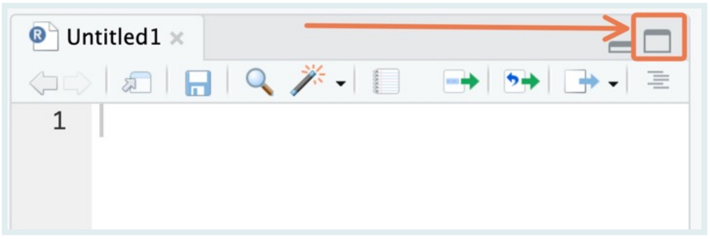
Finally, at the bottom right pane, you should check the following tabs: Environment, History, Files, Plots, Packages, Help and Viewer.

Great, now you should have an RStudio window that looks something like this:

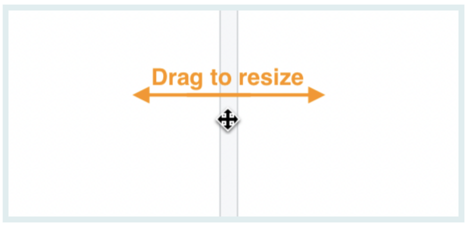


```
R 4.2.1 · /cloud/project/

R version 4.2.1 (2022-06-23) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical
Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARR
```

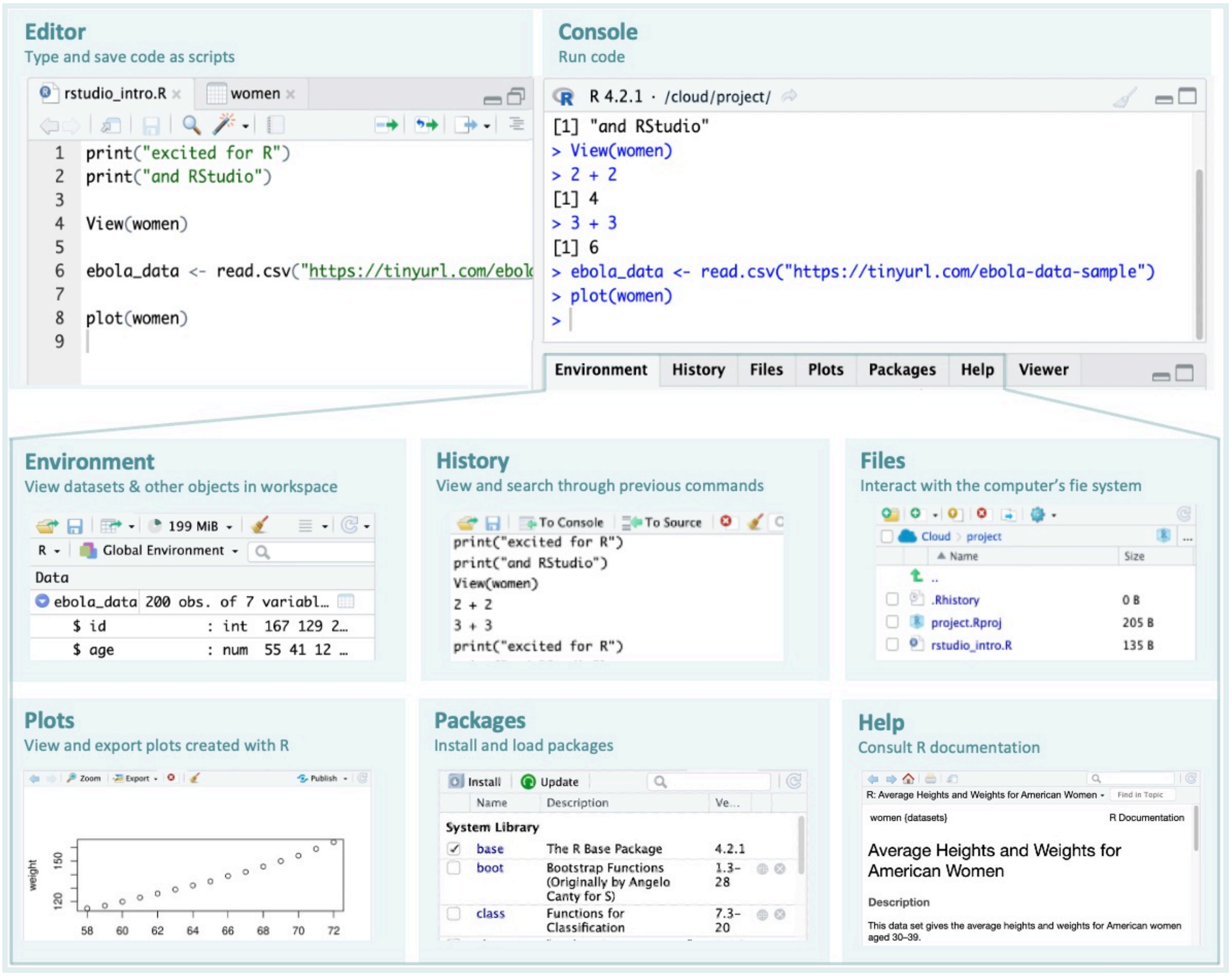
The top-left pane is where you will do most of the coding. Make this larger by clicking on its maximize icon:



Note that you can drag the bar that separates the window panes to resize them.



Now let's look at each of the RStudio tabs one by one. Below is a summary image of what we will discuss:


Editor
Type and save code as scripts
rstudio_intro.R
women
Console
Run code
R 4.2.1 · /cloud/project/
Environment
History
Files
Plots
Packages
Help
Viewer
Environment
View datasets & other objects in workspace
History
View and search through previous commands
Files
Interact with the computer's fie system
Plots
View and export plots created with R
Packages
Install and load packages
Help
Consult R documentation

```
print("excited for R")
print("and RStudio")

View(women)

ebola_data <- read.csv("https://tinyurl.com/ebol

plot(women)

```

```
[1] "and RStudio"
> View(women)
> 2 + 2
[1] 4
> 3 + 3
[1] 6
> ebola_data <- read.csv("https://tinyurl.com/ebola-data-sample")
> plot(women)
>
```

```
199 MiB
R   Global Environment
Data
ebola_data 200 obs. of 7 variabl…
  $ id    : int  167 129 2…
  $ age   : num  55 41 12 …
```

```
To Console   To Source
print("excited for R")
print("and RStudio")
View(women)
2 + 2
3 + 3
print("excited for R")
```

```
Cloud > project
  Name              Size
  ..
  .Rhistory         0 B
  project.Rproj     205 B
  rstudio_intro.R   135 B
```

	Name	Description	Ve…
	System Library		
✓	base	The R Base Package	4.2.1
	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-28
	class	Functions for Classification	7.3-20

R: Average Heights and Weights for American Women

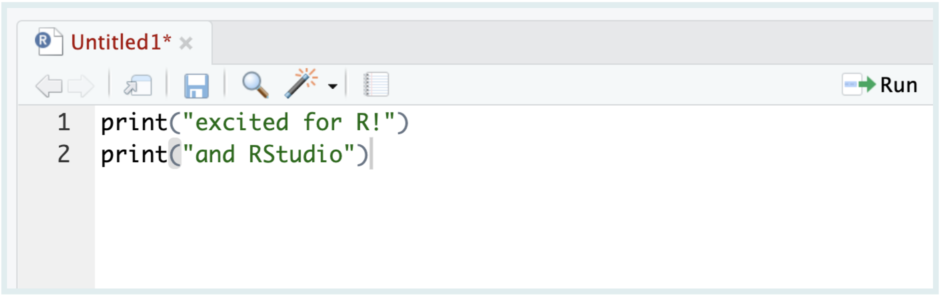
women {datasets} R Documentation

Average Heights and Weights for American Women

Description

This data set gives the average heights and weights for American women aged 30–39.

Source/Editor



```
print("excited for R!")
print("and RStudio")
```

The source or editor is where your R “scripts” go. A script is a text document where you write and save code.

Because this is where you will do most of your coding, it is important that you have a lot of visual space. That is why we rearranged the RStudio pane layout above—to give the Editor more space.

Now let’s see how to use this Editor.

First, **open a new script** under the File menu if one is not yet open: `File > New File > R Script`. In the script, type the following:

```
print("excited for R!")
```

To **run code**, place your cursor anywhere in the code, then hit `Command` + `Enter` on macOS, or `Control` + `Enter` on Windows.

This should send the code to the Console and run it.

You can also **run multiple lines at once**. To try this, add a second line to your script, so that it now reads:

```
print("excited for R!")
print("and RStudio!")
```

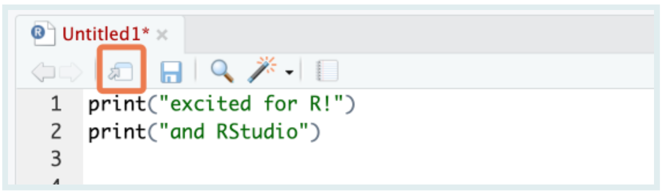
Now drag your cursor to highlight both lines and press `Command/Control` + `Enter`.

To **run the entire script**, you can use `Command/Control` + `A` to select all code, then press `Command/Control` + `Enter`. Try this now. Deselect your code, then try to the shortcut to select all.

SIDE NOTE

There is also a 'Run' button at the top right of the source panel (Run), with which you can run code (either the current line, or all highlighted code). But you should try to use the keyboard shortcut instead.

To **open the script in a new window**, click on the third icon in the toolbar directly above the script.



```
print("excited for R!")
print("and RStudio")
```

To put the window back, click on the same button on the now-external window.

Next, **save the script.** Hit `Command/Control` + `S` to bring up the Save dialog box. Give it a file name like "rstudio_intro".

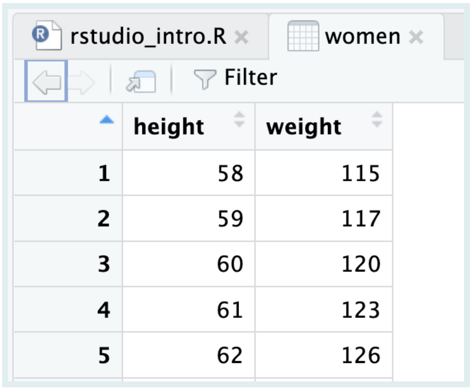
- If you are working with RStudio cloud, the file will be saved in your project folder.

- If you are working on your local computer, save the file in an easy-to-locate part of your computer, perhaps your desktop. (Later on we will think about the "proper" way to organize and store scripts).

You can **view data frames** (which are like spreadsheets in R) in the same pane. To observe this, type and run the code below on a new line in your script:

```
View(women)
```

Notice the uppercase "V" in `View()`.



	height	weight
1	58	115
2	59	117
3	60	120
4	61	123
5	62	126

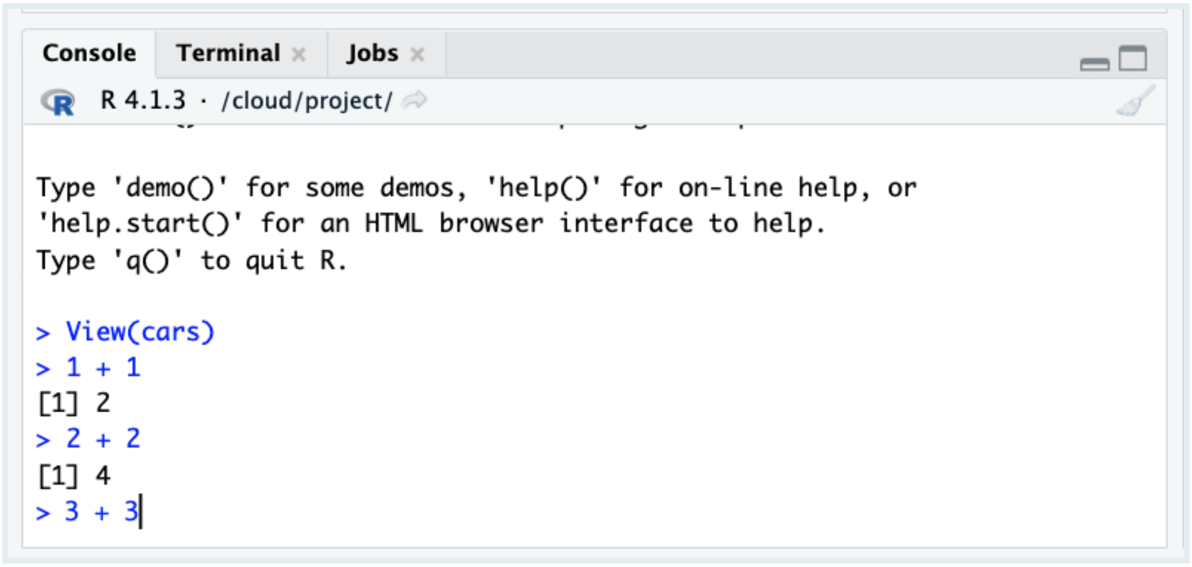
`women` is the name of a dataset that comes loaded with R. It gives the average heights and weights for American women aged 30-39.

You can click on the "x" icon to the right of the "women" tab to close this data viewer.

Console

The *console*, at the bottom left, is where **code is executed**. You can type code directly here, but it will not be saved.

Type a random piece of code (maybe a calculation like `3 + 3`) and press 'Enter'.

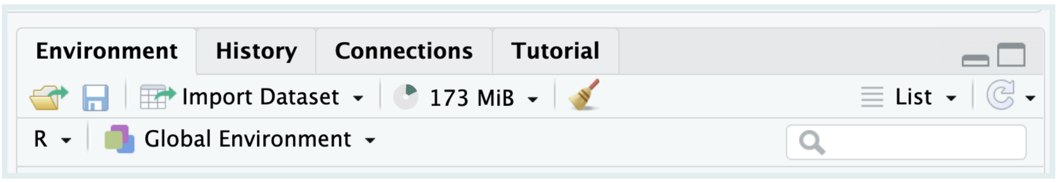


```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> View(cars)
> 1 + 1
[1] 2
> 2 + 2
[1] 4
> 3 + 3
```

If you place your cursor on the last line of the console, and you press the **up arrow**, you can go back to the last code that was run. Keep pressing it to cycle to the previous lines.

To run any of these previous lines, press *Enter*.

Environment



At the top right of the RStudio Window, you should see the **Environment** tab.

The Environment tab shows datasets and other objects that are loaded into R's working memory, or "workspace".

To explore this tab, let's import a dataset into your environment from the web. Type the code below into your script and run it:

```
ebola_data <- read.csv("https://tinyurl.com/ebola-data-sample")
```

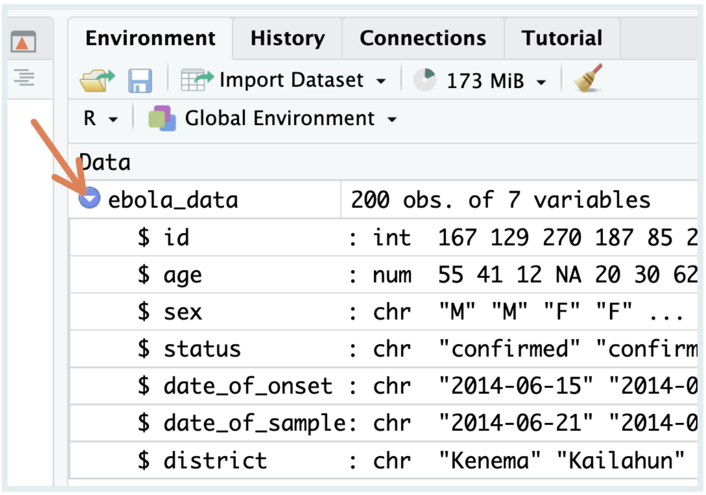
SIDE NOTE

You don't need to understand exactly what the code above is doing for now. We just want to quickly show you the basic features of the Environment pane; we'll look at data importing in detail later.

Also, if you do not have active internet access, the code above will not run. You can skip this section and move to the "History" tab.

You have now imported the dataset and stored it in an *object* named `ebola_data`. (You could have named the object anything you want.)

Now that the dataset is stored by R, you should be able to see it in the Environment pane. If you click on the blue drop-down icon beside the object's name in the Environment tab to reveal a summary.



```
ebola_data              200 obs. of 7 variables
  $ id            : int  167 129 270 187 85 2
  $ age           : num  55 41 12 NA 20 30 62
  $ sex           : chr  "M" "M" "F" "F" ...
  $ status        : chr  "confirmed" "confirm
  $ date_of_onset : chr  "2014-06-15" "2014-0
  $ date_of_sample: chr  "2014-06-21" "2014-0
  $ district      : chr  "Kenema" "Kailahun"
```

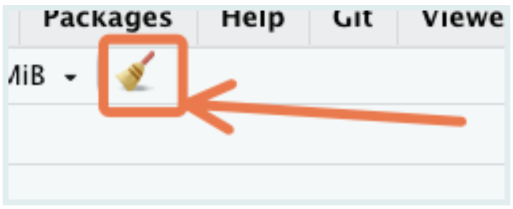
Try clicking directly on the `ebola_data` dataset from the Environment tab. This opens it in a 'View' tab.

You can **remove an object from the workspace** with the `rm()` function. Type and run the following in a new line on your R script.

```
rm(ebola_data)
```

Notice that the `ebola_data` object no longer shows up in your environment after having run that code.

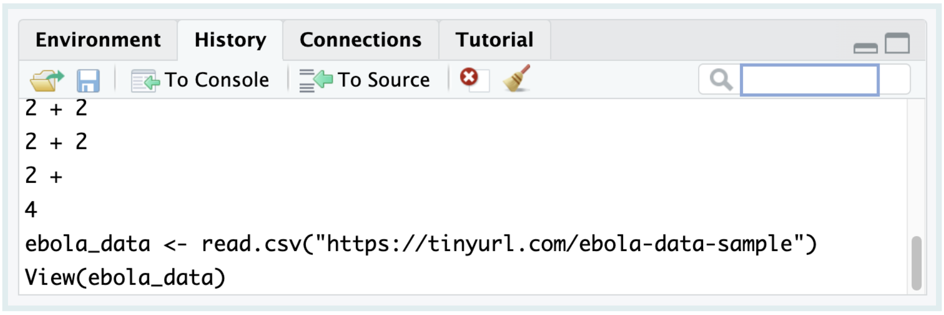
The broom icon, at the top of the Environment pane can also be used to clear your workspace.



To practice using it, try re-running the line above that imports the Ebola dataset, then clear the object using the broom icon.

History

Next, the **History** tab shows previous commands you have run.



```
2 + 2
2 + 2
2 +
4
ebola_data <- read.csv("https://tinyurl.com/ebola-data-sample")
View(ebola_data)
```

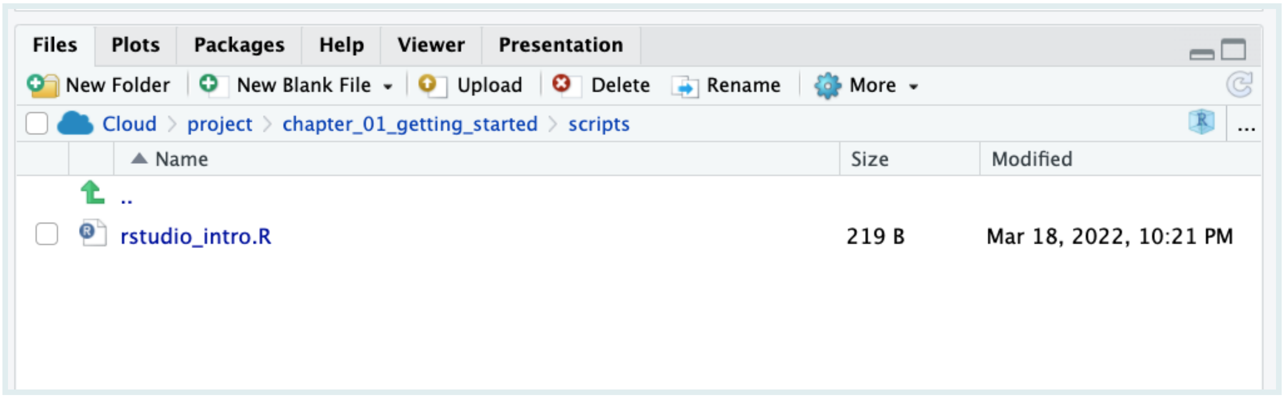
You can click a line to highlight it, then send it to the console or to your script with the "To Console" and "To Source" icons at the top of this tab.

To select multiple lines, use the "Shift-click" method: click the first item you want to select, then hold down the "Shift" key and click the last item you want to select.

Finally, notice that there is a search bar at the top right of the History pane where you can search for past commands that you have run.

Files

Next, the **Files** tab. This shows the files and folders in the folder you are working in.



Name	Size	Modified
..		
rstudio_intro.R	219 B	Mar 18, 2022, 10:21 PM

The tab allows you to interact with your computer's file system.

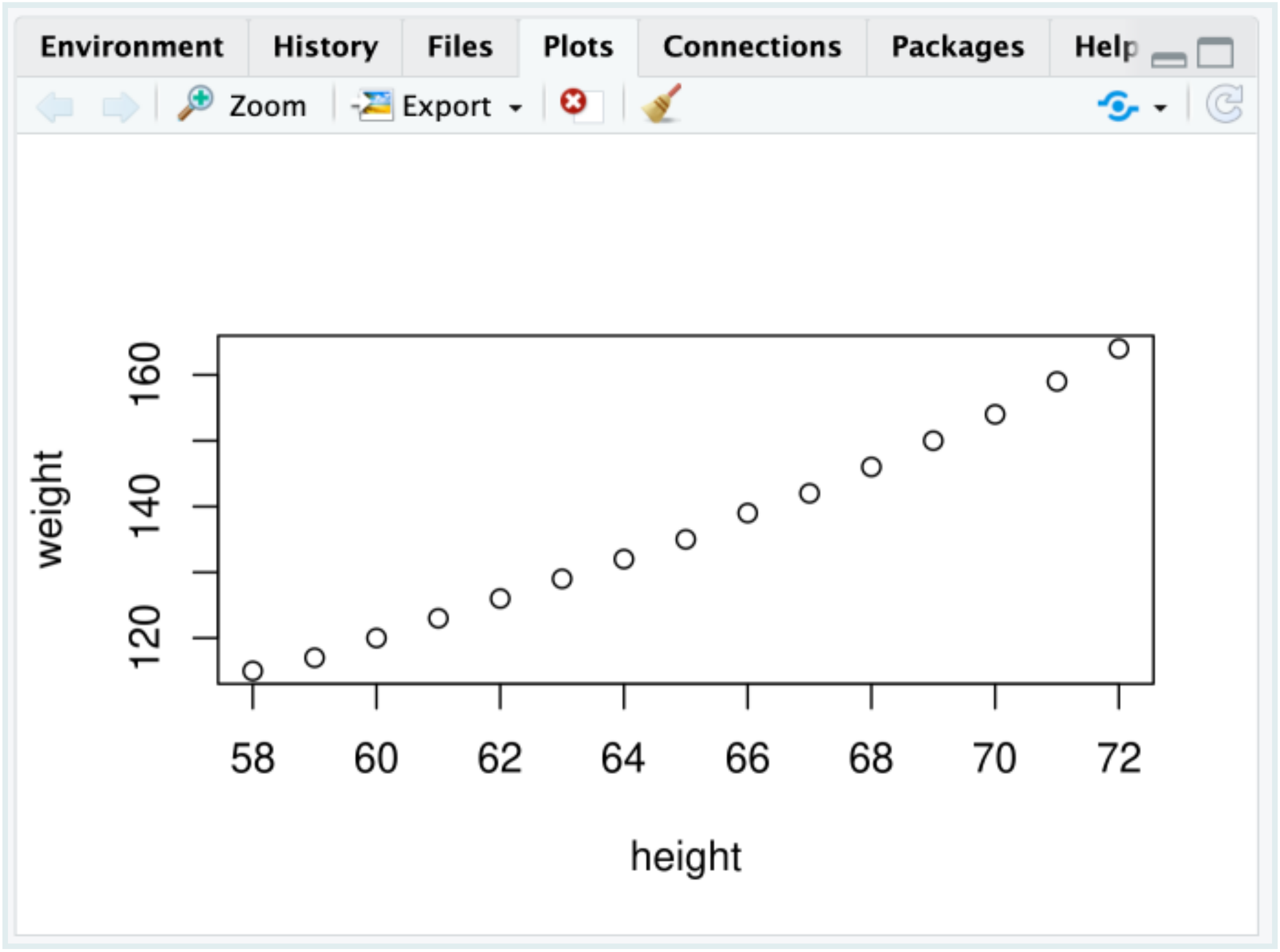
Try playing with some of the buttons here, to see what they do. You should try at least the following:

- Make a new folder
- Delete that folder
- Make a new R Script
- Rename that script

Plots

Next, the **Plots** tab. This is where figures that are generated by R will show up. Try creating a simple plot with the following code:

```
plot(women)
```

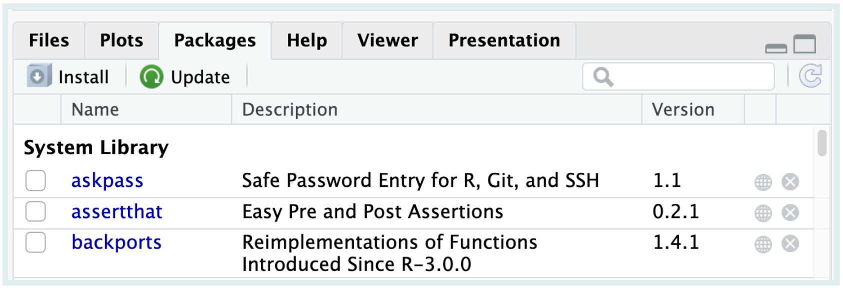


That code creates a plot of the two variables in the `women` dataset. You should see this figure in the Plots tab.

Now, test out the buttons at the top of this tab to explore what they do. In particular, try to export a plot to your computer.

Packages

Next, let's look at the **Packages** tab.



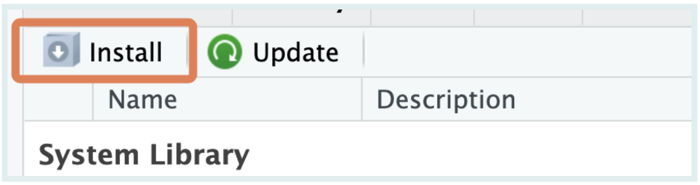
	Name	Description	Version
	System Library		
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.4.1

Packages are collections of R code that extend the functionality of R. We will discuss packages in detail in a future lesson.

For now, it is important to know that to use a package, you need to *install* then *load* it. Packages need to be installed only once, but must be loaded in each new R session.

All the package names you see (in blue font) are packages that are installed on your system. And packages with a checkmark are packages which are *loaded* in the current session.

You can install a package with the Install button of the Packages tab.



But it is better to install and load packages with R code, rather than the Install button. Let's try this. Type and run the code below to install the {highcharter} package.

```r
install.packages("highcharter")
library(highcharter)
```

The first line installs the package. The second line *loads* the package from your package library.

Because you only need to install a package once, you can now remove the installation line from your script.

Now that the {highcharter} package has been installed and loaded, you can use the functions that come in the package. To try this, type and run the code below:

```r
highcharter::hchart(women$weight)
```

This code uses the `hchart()` *function* from the {highcharter} package to plot an interactive histogram showing the distribution of weights in the `women` dataset.

(Of course, you may not yet know what a function is. We'll get to this soon.)

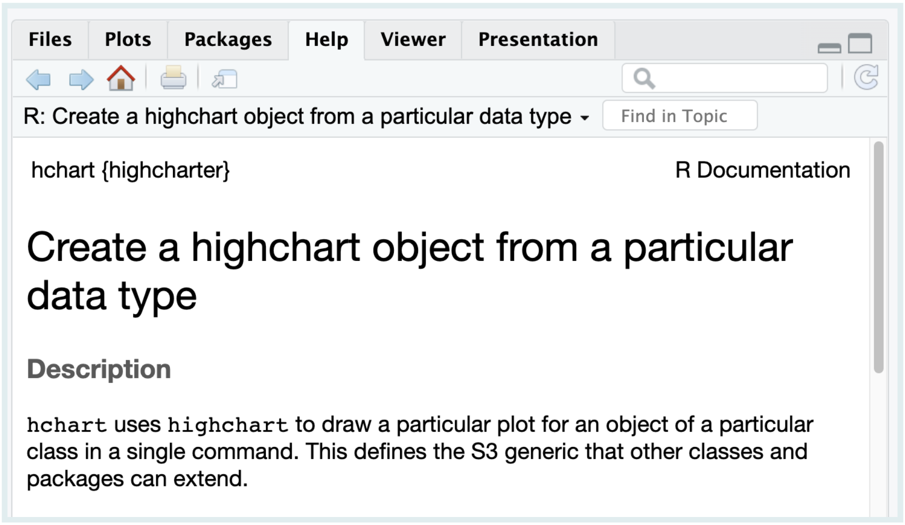
Viewer

Notice that the histogram above shows up in a **Viewer** tab. This tab allows you to preview HTML files and interactive objects.

Help

Lastly, the **Help** tab shows the documentation for different R objects. Try typing out and running each line below to see what this documentation looks like.

```
?hchart
?women
?read.csv
```



hchart {highcharter} R Documentation

Create a highchart object from a particular data type

Description

`hchart` uses `highchart` to draw a particular plot for an object of a particular class in a single command. This defines the S3 generic that other classes and packages can extend.

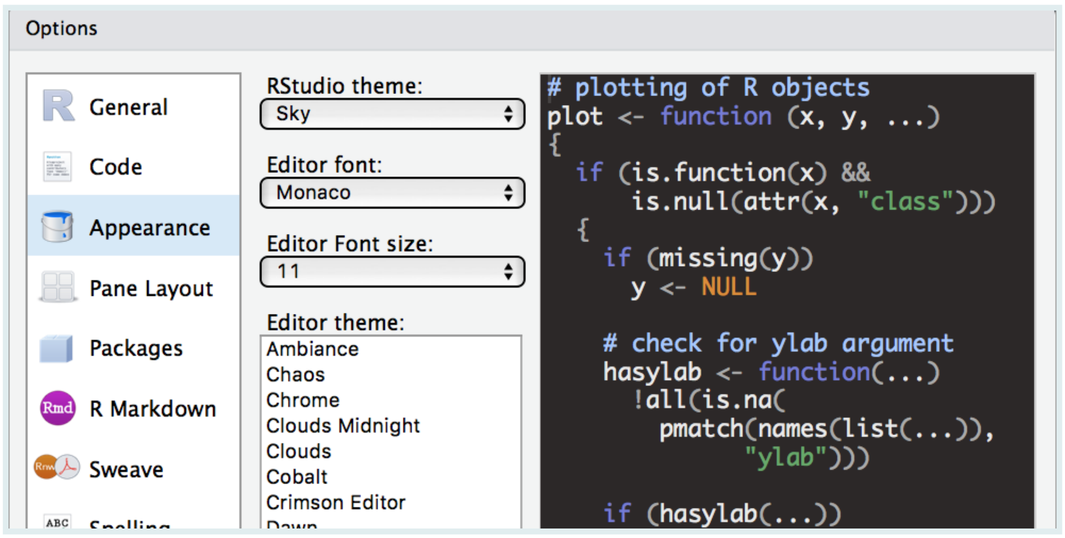
Help files are not always very easy to understand for beginners, but with time they will become more useful.

RStudio options

RStudio has a number of useful options for changing it's look and functionality. Let's try these. You may not understand all the changes made for now. That's fine.

In the RStudio menu at the top of the screen, select `Tools > Global Options` to bring up RStudio's options.

- Now, under `Appearance`, choose your ideal theme. (We like the "Crimson Editor" and "Tomorrow Night" themes.)


Options

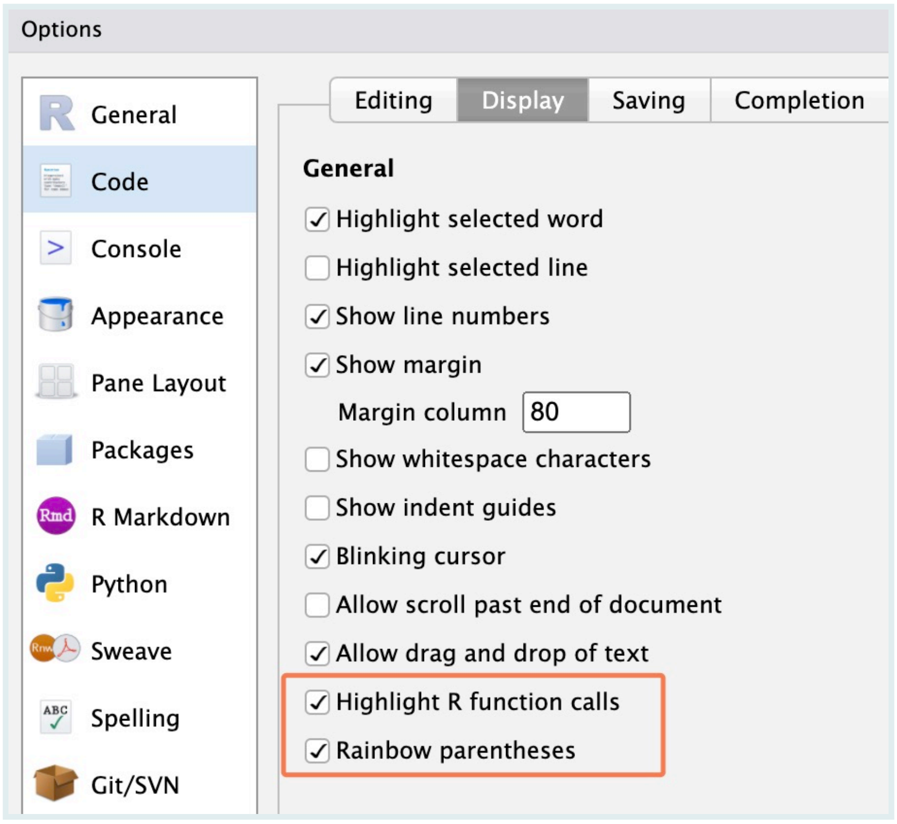
General
Code
Appearance
Pane Layout
Packages
R Markdown
Sweave

RStudio theme: Sky
Editor font: Monaco
Editor Font size: 11
Editor theme:
Ambiance
Chaos
Chrome
Clouds Midnight
Clouds
Cobalt
Crimson Editor

```
# plotting of R objects
plot <- function (x, y, ...)
{
  if (is.function(x) &&
      is.null(attr(x, "class")))
  {
    if (missing(y))
      y <- NULL

    # check for ylab argument
    hasylab <- function(...)
      !all(is.na(
        pmatch(names(list(...)),
               "ylab")))

    if (hasylab(...))
```

- Under `Code > Display`, check "Highlight R function calls". What this does is give your R *functions* a unique color, improving readability. You will understand this later.

- Also under `Code > Display`, check "Rainbow parentheses". What this does is make your "nested parentheses" easier to read by giving each pair a unique color.


Options

General
Code
Console
Appearance
Pane Layout
Packages
R Markdown
Python
Sweave
Spelling
Git/SVN

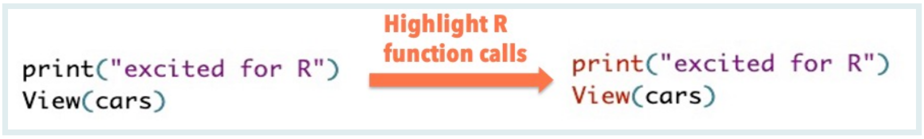
Editing | Display | Saving | Completion

General

- [x] Highlight selected word
- [ ] Highlight selected line
- [x] Show line numbers
- [x] Show margin

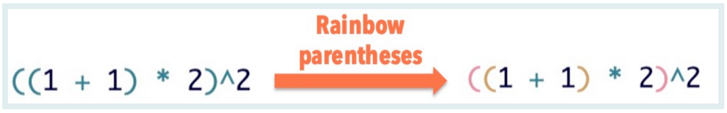
Margin column 80

- [ ] Show whitespace characters
- [ ] Show indent guides
- [x] Blinking cursor
- [ ] Allow scroll past end of document
- [x] Allow drag and drop of text
- [x] Highlight R function calls
- [x] Rainbow parentheses



```
print("excited for R")
View(cars)
```

```
print("excited for R")
View(cars)
```



```
((1 + 1) * 2)^2
```

```
((1 + 1) * 2)^2
```

- Finally under `General > Basic`, **uncheck** the box that says **"Restore .RData into workspace at startup"**. You don't want to restore any data to your workspace (or *environment)* when you start RStudio. Starting with a clean workspace each time is less likely to lead to errors.

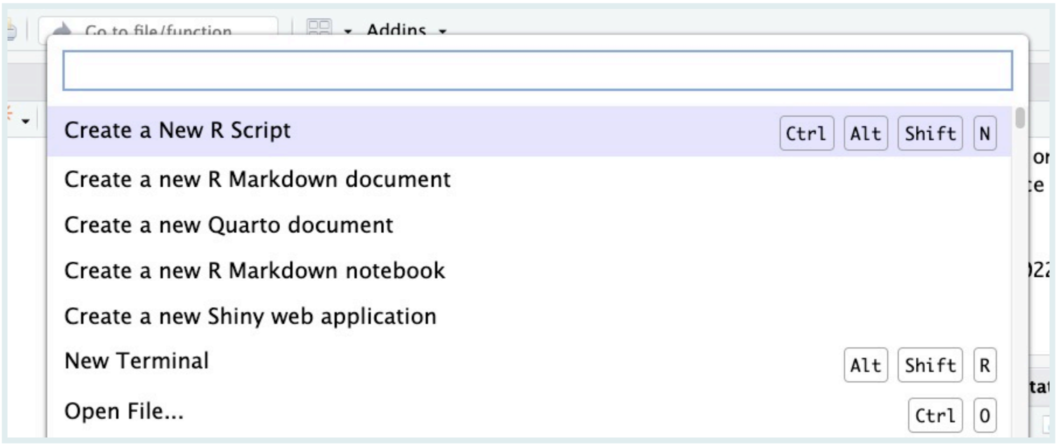
  This also means that you never want to **"save your workspace to .RData on exit"**, so set this to **Never**.

Command palette

The Rstudio command palette gives instant, searchable access to many of the RStudio menu options and settings that we have seen so far.

The palette can be invoked with the keyboard shortcut `Ctrl` + `Shift` + `P` (`Cmd` + `Shift` + `P` on macOS).

It's also available on the *Tools* menu (*Tools* -> *Show Command Palette*).


Go to file/function
Addins
Create a New R Script
Ctrl Alt Shift N
Create a new R Markdown document
Create a new Quarto document
Create a new R Markdown notebook
Create a new Shiny web application
New Terminal
Alt Shift R
Open File...
Ctrl O

Try using it to:

- Create a new script (Search "new script" and click on the relevant option)

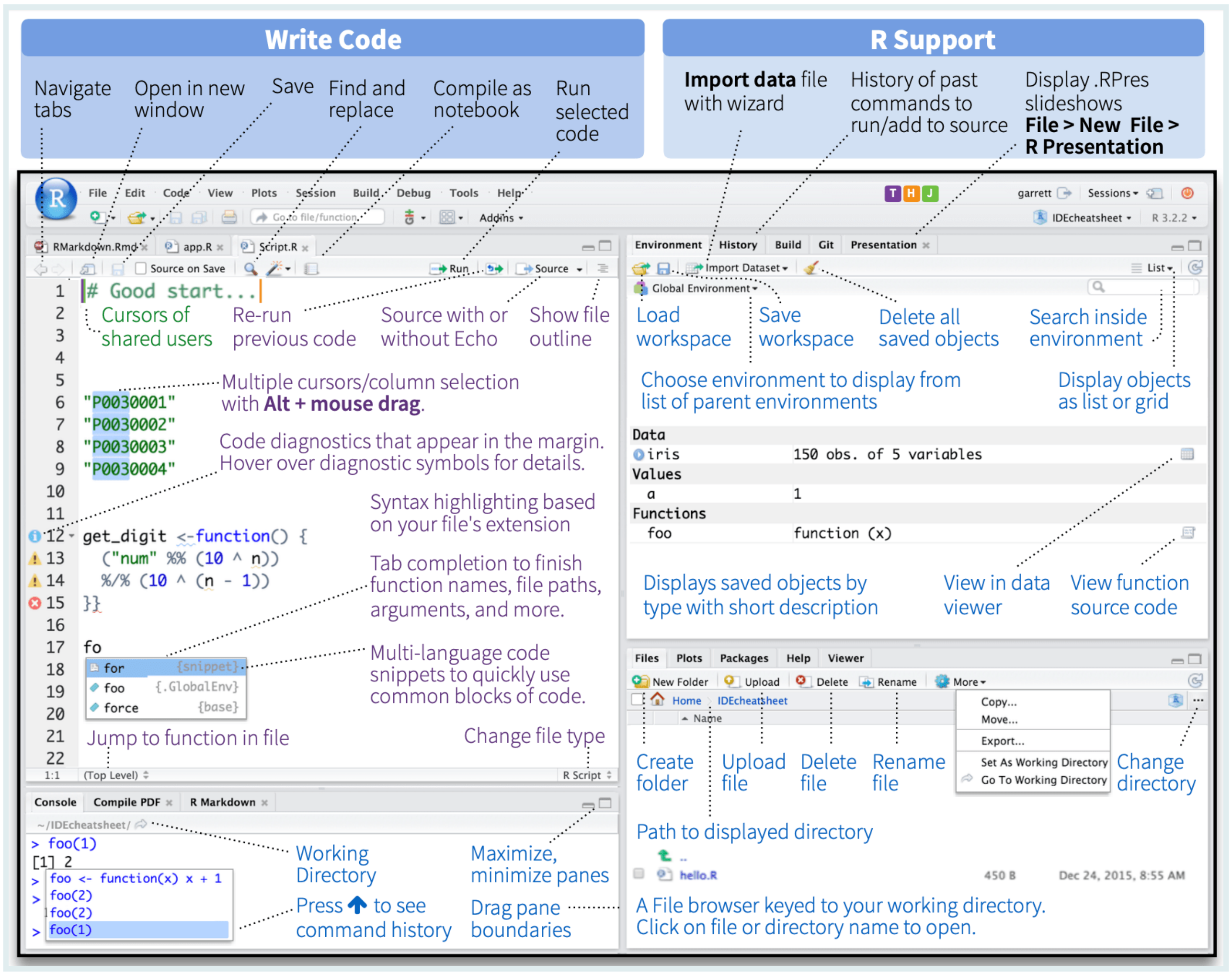
- Rename a script (Search "rename" and click on the relevant option)

Wrapping up

Congratulations! You are now a new citizen of RStudio.

Of course, you have only scratched the surface of RStudio functionality. As you advance in your R journey, you will discover new features, and you will hopefully grow to love the wonderful integrated development environment (IDE) that is RStudio. One good place to start is the official RStudio IDE cheatsheet.

Below is one section of that sheet:


Write Code
Navigate tabs
Open in new window
Save
Find and replace
Compile as notebook
Run selected code
R Support
Import data file with wizard
History of past commands to run/add to source
Display .RPres slideshows File > New File > R Presentation
Cursors of shared users
Re-run previous code
Source with or without Echo
Show file outline
Multiple cursors/column selection with Alt + mouse drag.
Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
Syntax highlighting based on your file's extension
Tab completion to finish function names, file paths, arguments, and more.
Multi-language code snippets to quickly use common blocks of code.
Jump to function in file
Change file type
Working Directory
Press ↑ to see command history
Maximize, minimize panes
Drag pane boundaries
Load workspace
Save workspace
Delete all saved objects
Search inside environment
Choose environment to display from list of parent environments
Display objects as list or grid
Displays saved objects by type with short description
View in data viewer
View function source code
Create folder
Upload file
Delete file
Rename file
Change directory
Path to displayed directory
A File browser keyed to your working directory. Click on file or directory name to open.
File
Edit
Code
View
Plots
Session
Build
Debug
Tools
Help
Addins
RMarkdown.Rmd
app.R
Script.R
Source on Save
Run
Source
Environment
History
Build
Git
Presentation
Import Dataset
Global Environment
List
Console
Compile PDF
R Markdown
Files
Plots
Packages
Help
Viewer
New Folder
Upload
Delete
Rename
More
Home
IDEcheatsheet
Name
Copy...
Move...
Export...
Set As Working Directory
Go To Working Directory
garrett
Sessions
R 3.2.2
(Top Level)
R Script
~/IDEcheatsheet/

```
# Good start...

"P0030001"
"P0030002"
"P0030003"
"P0030004"

get_digit <-function() {
  ("num" %% (10 ^ n))
  %/% (10 ^ (n - 1))
}

fo
  for   {snippet}
  foo   {.GlobalEnv}
  force {base}
```

```
> foo(1)
[1] 2
> foo <- function(x) x + 1
> foo(2)
  foo(2)
> foo(1)
```

```
Data
iris        150 obs. of 5 variables
Values
a           1
Functions
foo         function (x)
```

```
hello.R     450 B     Dec 24, 2015, 8:55 AM
```

See you in the next lesson!

Further resources

1. 23 RStudio Tips, Tricks, and Shortcuts

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAMECK AGASA

Statistician/Data Scientist

References

Some material in this lesson was adapted from the following sources:

- "Rstudio Cheatsheets." *RStudio*, https://www.rstudio.com/resources/cheatsheets/.
- "Chapter 1 Getting Started: Data Skills for Reproducible Research." *Chapter 1 Getting Started | Data Skills for Reproducible Research*, https://psyteachr.github.io/reprores-v2/intro.html.

Lesson notes | Coding basics

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

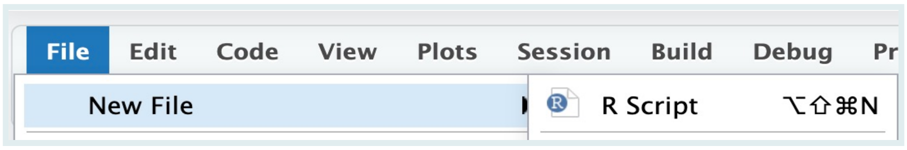
Learning objectives

1. You can write comments in R.
2. You can create section headers in RStudio.
3. You know how to use R as a calculator.
4. You can create, overwrite and manipulate R objects.
5. You understand the basic rules for naming R objects.
6. You understand the syntax for calling R functions.
7. You know how to nest multiple functions.
8. You can use install and load add-on R packages and call functions from these packages.

Introduction

In the last lesson, you learned how to use RStudio, the wonderful integrated development environment (IDE) that makes working with R much easier. In this lesson, you will learn the basics of using R itself.

To get started, open RStudio, and open a new script with `File > New File > R Script` on the RStudio menu.



Next, **save the script** with `File > Save` on the RStudio menu or by using the shortcut `Command`/`Control` + `S` . This should bring up the Save File dialog box. Save the file with a name like "coding_basics".

You should now type all the code from this lesson into that script.

Comments

There are two main types of text in an R script: commands and comments. A command is a line or lines of R code that instructs R to do something (e.g. `2 + 2`)

A comment is text that is ignored by the computer.

Anything that follows a `#` symbol (pronounced "hash" or "pound") on a given line is a comment. Try typing out and running the code below to see this:

```
# A comment
2 + 2 # Another comment
# 2 + 2
```

Since they are ignored by the computer, comments are meant for *humans.* They help you and others keep track of what your code is doing. Use them often! Like your mother always says, "too much everything is bad, except for R comments".

Question 1

True or False: both code chunks below are valid ways to comment code:?

```
# add two numbers
2 + 2
```

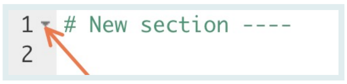
```
2 + 2 # add two numbers
```

Note: All question answers can be found at the end of the lesson.

A fantastic use of comments is to separate your scripts into sections. If you put four dashes after a comment, RStudio will create a new section in your code:

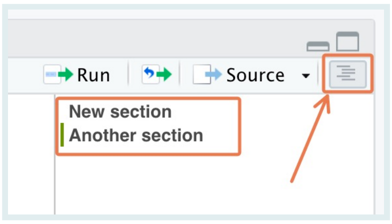
```
# New section ----
```

This has two nice benefits. Firstly, you can click on the little arrow beside the section header to fold, or collapse, that section of code:



```
# New section ----
```

Second, you can click on the "Outline" icon at the top right of the Editor to view and navigate through all the contents in your script:



R s a calculator

R works as a calculator, and obeys the correct order of operations. Type and run the following expressions and observe their output:

```
2 + 2
```

```
## [1] 4
```

```
2 - 2
```

```
## [1] 0
```

```
2 * 2 # two times two
```

```
## [1] 4
```

```
2 / 2 # two divided by two
```

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

```
2 ^ 2 # two raised to the power of two
```

```
## [1] 4
```

```
2 + 2 * 2   # this is evaluated following the order of operations
```

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

```
sqrt(100)   # square root
```

```
## [1] 10
```

The square root command shown on the last line is a good example of an R *function*, where `100` is the *argument* to the function. You will see more functions soon.

REMINDER

We hope you remember the shortcut to run code!

To **run a single line of code**, place your cursor anywhere on that line, then hit `Command` + `Enter` on macOS, or `Control` + `Enter` on Windows.

To **run multiple lines**, drag your cursor to highlight the relevant lines then again press `Command`/`Control` + `Enter`.

Question 2

In the following expression, which sign is evaluated first by R, the minus or the division?

```
2 - 2 / 2
```

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

Formatting code

R does not care how you choose to space out your code.

For the math operations we did above, all the following would be valid code:

```
2+2
```

```
## [1] 4
```

```
2 + 2
```

```
## [1] 4
```

```
2                +                   2
```

```
## [1] 4
```

Similarly, for the `sqrt()` function used above, any of these would be valid:

```
sqrt(100)
```

```
## [1] 10
```

```
sqrt(    100     )
```

```
## [1] 10
```

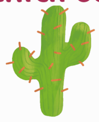
```
# you can even space the command out over multiple lines
sqrt(
  100
    )
```

```
## [1] 10
```

But of course, you should try to space out your code in sensible ways. What exactly is "sensible"? Well, it may be hard for you to know at the moment. Over time, as you read other people's code, you will learn that there are certain R *conventions* for code spacing and formatting.

In the meantime, you can ask RStudio to help format your code for you. To do this, highlight any section of code you want to reformat, and, on the RStudio menu, go to `Code > Reformat Code`, or use the shortcut `Shift` + `Command/Control` + `A`.

WATCH OUT



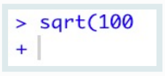
Stuck on the + sign

If you run an incomplete line of code, R will print a `+` sign to indicate that it is waiting for you to finish the code.

For example, if you run the following code:

```
sqrt(100
```

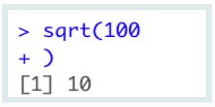
you will not get the output you expect (10). Rather the console will `sqrt(` and a `+` sign:



```
> sqrt(100
+ 
```

R is waiting for you complete the closing parenthesis. You can complete the code and get rid of the `+` by just entering the missing parenthesis:

```
)
```



```
> sqrt(100
+ )
[1] 10
```

Alternatively, press the escape key, `ESC` while your cursor is in the console to start over.

Objects in R

Create an object

When you run code as we have been doing above, the result of the command (or its *value*) is simply displayed in the console—it is not stored anywhere.

```
2 + 2 # R prints this result, 4, but does not store it
```

```
## [1] 4
```

To store a value for future use, assign it to an *object* with the *assignment operator*, <- :

```
my_obj <- 2 + 2 # assign the result of `2 + 2 ` to the object called `my_obj`
my_obj # print my_obj
```

```
## [1] 4
```

The assignment operator, <- , is made of the 'less than' sign, < , and a minus, -. You will use it thousands of times over your R lifetime, so please don't type it manually! Instead, use RStudio's shortcut, `alt` + - (**alt** AND **minus**) on Windows or `option` + - (**option** AND **minus**) on macOS.

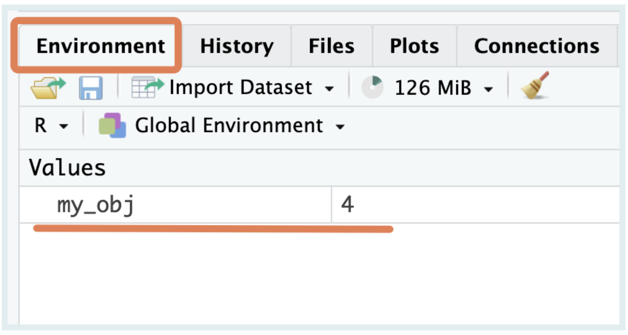
SIDE NOTE

Also note that you can use the *equals* sign, =, for assignment.

```
my_obj = 2 + 2
```

But this is not commonly used by the R community (mostly for historical reasons), so we discourage it too. Follow the convention and use <-.

Now that you've created the object `my_obj`, R knows all about it and will keep track of it during this R session. You can view any created objects in the *Environment* tab of RStudio.



Values

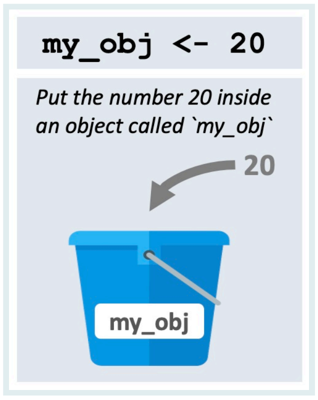
```
my_obj    4
```

What is an object?

So what exactly is an object? Think of it as a named bucket that can contain anything. When you run the code below:

```
my_obj <- 20
```

you are telling R, "put the number 20 inside a bucket named 'my_obj'".

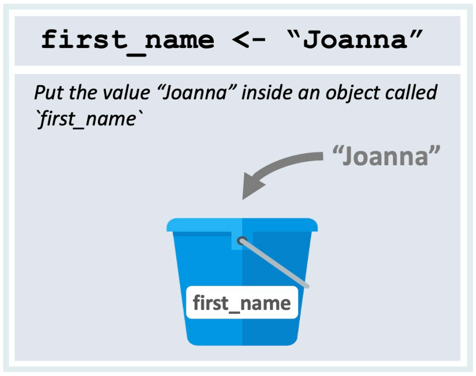


Once the code is run, we would say, in R terms, that "the value of object called `my_obj` is 20".

And if you run this code:

```
first_name <- "Joanna"
```

you are instructing R to "put the value 'Joanna' inside the bucket called 'first_name'".



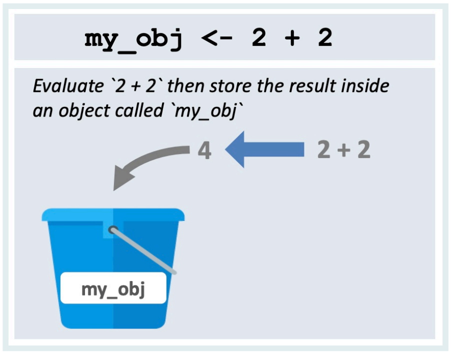
Once the code is run, we would say, in R terms, that "the value of the `first_name` object is Joanna".

Note that R evaluates the code *before* putting it inside the bucket.

So, before when we ran this code,

```
my_obj <- 2 + 2
```

R firsts does the calculation of `2 + 2`, then stores the result, 4, inside the object.


my_obj <- 2 + 2
Evaluate `2 + 2` then store the result inside an object called `my_obj`
4
2 + 2
my_obj

Question 3

Consider the code chunk below:

```
result <-  2 + 2 + 2
```

What is the value of the `result` object created?

A. `2 + 2 + 2`

B. numeric

C. 6

Datasets are objects too

So far, you have been working with very simple objects. You may be thinking "Where are the spreadsheets and datasets? Why are we writing `my_obj <- 2 + 2`? Is this a primary school maths class?!"

Be patient.

We want you to get familiar with the concept of an R object because once you start dealing with real datasets, these will also be stored as R objects.

Let's see a preview of this now. Type out the code below to download a dataset on Ebola cases that we stored on Google Drive and put it in the object `ebola_sierra_leone_data`.

```
ebola_sierra_leone_data <- read.csv("https://tinyurl.com/ebola-data-sample")
ebola_sierra_leone_data # print ebola_data
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
## 4 187  NA   F confirmed    2014-06-19     2014-06-24 Kailahun
## 5  85  20   M confirmed    2014-06-08     2014-06-24 Kailahun
```

This data contains a sample of patient information from the 2014-2016 Ebola outbreak in Sierra Leone.

Because you can store datasets as objects, its very easy to work with multiple datasets at the same time.

Below, we import and view another dataset from the web:

```
diabetes_china <- read.csv("https://tinyurl.com/diabetes-china")
```

Because the dataset above is quite large, it may be helpful to look at it in the data viewer:

```
View(diabetes_china)
```

Notice that both datasets now appear in your *Environment* tab.

SIDE NOTE

Rather than reading data from an internet drive as we did above, it is more likely that you will have the data on your computer, and you will want to read it into R from your there. We will cover this in a future lesson.

Later in the course, we will also show you how to store and read data from a web service like Google Drive, which is nice for easy portability.

Rename an object

You sometimes want to rename an object. It is not possible to do this directly.

To rename an object, you make a copy of the object with a new name, and delete the original.

For example, maybe we decide that the name of the `ebola_sierra_leone_data` object is too long. To change it to the shorter "ebola_data" run:

```
ebola_data <- ebola_sierra_leone_data
```

This has copied the contents from the `ebola_sierra_leone_data` *bucket* to a new `ebola_data` *bucket*.

You can now get rid of the old `ebola_sierra_leone_data` bucket with the `rm()` function, which stands for "remove":

```
rm(ebola_sierra_leone_data)
```

Overwrite an object

Overwriting an object is like changing the *contents* of a *bucket*.

For example, previously we ran this code to store the value "Joanna" inside the `first_name` object:

```
first_name <- "Joanna"
```

To change this to a different, simply re-run the line with a different value:

```
first_name <- "Luigi"
```

You can take a look at the Environment tab to observe the change.

Working with objects

Most of your time in R will be spent manipulating R objects. Let's see some quick examples.

You can run simple commands on objects. For example, below we store the value `100` in an object and then take the square root of the object:

```
my_number <- 100
sqrt(my_number)
```

```
## [1] 10
```

R "sees" `my_number` as the number 100, and so is able to evaluate it's square root.

You can also combine existing objects to create new objects. For example, type out the code below to add `my_number` to itself, and store the result in a new object called `my_sum`:

```
my_sum <- my_number + my_number
```

What should be the value of `my_sum`? First take a guess, then check it.

SIDE NOTE To check the value of an object, such as `my_sum`, you can type and run just the code `my_sum` in the Console or the Editor. Alternatively, you can simply highlight the value `my_sum` in the existing code and press `Command/Control` + `Enter`.

But of course, most of your analysis will involve working with *data* objects, such as the `ebola_data` object we created previously.

Let's see a very simple example of how to interact with a data object; we will tackle it properly in the next lesson.

To get a table of the different sex distribution of patients in the `ebola_data` object, we can run the following:

```
table(ebola_data$sex)
```

```
##
##   F   M
## 124  76
```

The dollar sign symbol, `$`, above allowed us subset to a specific column.

Question 4

a. Consider the code below. What is the value of the `answer` object?

```
eight <- 9
answer <- eight - 8
```

b. Use `table()` to make a table with the distribution of patients across districts in the `ebola_data` object.

Some errors with objects

```
first_name <- "Luigi"
last_name <- "Fenway"
```

```
full_name <- first_name + last_name
```

```
Error in first_name + last_name : non-numeric argument to binary operator
```

The error message tells you that these objects are not numbers and therefore cannot be added with `+`. This is a fairly common error type, caused by trying to do inappropriate things to your objects. Be careful about this.

In this particular case, we can use the function `paste()` to put these two objects together:

```
full_name <- paste(first_name, last_name)
full_name
```

```
## [1] "Luigi Fenway"
```

Another error you'll get a lot is `Error: object 'XXX' not found`. For example:

```
my_number <- 48 # define `my_obj`
My_number + 2 # attempt to add 2 to `my_obj`
```

```
Error: object 'My_number' not found
```

Here, R returns an error message because we haven't created (or *defined*) the object `My_obj` yet. (Recall that R is case-sensitive.)

When you first start learning R, dealing with errors can be frustrating. They're often difficult to understand (e.g. what exactly does "*non-numeric argument to binary operator*" mean?).

Try Googling any error messages you get and browsing through the first few results. This will lead you to forums (e.g. stackoverflow.com) where other R learners have complained about the same error. Here you may find explanations of, and solutions to, your problems.

Question 5

a. The code below returns an error. Why?

```
my_first_name <- "Kene"
my_last_name <- "Nwosu"
my_first_name + my_last_name
```

b. The code below returns an error. Why? (Look carefully)

```
my_1st_name <- "Kene"
my_last_name <- "Nwosu"

paste(my_Ist_name, my_last_name)
```

Naming objects

There are only ***two hard things*** in Computer Science: cache invalidation and ***naming things***.

– Phil Karlton.

Because much of your work in R involves interacting with objects you have created, picking intelligent names for these objects is important.

Naming objects is difficult because names should be both **short** (so that you can type them quickly) and **informative** (so that you can easily remember what is inside the object), and these two goals are often in conflict.

So names that are too long, like the one below, are bad because they take forever to type.

```
sample_of_the_ebola_outbreak_dataset_from_sierra_leone_in_2014
```

And a name like `data` is bad because it is not informative; the name does not give a good idea of what the object is.

As you write more R code, you will learn how to write short and informative names.

For names with multiple words, there are a few conventions for how to separate the words:

```
snake_case <- "Snake case uses underscores"
period.case <- "Period case uses periods"
camelCase <- "Camel case capitalizes new words (but not the first word)"
```

We recommend snake_case, which uses all lower-case words, and separates words with _.

Note too that there are some limitations on objects' names:

- names must start with a letter. So `2014_data` is not a valid name (because it starts with a number).
- names can only contain letters, numbers, periods (`.`) and underscores (_). So `ebola-data` or `ebola~data` or `ebola data` with a space are not valid names.

If you really want to use these characters in your object names, you can enclose the names in backticks:

```
`ebola-data`
`ebola~data`
`ebola data`
```

All of the above are valid R object names. For example, type and run the following code:

```
`ebola~data` <- ebola_data
`ebola~data`
```

But in general you should avoid using backticks to rescue bad object names. Just write proper names.

Question 6

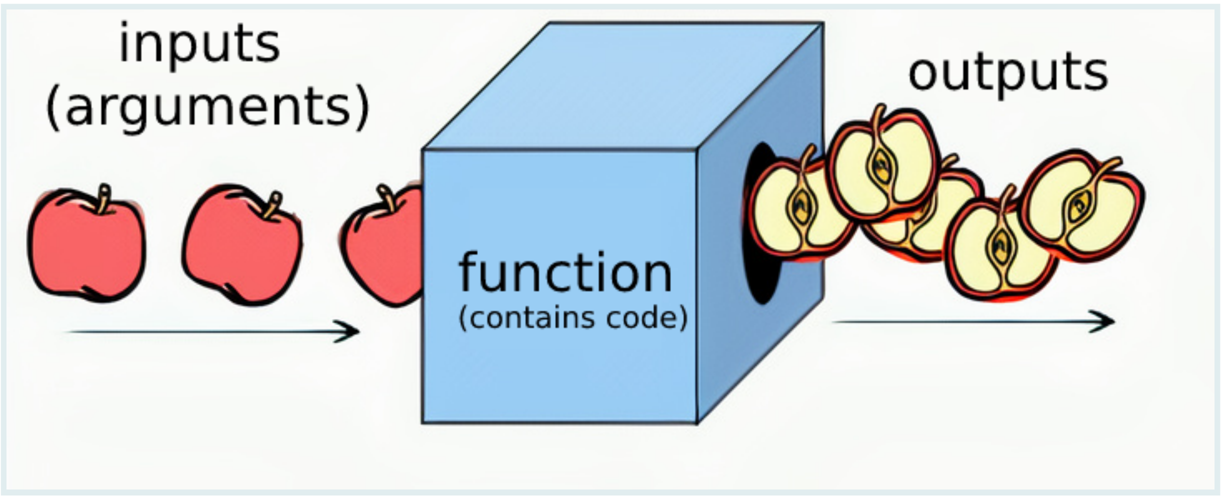
In the code chunk below, we are attempting to take the top 20 rows of the `ebola_data` table. All but one of these lines has an error. Which line will run properly?

```
20_top_rows <- head(ebola_data, 20)
twenty-top-rows <- head(ebola_data, 20)
top_20_rows <- head(ebola_data, 20)
```

Functions

Much of your work in R will involve calling *functions*.

You can think of each function as a machine that takes in some input (or *arguments*) and returns some output.



So far you have already seen many functions, including, `sqrt()`, `paste()` and `plot()`. Run the lines below to refresh your memory:

```
sqrt(100)
paste("I am number", 2 + 2)
plot(women)
```

Basic function syntax

The standard way to call a function is to provide a *value* for each *argument*:

```
function_name(argument1 = "value", argument2 = "value")
```

Let's demonstrate this with the `head()` function, which returns the first few elements of an object.

To return the first three rows of the Ebola dataset, you run:

```
head(x = ebola_data, n = 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

In the code above, `head()` takes in two arguments:

- `x` , the object of interest, and
- `n`, the number of elements to return.

We can also swap the order of the arguments:

```
head(n = 3, x = ebola_data)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

If you put the argument values in the right order, you can skip typing their names. So the following two lines of code are equivalent and both run:

```
head(x = ebola_data, n = 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

```
head(ebola_data, 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

But if the argument values are in the wrong order, you will get an error if you do not type the argument names. Below, the first line runs but the second does not run:

```
head(n = 3, x = ebola_data)
head(3, ebola_data)
```

(To see the "correct order" for the arguments, take a look at the help file for the `head()` function)

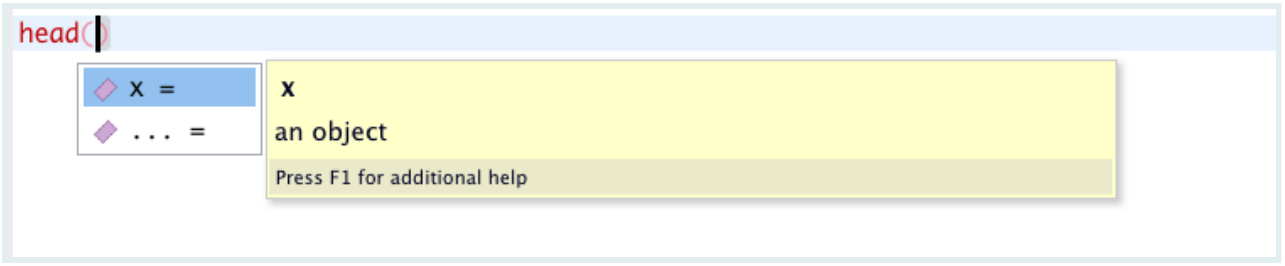
Some function arguments can be skipped altogether, because they have *default* values.

For example, with `head()`, the default value of `n` is 6, so running just `head(ebola_data)` will return the first 6 rows.

```
head(ebola_data)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
## 4 187  NA   F confirmed    2014-06-19     2014-06-24 Kailahun
## 5  85  20   M confirmed    2014-06-08     2014-06-24 Kailahun
## 6 277  30   F confirmed    2014-06-29     2014-07-01   Kenema
```

To see the arguments to a function, press the **Tab** key when your cursor is inside the function's parentheses:



```
head()
```

Question 7

In the code lines below, we are attempting to take the top 6 rows of the `women` dataset (which is built into R). Which line is invalid?

```
head(women)
head(women, 6)
head(x = women, 6)
head(x = women, n = 6)
head(6, women)
```

(If you are not sure, just try typing and running each line. Remember that the goal here is for you to gain some practice.)

Let's spend some time playing with another function, the `paste()` function, which we already saw above, This function is a bit special because it can take in any number of input arguments.

So you could have two arguments:

```
paste("Luigi", "Fenway")
```

```
## [1] "Luigi Fenway"
```

Or four arguments:

```
paste("Luigi", "Fenway", "Luigi", "Fenway")
```

```
## [1] "Luigi Fenway Luigi Fenway"
```

And so on up to infinity.

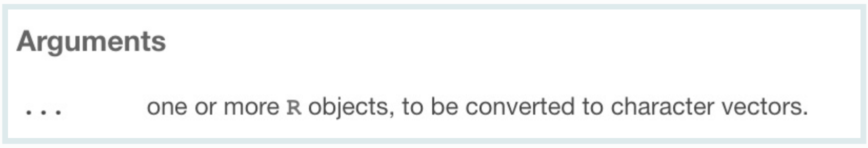
And as you might recall, we can also `paste()` named objects:

```
first_name <- "Luigi"
paste("My name is", first_name, "and my last name is", last_name)
```

```
## [1] "My name is Luigi and my last name is Fenway"
```

PRO TIP

Functions like `paste()` can take in many values because they have a special argument, an ellipsis: … If you consult the help file for the paste function, you will see this:



Arguments

`...` one or more `R` objects, to be converted to character vectors.

Another useful argument for `paste()` is called `sep`. It tells R what character to use to separate the terms:

```
paste("Luigi", "Fenway", sep = "-")
```

```
## [1] "Luigi-Fenway"
```

Nesting functions

The output of a function can be immediately taken in by another function. This is called function nesting.

For example, the function `tolower()` converts a string to lower case.

```
tolower("LUIGI")
```

```
## [1] "luigi"
```

You can take the output of this and pass it directly into another function:

```
paste(tolower("LUIGI"), "is my name")
```

```
## [1] "luigi is my name"
```

Without this option of nesting, you would have to assign an intermediate object:

```
my_lowercase_name <- tolower("LUIGI")
paste(my_lowercase_name, "is my name")
```

```
## [1] "luigi is my name"
```

Function nesting will come in very handy soon.

Question 8

The code chunks below are all examples of function nesting. One of the lines has an error. Which line is it, and what is the error?

```
sqrt(head(women))
```

```
paste(sqrt(9), "plus 1 is", sqrt(16))
```

```
sqrt(tolower("LUIGI"))
```

Packages

As we mentioned previously, R is wonderful because it is user extensible: anyone can create a software *package* that adds new functionality. Most of R's power comes from these packages.

In the previous lesson, you installed and loaded the {highcharter} package using the `install.packages()` and `library()` functions. Let's learn a bit more about packages now.

A first example: the {tableone} package

Let's now install and use another R package, called `tableone`:

```
install.packages("tableone")
```

```
library(tableone)
```

Note that you only need to install a package once, but you have to load it with `library()` each time you want to use it. This means that you should generally run the `install.packages()` line directly from the console, rather than typing it into your script.

The package eases the construction of "Table 1", i.e. a table with characteristics of the study sample that is commonly found in biomedical research papers.

The simplest use case is summarizing the whole dataset. You can just feed in the data frame to the `data` argument of the main workhorse function `CreateTableOne()`.

```
CreateTableOne(data = ebola_data)
```

```
##
##                           Overall
##   n                         200
##   id (mean (SD))          146.00 (82.28)
##   age (mean (SD))          33.12 (17.85)
##   sex = M (%)                 76 (38.0)
##   status = suspected (%)      18 ( 9.0)
##   date_of_onset (%)
##      2014-05-18                1 ( 0.5)
##      2014-05-20                1 ( 0.5)
##      2014-05-21                1 ( 0.5)
##      2014-05-22                2 ( 1.0)
##      2014-05-23                1 ( 0.5)
##      2014-05-24                2 ( 1.0)
##      2014-05-26                8 ( 4.0)
##      2014-05-27                7 ( 3.5)
##      2014-05-28                1 ( 0.5)
##      2014-05-29                9 ( 4.5)
##      2014-05-30                4 ( 2.0)
##      2014-05-31                2 ( 1.0)
##      2014-06-01                2 ( 1.0)
##      2014-06-02                1 ( 0.5)
##      2014-06-03                1 ( 0.5)
##      2014-06-05                1 ( 0.5)
##      2014-06-06                5 ( 2.5)
##      2014-06-07                3 ( 1.5)
##      2014-06-08                4 ( 2.0)
##      2014-06-09                1 ( 0.5)
##      2014-06-10               22 (11.0)
##      2014-06-11                1 ( 0.5)
##      2014-06-12                7 ( 3.5)
##      2014-06-13               15 ( 7.5)
##      2014-06-14                8 ( 4.0)
##      2014-06-15                3 ( 1.5)
##      2014-06-16                1 ( 0.5)
##      2014-06-17                4 ( 2.0)
##      2014-06-18                5 ( 2.5)
##      2014-06-19                8 ( 4.0)
##      2014-06-20                7 ( 3.5)
##      2014-06-21                2 ( 1.0)
##      2014-06-22                1 ( 0.5)
##      2014-06-23                2 ( 1.0)
##      2014-06-24                8 ( 4.0)
##      2014-06-25                6 ( 3.0)
##      2014-06-26               10 ( 5.0)
##      2014-06-27                9 ( 4.5)
##      2014-06-28               17 ( 8.5)
##      2014-06-29                7 ( 3.5)
##    date_of_sample (%)
##      2014-05-23                1 ( 0.5)
##      2014-05-25                1 ( 0.5)
##      2014-05-26                1 ( 0.5)
##      2014-05-27                2 ( 1.0)
##      2014-05-28                1 ( 0.5)
##      2014-05-29                2 ( 1.0)
##      2014-05-31                9 ( 4.5)
##      2014-06-01                6 ( 3.0)
##      2014-06-02                1 ( 0.5)
##      2014-06-03                9 ( 4.5)
##      2014-06-04                4 ( 2.0)
##      2014-06-05                1 ( 0.5)
##      2014-06-06                2 ( 1.0)
##      2014-06-07                2 ( 1.0)
##      2014-06-10                2 ( 1.0)
##      2014-06-11                4 ( 2.0)
##      2014-06-12                3 ( 1.5)
##      2014-06-13                3 ( 1.5)
##      2014-06-14                1 ( 0.5)
##      2014-06-15               21 (10.5)
##      2014-06-16                1 ( 0.5)
##      2014-06-17                5 ( 2.5)
##      2014-06-18               13 ( 6.5)
##      2014-06-19                9 ( 4.5)
##      2014-06-21                8 ( 4.0)
##      2014-06-22                7 ( 3.5)
##      2014-06-23                6 ( 3.0)
##      2014-06-24                6 ( 3.0)
##      2014-06-25                3 ( 1.5)
##      2014-06-27                5 ( 2.5)
##      2014-06-28                2 ( 1.0)
##       2014-06-29                  8 ( 4.0)
##       2014-06-30                  6 ( 3.0)
##       2014-07-01                  4 ( 2.0)
##       2014-07-02                 16 ( 8.0)
##       2014-07-03                 13 ( 6.5)
##       2014-07-04                  2 ( 1.0)
##       2014-07-05                  2 ( 1.0)
##       2014-07-06                  1 ( 0.5)
##       2014-07-08                  3 ( 1.5)
##       2014-07-12                  1 ( 0.5)
##       2014-07-14                  1 ( 0.5)
##       2014-07-17                  1 ( 0.5)
##       2014-07-21                  1 ( 0.5)
##    district (%)
##       Bo                          4 ( 2.0)
##       Kailahun                  146 (73.0)
##       Kenema                     41 (20.5)
##       Kono                        2 ( 1.0)
##       Port Loko                   2 ( 1.0)
##       Western Urban               5 ( 2.5)
```

You can see there are 200 patients in this dataset, the mean age is 33 and 38% of the sample of the sample is male, among other details.

Very cool! (One problem is that the package is assuming that the date variables are categorical; because of this the output table is much too long!)

The point of this demonstration of {tableone} is to show you that there is a lot of power in external R packages. This is a big strength of working with R, an open-source language with a vibrant ecosystem of contributors. Thousands of people are working right now on packages that may be helpful to you one day.

You can Google search "Cool R packages" and browse through the answers if you are eager to learn about more R packages.

SIDE NOTE

You may have noticed that we embrace package names in curly braces, e.g. {tableone}. This is just a styling convention among R users/teachers. The braces do not *mean* anything.

Full signifiers

The *full signifier* of a function includes both the package name and the function name: `package::function()`.

So for example, instead of writing:

```
CreateTableOne(data = ebola_data)
```

We could write this function with its full signifier, `package::function()`:

```r
tableone::CreateTableOne(data = ebola_data)
```

You usually do not need to use these full signifiers in your scripts. But there are some situations where it is helpful:

The most common reason is that you want to make it very clear which package a function comes from.

Secondly, you sometimes want to avoid needing to run `library(package)` before accessing the functions in a package. That is, you want to use a function from a package without first loading that package from the library. In that case, you can use the full signifier syntax.

So the following:

```r
tableone::CreateTableOne(data = ebola_data)
```

is equivalent to:

```r
library(tableone)
CreateTableOne(data = ebola_data)
```

Question 9

PRACTICE (in RMD)

Consider the code below:

```r
tableone::CreateTableOne(data = ebola_data)
```

Which of the following is a correct interpretation of what this code means:

A. The code applies the `CreateTableOne` function from the {tableone} package on the `ebola_data` object.

B. The code applies the `CreateTableOne` argument from the {tableone} function on the `ebola_data` package.

C. The code applies the `CreateTableOne` function from the {tableone} package on the `ebola_data` package.

pacman::p_load()

Rather than use two separate functions, `install.packages()` then `library()`, to install then load packages, you can use a single function, `p_load()`, from the {pacman} package to automatically install a package if it is not yet installed, *and* load the package. We encourage this approach in the rest of this course.

Install {pacman} now by running this in your console:

```
install.packages("pacman")
```

From now on, when you are introduced to a new package, you can simply use, `pacman::p_load(package_name)` to both install and load the package:

Try this now for the `outbreaks` package, which we will use soon:

```
pacman::p_load(outbreaks)
```

Now we have a small problem. The wonderful function `pacman::p_load()` automatically installs and loads packages.

But it would be nice to have some code that automatically installs the {pacman} package itself, if it is missing on a user's computer.

But if you put the `install.packages()` line in a script, like so:

```
install.packages("pacman")
pacman::p_load(here, rmarkdown)
```

you will waste a lot of time. Because every time a user opens and runs a script, it will *reinstall* {pacman}, which can take a while. Instead we need code that first *checks whether pacman is not yet installed* and installs it if this is not the case.

We can do this with the following code:

```
if(!require(pacman)) install.packages("pacman")
```

You do not have to understand it at the moment, as it uses some syntax that you have not yet learned. Just note that in future chapters, we will often start a script with code like this:

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(here, rmarkdown)
```

The first line will install {pacman} if it is not yet installed. The second line will use `p_load()` function from {pacman} to load the remaining packages (and `pacman::p_load()` installs any packages that are not yet installed).

Phew! Hope your head is still intact.

Question 10

At the start of an R script, we would like to install and load the package called {janitor}. Which of the following code chunks do we recommend you have in your script?

A.

```r
if(!require(pacman)) install.packages("pacman")
pacman::p_load(janitor)
```

B.

```r
install.packages("janitor")
library(janitor)
```

C.

```r
install.packages("janitor")
pacman::p_load(janitor)
```

Wrapping up

With your new knowledge of R objects, R functions and the packages that functions come from, you are ready, believe it or not, to do basic data analysis in R. We'll jump into this head first in the next lesson. See you there!

Answers

1. True.
2. The division sign is evaluated first.
3. The answer is C. The code `2 + 2 + 2` gets evaluated before it is stored in the object.
4. a. The value is 1. The code evaluates to `9-8`.

   b. table(ebola_data$district)
5. a. You cannot add two character strings. Adding only works for numbers.

   b. `my_1st_name` is typed with the number 1 initially, but in the `paste()` command, it is typed with the letter "l".

6. The third line is the only line with a valid object name: `top_20_rows`

7. The last line, `head(6, women)`, is invalid because the arguments are in the wrong order and they are not named.

8. The third code chunk has a problem. It attempts to find the square root of a character, which is impossible.

9. The first line, A, is the correct interpretation.

10. The first code chunk is the recommended way to install and load the package {janitor}

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAMECK AGASA

Statistician/Data Scientist

OLIVIA KEISER

Head of division of Infectious Diseases and Mathematical Modelling, University of Geneva

References

Some material in this lesson was adapted from the following sources:

- "File:Apple slicing function.png." *Wikimedia Commons, the free media repository*. 1 Oct 2021, 04:26 UTC. 20 Mar 2022, 17:27 <https://commons.wikimedia.org/w/index.php?title=File:Apple_slicing_function.png&oldid=594767630>.

- "PsyteachR | Data Skills for Reproducible Research." 2021. Github.io. 2021. https://psyteachr.github.io/reprores-v2/index.html.

- Douglas, Alex, Deon Roos, Francesca Mancini, Ana Couto, and David Lusseau. 2022. "An Introduction to R." Intro2r.com. January 27, 2022. https://intro2r.com/.

Lesson notes | Data dive: Ebola in Sierra Leone

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Learning objectives

1. You can use RStudio's graphic user interface to import CSV data into R.
2. You can explain the concept of reproducibility.
3. You can use the `nrow()`, `ncol()` and `dim()` functions to get the dimensions of a dataset, and the `summary()` function to get a summary of the dataset's variables.
4. You can use `vis_dat()`, `inspect_num()` and `inspect_cat()` to obtain visual summaries of a dataset.
5. You can inspect a numeric variable:
   - with the summary functions `mean()` , `median()`, `max()`, `min()`, `length()` and `sum()`;
   - with esquisse-generated ggplot2 code.
6. You can inspect a categorical variable:
   - with the summary functions `table()` and `janitor::tabyl()`;
   - with the graphical functions `barplot()` and `pie()`.

Introduction

With your newly-acquired knowledge of functions and objects, you now have the basic building blocks required to do simple data analysis in R. So let's get started. The goal is to start working with data as quickly as possible, even before you feel ready.

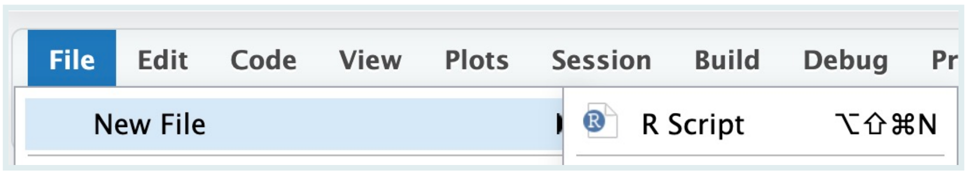
Here you will analyze a dataset of confirmed and suspected cases of Ebola hemorrhagic fever in Sierra Leone in May and June of 2014 (Fang et al., 2016). The data is shown below:

You will import and explore this dataset, then use R to answer the following questions about the outbreak:

- **When was the first case reported?**
- **What was the median age of those affected?**
- **Had there been more cases in men or women?**
- **What district had had the most reported cases?**
- **By the end of June 2014, was the outbreak growing or receding?**

Script setup

First, open a new script in RStudio with `File > New File > R Script`. (If you are on RStudio, you can open up any of your previously-created projects.)



Next, save the script with `File > Save As` or press `Command`/`Control` + `S` to bring up the Save File dialog box. Save the file with the name "ebola_analysis" or something similar

SIDE NOTE

Empty your environment at the start of the analysis

When you start a new analysis, your R environment should usually be empty. Verify this by opening the *Environment* tab; it should say "Environment is empty". If instead, it shows some previously-loaded objects, it is recommended to restart R by going to the menu option `Session > Restart R`

Header

Add a title, name and date to the start of the script, as code comments. This is generally good practice for writing R scripts, as it helps give you and your collaborators context about your script. Your header may look like this:

```
# Ebola Sierra Leone analysis
# John Sample-Name Doe
# 2024-01-01
```

Packages

Next, use the `p_load()` function from {pacman} to load the packages you will be using. Put this under a section header called "Load packages", with four hyphens, as shown below:

```r
# Load packages ----
if(!require(pacman)) install.packages("pacman")
pacman::p_load(
  tidyverse, # meta-package
  inspectdf,
  plotly,
  janitor,
  visdat,
  esquisse
)
```

REMINDER

Remember that the *full signifier* of a function includes both the package name and the function name, `package::function()`. This full signifier is handy if you want to use a function before you have loaded its source package. This is the case in the code chunk above: we want use `p_load()` from {pacman} without formally loading the {pacman} package, so we type `pacman::p_load()`

We could also first load {pacman} before using the p_load function:

```r
library(pacman) # first load {pacman}
p_load(tidyverse) # use `p_load` from {pacman} to load other
        packages
```

(Also recall that the benefit of `p_load()` is that it automatically installs a package if it is not yet installed. Without `p_load()`, you have to first install the package with `install.packages()` before you can load it with `library()`.)

Importing data into R

Now that the needed packages are loaded, you should import the dataset.

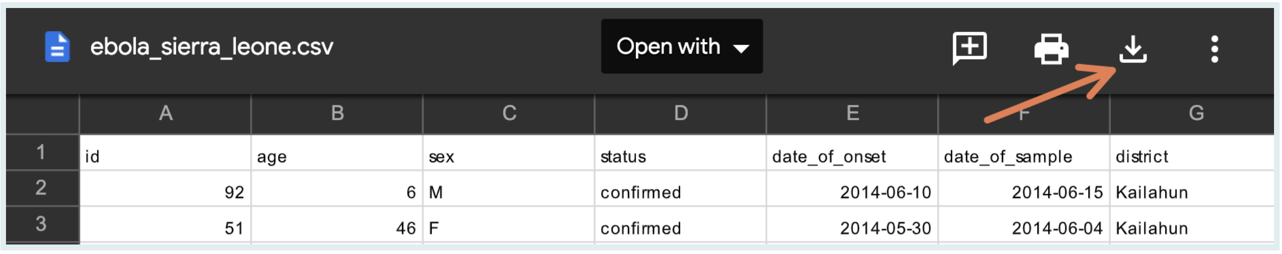
SIDE NOTE

About the Ebola dataset

The data you will be working on contains a sample of patient information from the 2014-2016 Ebola outbreak in Sierra Leone. It comes from a research paper which analyzed the transmission dynamics of that outbreak. Key variables include the `status` of a case, whether the case

SIDE NOTE sample was taken. To learn more about these data, visit the source publication here: bit.ly/ebola-data-source. Or search the following DOI on DOI.org: 10.1073/pnas.1518587113.

Go to bit.ly/view-ebola-data to view the dataset you will be working on. Then click the download icon at the top to download it to your computer.

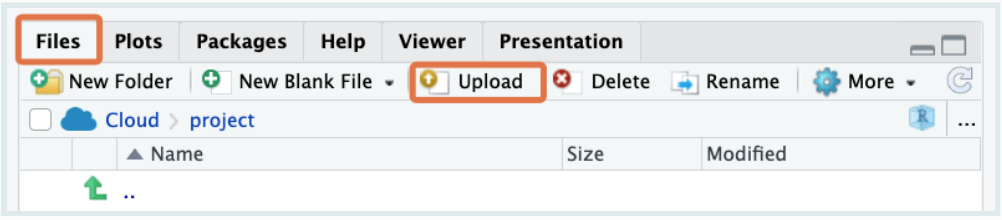


	A	B	C	D	E	F	G
1	id	age	sex	status	date_of_onset	date_of_sample	district
2	92	6	M	confirmed	2014-06-10	2014-06-15	Kailahun
3	51	46	F	confirmed	2014-05-30	2014-06-04	Kailahun

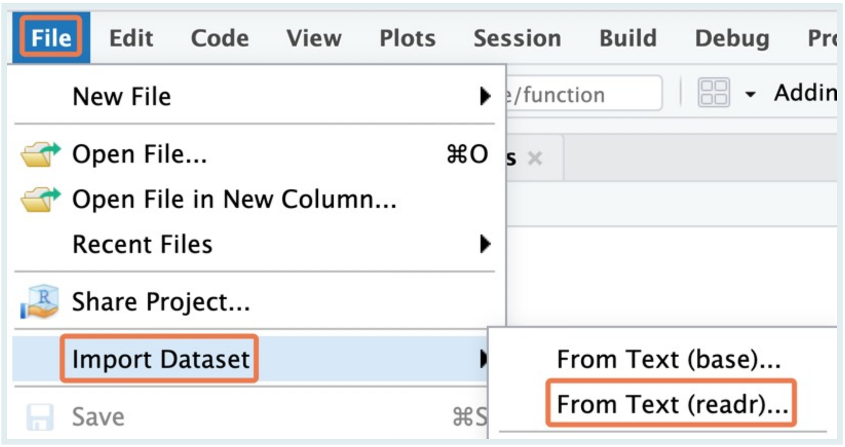
You can leave the dataset in your downloads folder, or move it to somewhere more respectable; the upcoming steps will work independent of where the data is stored. In the next lesson, you will learn how to organize your data analysis projects properly, and we will think about the ideal folder setup for storing data.

RSTUDIO CLOUD

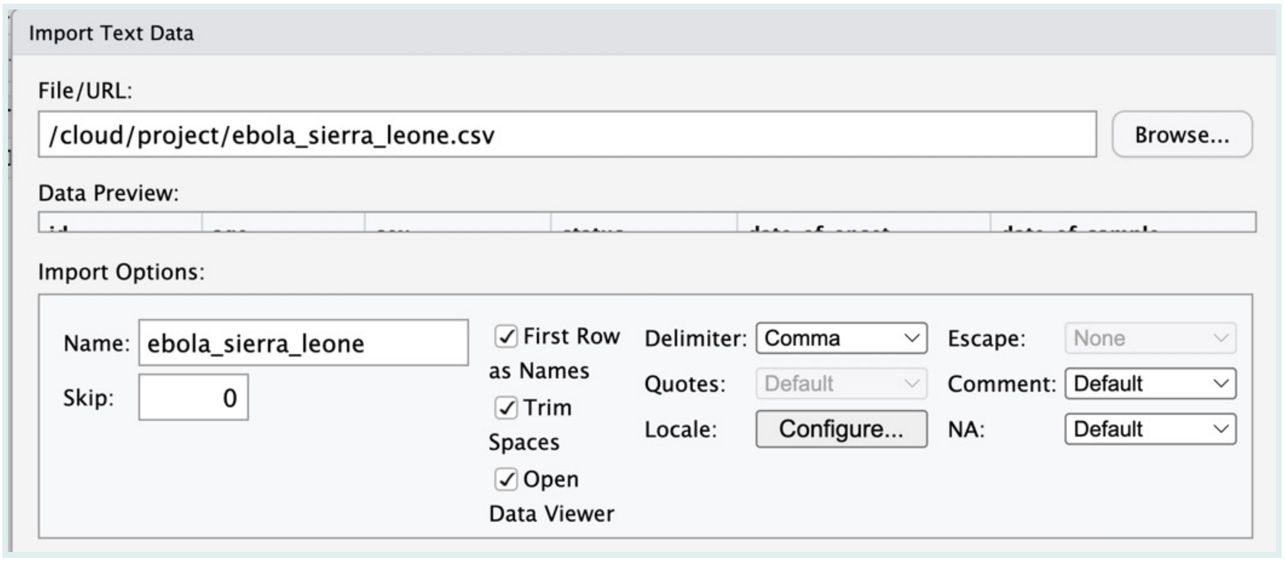
NOTE: If you are using RStudio Cloud, you need to upload your dataset to the cloud. Do this in the "Files" tab by clicking on the "Upload" button.



Next, on the RStudio menu, go to `File > Import Dataset > From Text (readr)`.



Browse through the computer's files and navigate to the downloaded dataset. Click to open it. You should see an import dialog box like this:


Import Text Data

File/URL: /cloud/project/ebola_sierra_leone.csv

Data Preview:

Import Options:

Name: ebola_sierra_leone

Skip: 0

✓ First Row as Names

✓ Trim Spaces

✓ Open Data Viewer

Delimiter: Comma

Quotes: Default

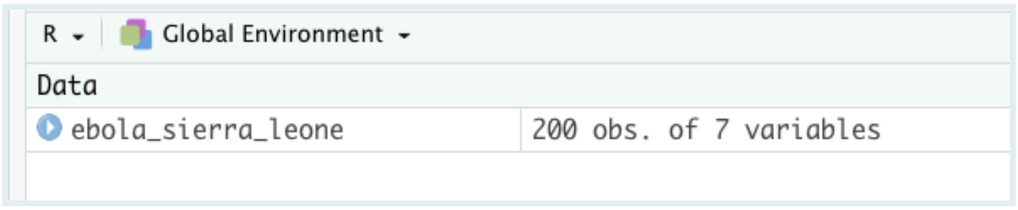
Locale: Configure...

Escape: None

Comment: Default

NA: Default

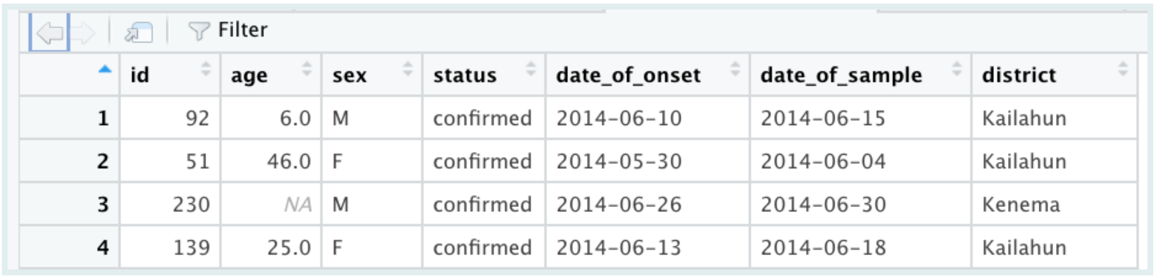
Leave all the import settings at the default values; simply click on "Import" at the bottom; this should load the dataset into R. You can tell this by looking at your environment pane, which should now feature an object called "ebola_sierra_leone" or something similar:



Data

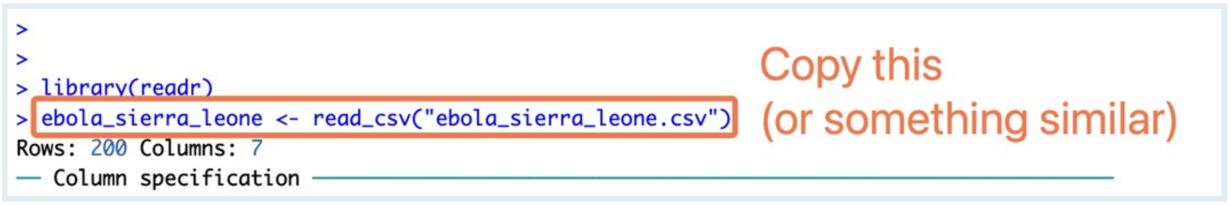
ebola_sierra_leone 200 obs. of 7 variables

RStudio should also have called the `View()` function on your dataset, so you should see a familiar spreadsheet view of this data:



	id	age	sex	status	date_of_onset	date_of_sample	district
1	92	6.0	M	confirmed	2014-06-10	2014-06-15	Kailahun
2	51	46.0	F	confirmed	2014-05-30	2014-06-04	Kailahun
3	230	NA	M	confirmed	2014-06-26	2014-06-30	Kenema
4	139	25.0	F	confirmed	2014-06-13	2014-06-18	Kailahun

Now take a look at your console. Do you observe that your actions in the graphical user interface actually triggered some R code to be run? Copy the line of code that includes the `read_csv()` function, leaving out the > symbol.



```
>
>
> library(readr)
> ebola_sierra_leone <- read_csv("ebola_sierra_leone.csv")
Rows: 200 Columns: 7
── Column specification ──────────────────────────────
```

Paste the copied code into your R script, and label this section "Load data". This may look something like the below (the file path inside quotes will differ from computer to computer.

```
# Load data ----
ebola_sierra_leone <- read_csv("~/Downloads/ebola_sierra_leone.csv")
```

RECAP

Nice work so far!

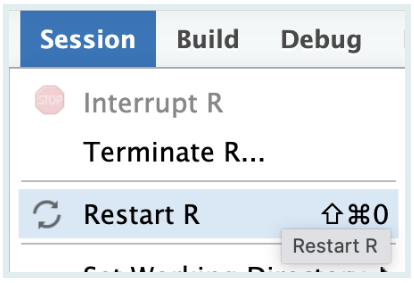
Your R script should look similar to this:

```
# Ebola Sierra Leone analysis
# John Sample-Name Doe
# 2024-01-01

# Load packages ----
if(!require(pacman)) install.packages("pacman")
pacman::p_load(
  tidyverse,
  inspectdf,
  plotly,
  janitor,
  visdat
)

# Load data ----
ebola_sierra_leone <-
        read_csv("~/Downloads/ebola_sierra_leone.csv")
```

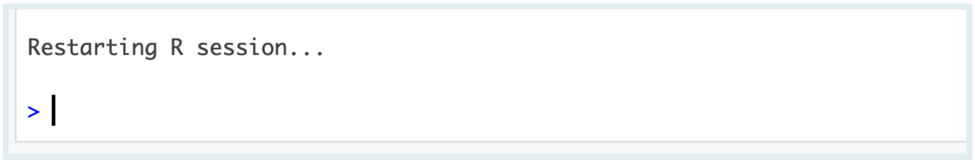
Intro to reproducibility

Now that the code for importing data is in your R script, you can easily rerun this script anytime to reimport the dataset; there will be no need to redo the manual point-and-click procedure for data import.

Try restarting R and rerunning the script now. Save your script with `Control/Command` + `s` , then *restart* R with the RStudio Menu, at `Session > Restart R`. On RStudio Cloud, the menu option looks like this:

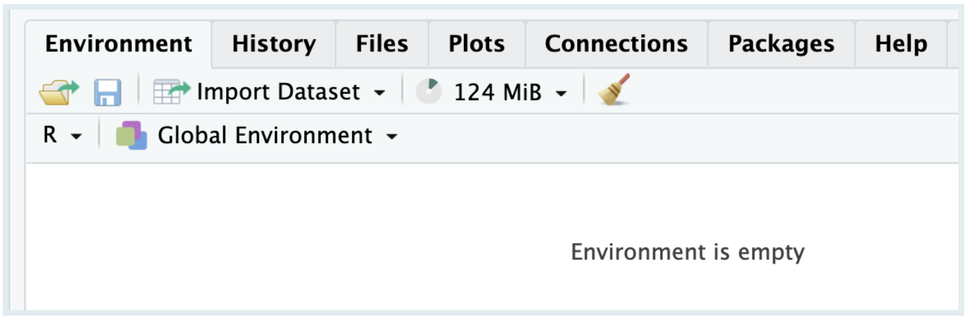


If restarting is successful, your console should print this message:



```
Restarting R session...

> |
```

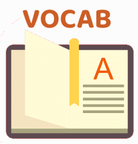
You should also see the phrase "Environment is empty" in the Environment tab, indicating that the dataset you imported is no longer stored by R—you are starting with a fresh workspace.



To re-run your script, use `Command/Control` + `a` to highlight all the code, then `Command/Control` + `Enter` to run it.

If this worked, congratulations; you have the beginnings of your first "reproducible" analysis script!

What does "reproducible" mean?



When you do things with code rather than by pointing and clicking, it is easy for anyone to re-run, or *reproduce* these steps, by simply re-running your script.

While you can use RStudio's graphical user interface to point-and-click your way through the data import process, you should always copy the relevant code to your script so that your script remains a reproducible record of all your analysis steps.

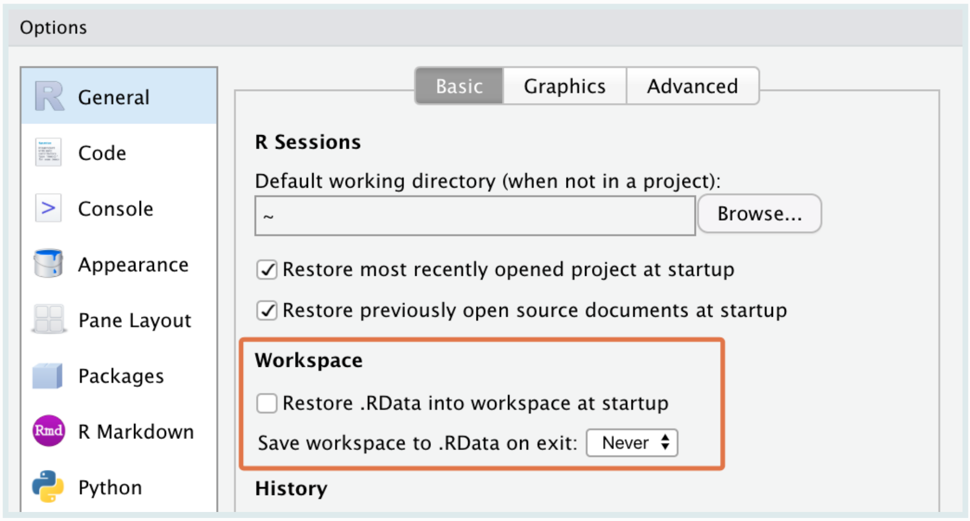
VOCAB

Of course, your script so far is not yet *entirely* reproducible, because the file path for the dataset (the one that looks like this: "...intro-to-data-analysis-with-r/ch01_getting_started/data...") is specific to just your computer. Later on we will see how to use relative file paths, so that the code for importing data can work on anyone's computer.

WATCH OUT

If your environment was not empty after restarting R, it means you skipped a step in a previous lesson. Do this now:

- In the RStudio Menu, go to `Tools > Global Options` to bring up RStudio's options dialog box.
- Then go to `General > Basic`, and **uncheck** the box that says "Restore .RData into workspace at startup".
- For the option, "save your workspace to .RData on exit", set this to "Never".



R Sessions

Default working directory (when not in a project): ~

[✓] Restore most recently opened project at startup

[✓] Restore previously open source documents at startup

Workspace

[ ] Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

Quick data exploration

Now let's walk through some basic steps of data exploration—taking a broad, bird's eye look at the dataset. You should put this section under a heading like "Explore data" in your script.

To view the top and bottom 6 rows of the dataset, you can use the `head()` and `tail()` functions:

```
# Explore data ----
head(ebola_sierra_leone)
```

```
## # A tibble: 6 × 7
##      id   age sex   status    date_of_onset date_of_sample
##   <dbl> <dbl> <chr> <chr>     <date>        <date>
## 1    92     6 M     confirmed 2014-06-10    2014-06-15
## 2    51    46 F     confirmed 2014-05-30    2014-06-04
## 3   230    NA M     confirmed 2014-06-26    2014-06-30
## 4   139    25 F     confirmed 2014-06-13    2014-06-18
## 5     8     8 F     confirmed 2014-05-22    2014-05-27
## 6   215    49 M     confirmed 2014-06-24    2014-06-29
## # … with 1 more variable: district <chr>
```

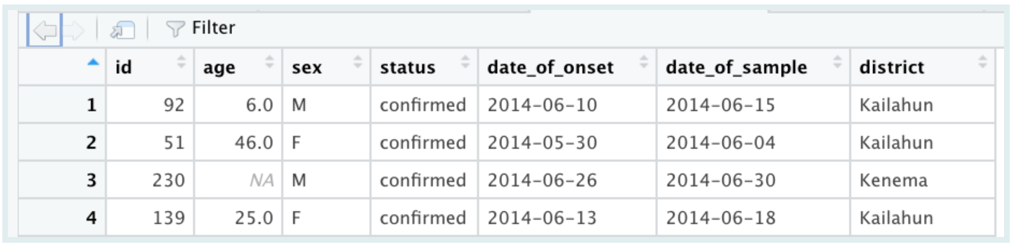
```
tail(ebola_sierra_leone)
```

```
## # A tibble: 6 × 7
##      id   age sex   status    date_of_onset date_of_sample
##   <dbl> <dbl> <chr> <chr>     <date>        <date>
## 1   214     6 F     confirmed 2014-06-24    2014-06-30
## 2    28    45 F     confirmed 2014-05-27    2014-06-01
## 3    12    27 F     confirmed 2014-05-22    2014-05-27
## 4   110     6 M     confirmed 2014-06-10    2014-06-15
## 5   209    40 F     confirmed 2014-06-24    2014-06-27
## 6    35    29 M     suspected 2014-05-28    2014-06-01
## # … with 1 more variable: district <chr>
```

To view the whole dataset, use the `View()` function.

```
View(ebola_sierra_leone)
```

This will again open a familiar spreadsheet view of the data:



	id	age	sex	status	date_of_onset	date_of_sample	district
1	92	6.0	M	confirmed	2014-06-10	2014-06-15	Kailahun
2	51	46.0	F	confirmed	2014-05-30	2014-06-04	Kailahun
3	230	NA	M	confirmed	2014-06-26	2014-06-30	Kenema
4	139	25.0	F	confirmed	2014-06-13	2014-06-18	Kailahun

You can close this tab and return to your script.

The functions `nrow()`, `ncol()` and `dim()` give you the dimensions of your dataset:

```
nrow(ebola_sierra_leone) # number of rows
```

```
## [1] 200
```

```
ncol(ebola_sierra_leone) # number of columns
```

```
## [1] 7
```

```
dim(ebola_sierra_leone) # number of rows and columns
```

```
## [1] 200   7
```

REMINDER

If you're not sure what a function does, remember that you can get function help with the question mark symbol. For example, to get help on the `ncol()` function, run:

```
?ncol
```

Another often-helpful function is `summary()`:

```
summary(ebola_sierra_leone)
```

```
##        id               age             sex                 status           
date_of_onset
##  Min.   :  1.00   Min.   : 1.80   Length:200         Length:200         
Min.   :2014-05-18
##  1st Qu.: 62.75   1st Qu.:20.00   Class :character   Class :character   
1st Qu.:2014-06-01
##  Median :131.50   Median :35.00   Mode  :character   Mode  :character   
Median :2014-06-13
##  Mean   :136.72   Mean   :33.85                                        
Mean   :2014-06-12
##  3rd Qu.:208.25   3rd Qu.:45.00                                        
3rd Qu.:2014-06-23
##  Max.   :285.00   Max.   :80.00                                        
Max.   :2014-06-29
##                   NA's   :4                                            
##  date_of_sample         district        
##  Min.   :2014-05-23   Length:200        
##  1st Qu.:2014-06-07   Class :character  
##  Median :2014-06-18   Mode  :character  
##  Mean    :2014-06-17
##  3rd Qu.:2014-06-29
##  Max.    :2014-07-17
##
```

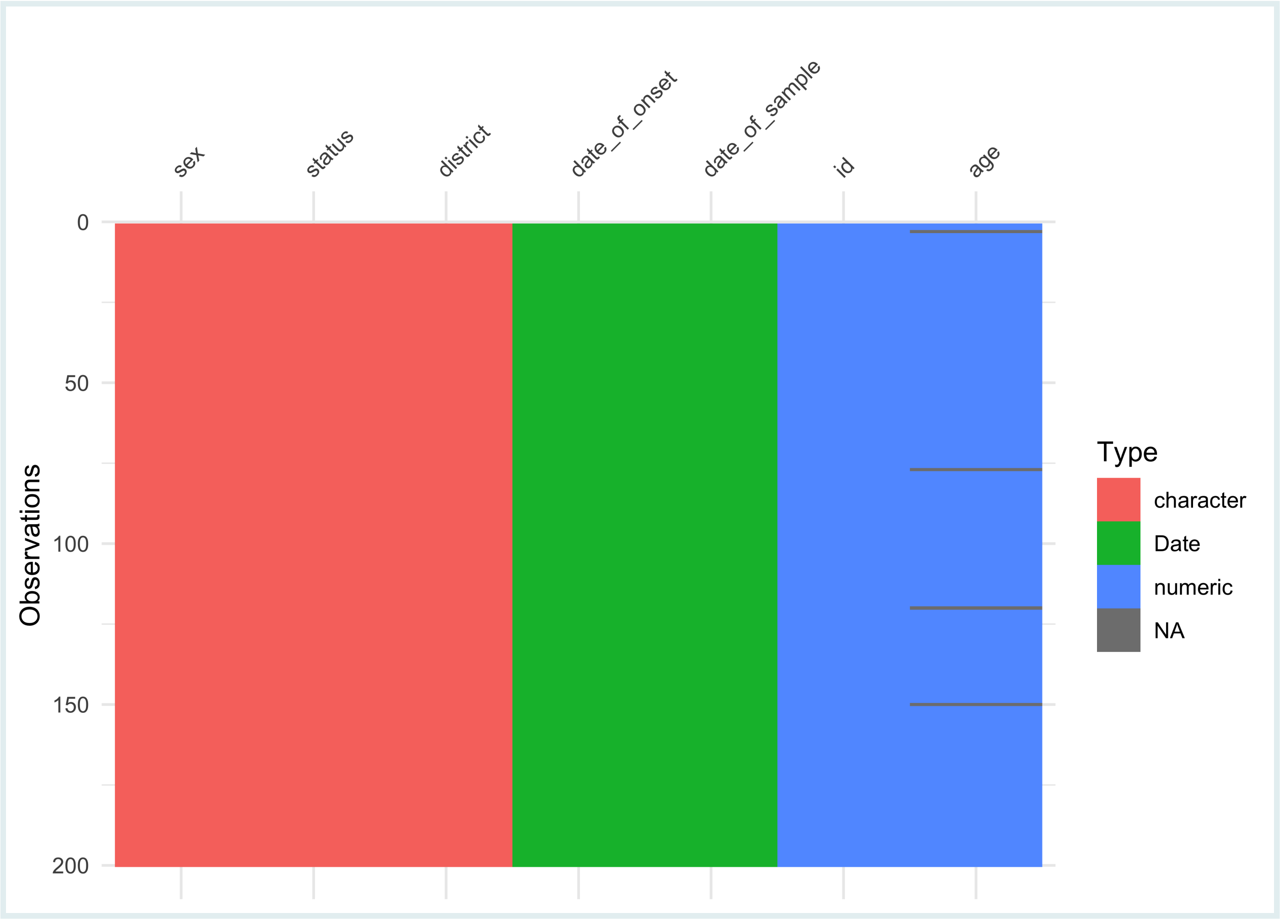
As you can see, for numeric columns in your dataset, `summary()` gives you the minimum value, the maximum value, the mean, median and the 1st and 3rd quartiles.

For character columns it gives you just the length of the column (the number of rows), the "class" and the "mode". We will discuss what "class" and "mode" mean later.

`vis_dat()`

The `vis_dat()` function from the {visdat} package is a wonderful way to quickly visualize the data types and the missing values in a dataset. Try this now:

```
vis_dat(ebola_sierra_leone)
```



From this figure, you can quickly see the character, date and numeric data types, and you can note that age is missing for some cases.

`inspect_cat()` and `inspect_num()`

Next, `inspect_cat()` and `inspect_num()` from the {inspectdf} package give you visual summaries of the distribution of variables in the dataset.

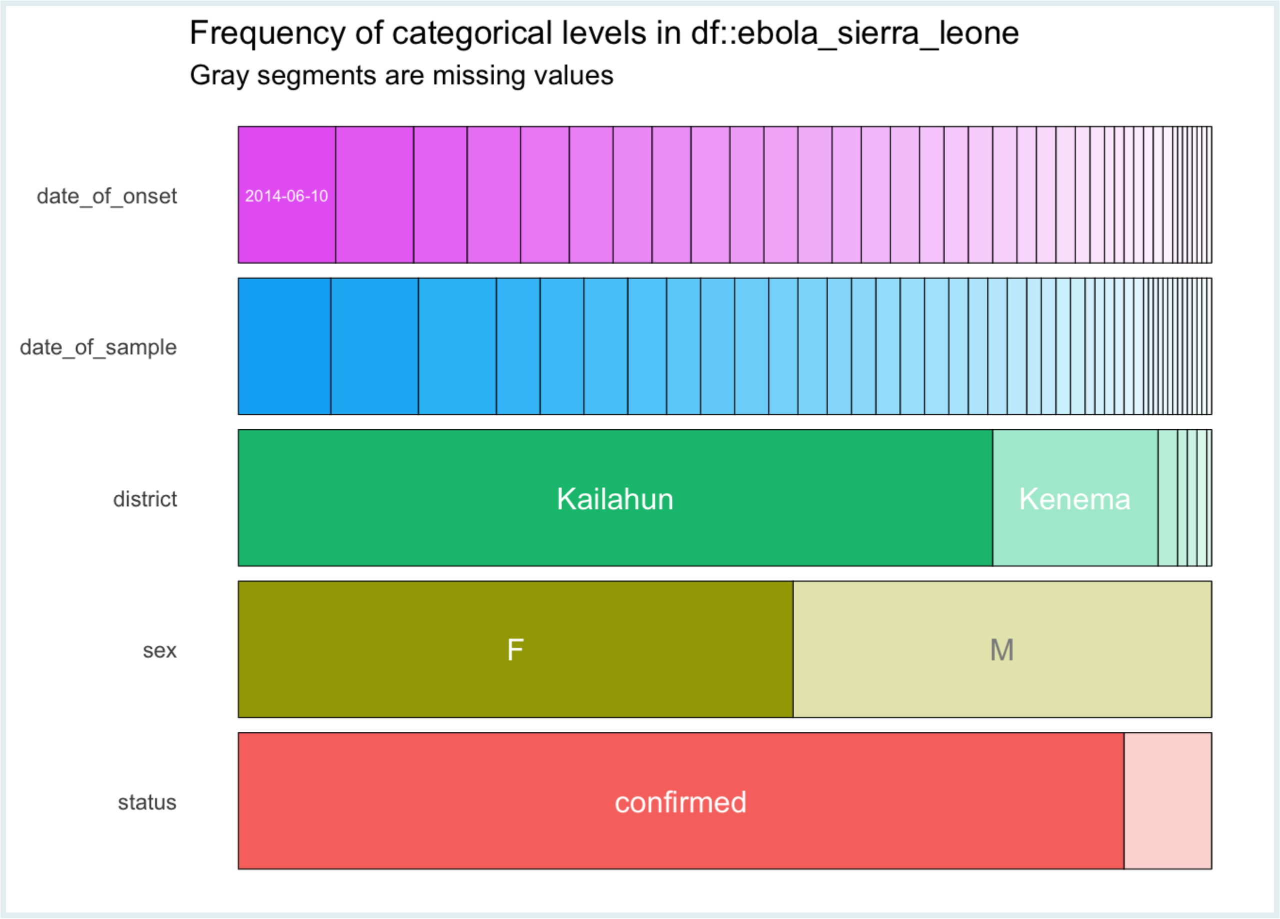
If you run `inspect_cat()` on the data object, you get a tabular summary of the categorical variables in the dataset, with some information hidden in the `levels` column (later you will learn how to extract this information).

```r
inspect_cat(ebola_sierra_leone)
```

```
## # A tibble: 5 × 5
##   col_name          cnt common     common_pcnt levels
##   <chr>           <int> <chr>            <dbl> <named list>
## 1 date_of_onset      39 2014-06-10        10   <tibble>
## 2 date_of_sample     45 2014-06-15         9.5 <tibble>
## 3 district            7 Kailahun          77.5 <tibble>
## 4 sex                 2 F                 57   <tibble>
## 5 status              2 confirmed         91   <tibble>
```

But the magic happens when you run `show_plot()` on the result from `inspect_cat()`:

```r
# store the output of `inspect_cat()` in `cat_summary`
cat_summary <- inspect_cat(ebola_sierra_leone)

# call the `show_plot()` function on that summmary.
show_plot(cat_summary)
```



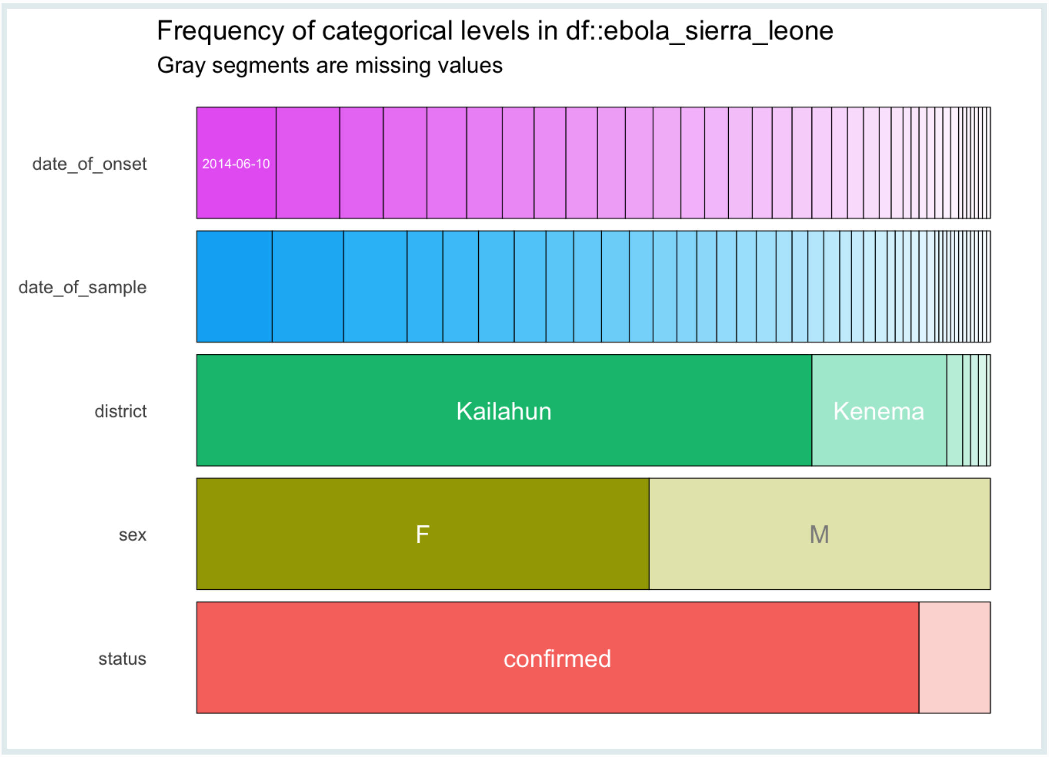
You get a wonderful figure showing the distribution of all categorical and date variables!

SIDE NOTE

You could also run:

```
show_plot(inspect_cat(ebola_sierra_leone))
```

SIDE NOTE

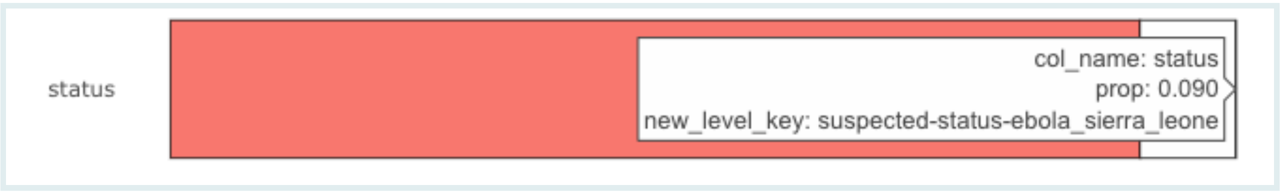


From this plot, you can quickly tell that most cases are in Kailahun, and that there are more cases in women than in men ("F" stands for "female").

One problem is that in this plot, the smaller categories are not labelled. So, for example, we are not sure what value is represented by the white section for "status" at the bottom right. To see labels on these smaller categories, you can turn this into an interactive plot with the `ggplotly()` function from the {plotly} package.

```
cat_summary_plot <- show_plot(cat_summary)
ggplotly(cat_summary_plot)
```

Wonderful! Now you can hover over each of the bars to see the proportion of each bar section. For example you can now tell that 9% (0.090) of the cases have a suspected status:



REMINDER

REMINDER

The assignment arrow, `<-`, can be written with the RStudio shortcut `alt` + `-` (**alt** AND **minus**) on Windows or `option` + `-` (**option** AND **minus**) on macOS.

You can obtain a similar plot for the numerical (continuous) variables in the dataset with `inspect_num()`. Here, we show all three steps in one go.

```
num_summary <- inspect_num(ebola_sierra_leone)
num_summary_plot <- show_plot(num_summary)
ggplotly(num_summary_plot)
```

This gives you an overview of the numerical columns, `age` and `id`. (Of course, the distribution of the `id` variable is not meaningful.)

You can tell that individuals aged 35 to 40 (mid-point 37.5) are the largest age group, making up 13.8% (0.1377…) of the cases in the dataset.

Analyzing a single numeric variable

Now that you have a sense of what the entire dataset looks like, you can isolate and analyze single variables at a time—this is called *univariate analysis*.

Go ahead and create a new section in your script for this univariate analysis.

```
# Univariate analysis, numeric variables ----
```

Let's start by analyzing the numeric `age` variable.

Extract a column vector with `$`

To extract a single variable/column from a dataset, use the dollar sign, `$` operator:

```
ebola_sierra_leone$age # extract the age column in the dataset
```

```
##   [1]  6.0 46.0   NA 25.0  8.0 49.0 13.0 50.0 35.0 38.0 60.0 18.0 10.0
14.0 50.0 35.0 43.0 17.0  3.0
##  [20] 60.0 38.0 41.0 49.0 12.0 74.0 21.0 27.0 41.0 42.0 60.0 30.0 50.0
50.0 22.0 40.0 35.0 19.0  3.0
##  [39] 34.0 21.0 73.0 65.0 30.0 70.0 12.0 15.0 42.0 60.0 14.0 40.0 33.0
43.0 45.0 14.0 14.0 40.0 35.0
##  [58] 30.0 17.0 39.0 20.0  8.0 40.0 42.0 53.0 18.0 40.0 20.0 45.0 40.0
60.0 44.0 33.0 23.0 45.0  7.0
```

```
##  [96] 26.0 37.0 30.0  3.0 56.0 32.0 35.0 54.0 42.0 48.0 11.0  1.8 63.0
55.0 20.0 62.0 62.0 42.0 65.0
## [115] 29.0 20.0 33.0 30.0 35.0   NA 50.0 16.0  3.0 22.0  7.0 50.0 17.0
40.0 21.0  9.0 27.0 52.0 50.0
## [134] 25.0 10.0 30.0 32.0 38.0 30.0 50.0 26.0 35.0  3.0 50.0 60.0 40.0
34.0  4.0 42.0   NA 54.0 18.0
## [153] 45.0 30.0 35.0 35.0 16.0 26.0 23.0 45.0 45.0 45.0 38.0 45.0 35.0
30.0 60.0  5.0 18.0  2.0 70.0
## [172] 35.0  3.0 30.0 80.0 62.0 20.0 45.0 18.0 28.0 48.0 38.0 39.0 26.0
60.0 35.0 20.0 50.0 11.0 36.0
## [191] 29.0 57.0 35.0 26.0  6.0 45.0 27.0  6.0 40.0 29.0
```

VOCAB

This list of values is called a *vector* in R. A vector is a kind of data structure that has elements of one *type*. In this case, the type is "numeric". We will formally introduce you to vectors and other data structures in a future chapter. In this lesson, you can take "vector" and "variable" to be synonyms.

Basic operations on a numeric variable

To get the mean of these ages, you could run:

```
mean(ebola_sierra_leone$age)
```

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

But it seems we have a problem. R says the mean is `NA`, which means "not applicable" or "not available". This is because there are some missing values in the vector of ages. (Did you notice this when you printed the vector?) By default, R cannot find the mean if there are missing values. To ignore these values, use the argument `na.rm` (which stands for "NA remove") setting it to `T`, or `TRUE`:

```
mean(ebola_sierra_leone$age, na.rm = T)
```

```
## [1] 33.84592
```

Great! This need to remove the `NA`s before computing a statistic applies to many functions. The `median()` function for example, will also return `NA` by default if it is called on a vector with any `NA`s:

```
median(ebola_sierra_leone$age) # does not work
median(ebola_sierra_leone$age, na.rm = T) # works
```

```
## [1] 35
```

`mean` and `median` are just two of many R functions that can be used to inspect a numerical variable. Let's look at some others.

But first, we can assign the age vector to a new object, so you don't have to keep typing `ebola_sierra_leone$age` each time.

```
age_vec <- ebola_sierra_leone$age # assign the vector to the object "age_vec"
```

Now run these functions on `age_vec` and observe their outputs:

```
sd(age_vec, na.rm = T) # standard deviation
```

```
## [1] 17.26864
```

```
max(age_vec, na.rm = T) # maximum age
```

```
## [1] 80
```

```
min(age_vec, na.rm = T) # minimum age
```

```
## [1] 1.8
```

```
summary(age_vec) # min, max, mean, quartiles and NAs
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.80   20.00   35.00   33.85   45.00   80.00       4
```

```
length(age_vec) # number of elements in the vector
```

```
## [1] 200
```

```
sum(age_vec, na.rm = T) # sum of all elements in the vector
```

```
## [1] 6633.8
```

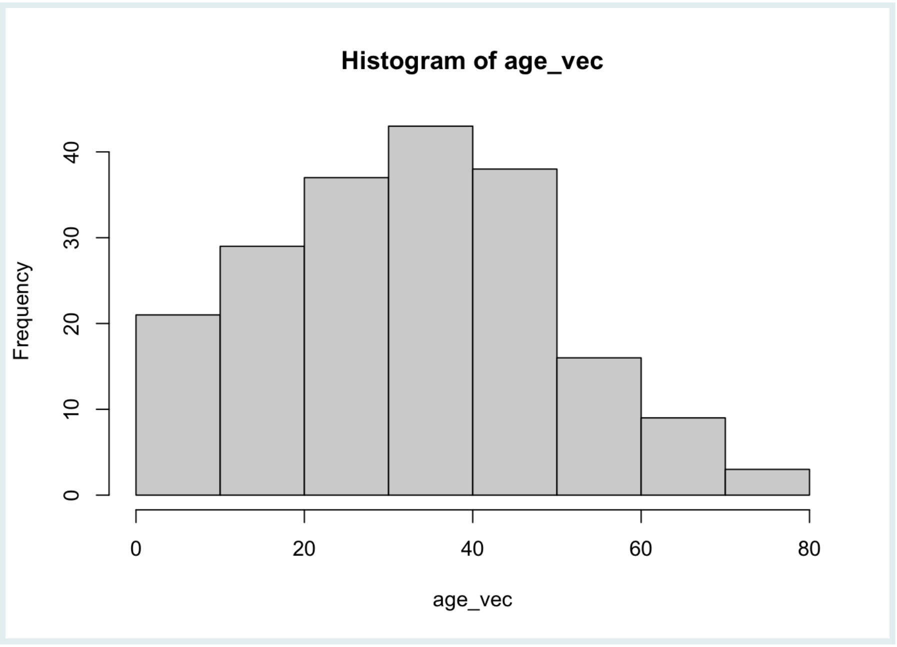
Do not feel intimidated by the long list of functions! You should not have to memorize them; rather you should feel free to Google the function for whatever operation you want to carry out. You might search something like "what is the function for standard deviation in R". One of the first results should lead you to what you need.

Visualizing a numeric variable

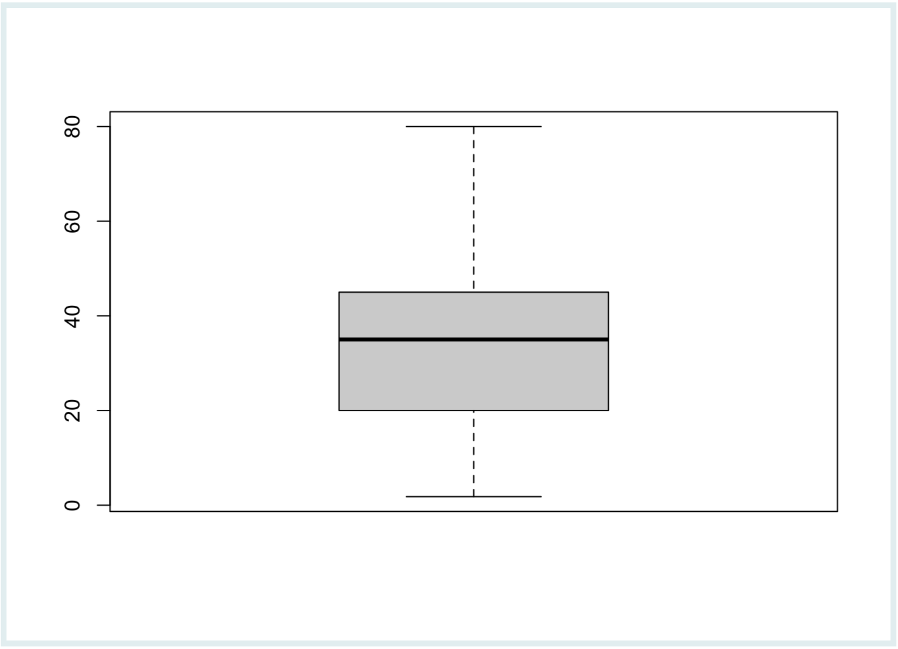
Now let's create a graph to visualize the age variable. The two most common graphics for inspecting the distribution of numerical variables are histograms (like the output of the `inspect_num()` function you saw earlier) and boxplots.

R has built-in functions for these:

```
hist(age_vec)
```



```
boxplot(age_vec)
```



Nice and easy!

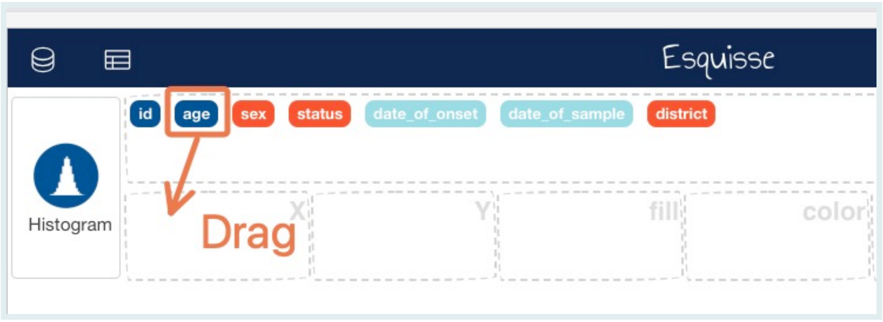
Graphical functions like boxplot() and hist() are part of R's base graphics package. These functions are quick and easy to use, but they do not offer a lot of flexibility, and it is difficult to make beautiful plots with them. So most people in the R community use an extension package, {ggplot2}, for their data visualization.

In this course, we'll use ggplot indirectly; by using the {esquisse} package, which provides a user-friendly interface for creating ggplot2 plots.

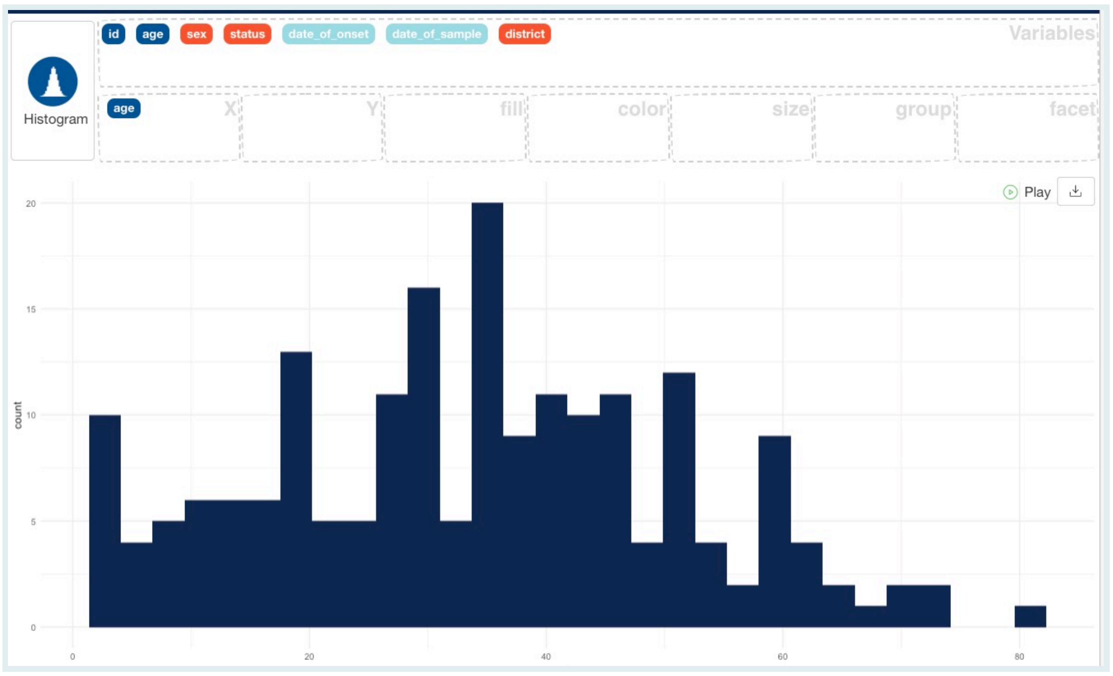
The workhorse function of the {esquisse} package is `esquisser()`, and this function takes a single argument—the dataset you want to visualize. So we can run:

```
esquisser(ebola_sierra_leone)
```

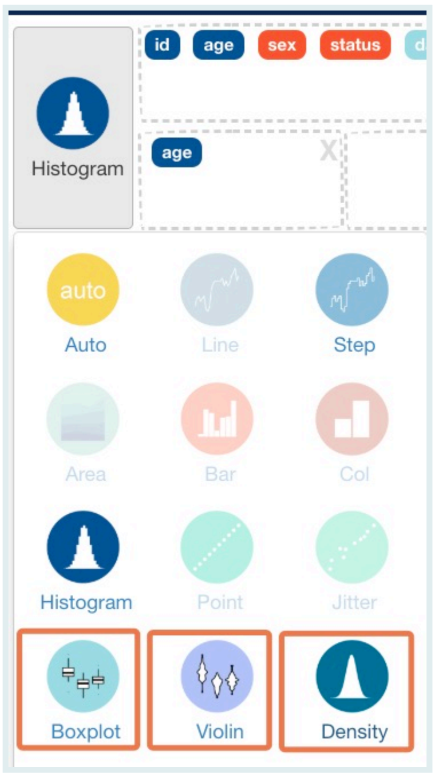
This should bring a graphic user interface that you can use to plot different variables. To visualize the age variable, simply drag `age` from the list of variables into the x axis box:



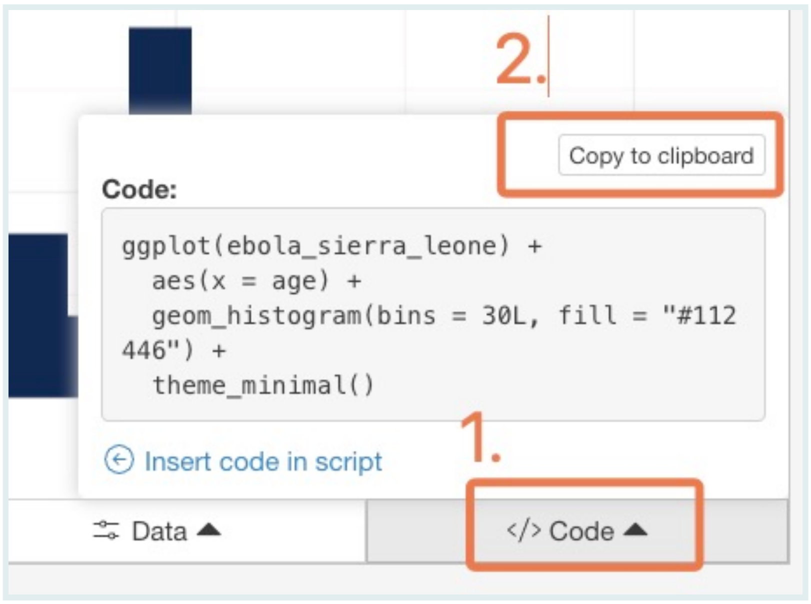
When `age` is in the x axis box, you should automatically get a histogram of ages:



You can change the plot type by clicking on the "Histogram" button and selecting one of the other valid plot types. Try out the boxplot, violin plot and density plot and observe the outputs.



When you are done creating a plot with {esquisse}, you should copy the code that was created by clicking on the "Code" button at the bottom right then "Copy to clipboard":



```
ggplot(ebola_sierra_leone) +
  aes(x = age) +
  geom_histogram(bins = 30L, fill = "#112
446") +
  theme_minimal()
```

Now, paste that code into your script, and make sure you can run it from there. The code should look something like this:

```
ggplot(ebola_sierra_leone) +
  aes(x = age) +
  geom_histogram(bins = 30L, fill = "#112446") +
  theme_minimal()
```

By copying the generated code into your script, you ensure that the data visualization you created is fully reproducible.

PRO TIP

{esquisse} can only create fairly simple graphics, so when you want to make highly customized or complex plots, you will need to learn how to write {ggplot} code manually. This will be the focus of a later course.

You should also test out the other tabs on the bottom toolbar to see what they do: Labels & Title, Plot options, Appearance and Data.

CHALLENGE

Easy bivariate and multivariate plots

In this lesson we are focusing on univariate analysis: exploring and visualizing one variable at a time. But with esquisse; it is *so* easy to make a bivariate or multivariate plot, so you can already get your feet wet with this.

Try the following plots:

CHALLENGE

- Drag `age` to the X box and `sex` to the Y box.
- Drag `age` to the X box, `sex` to the Y box, and `sex` to the fill box.
- Drag `age` to the X box and `district` to the Y box.

Analyzing a single categorical variable

Next, let's look at a categorical variable, the districts of reported cases:

```
# Univariate analysis, categorical variables ----
ebola_sierra_leone$district
```

```
##   [1] "Kailahun"      "Kailahun"      "Kenema"        "Kailahun"     
"Kailahun"      "Kailahun"     
##   [7] "Kailahun"      "Kailahun"      "Kenema"        "Kailahun"     
"Kailahun"      "Kailahun"     
##  [13] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kenema"       
##  [19] "Kono"          "Kailahun"      "Kailahun"      "Kailahun"     
"Kenema"        "Kailahun"     
##  [25] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kenema"        "Kenema"       
##  [31] "Kenema"        "Kailahun"      "Kailahun"      "Bo"           
"Kailahun"      "Kailahun"     
##  [37] "Kailahun"      "Kenema"        "Kenema"        "Kenema"       
"Kailahun"      "Kailahun"     
##  [43] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Western Urban" "Kailahun"     
##  [49] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kailahun"     
##  [55] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kailahun"     
##  [61] "Kailahun"      "Kenema"        "Western Urban" "Kambia"       
"Kailahun"      "Kailahun"     
##  [67] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kailahun"     
##  [73] "Kenema"        "Kailahun"      "Kailahun"      "Kenema"       
"Kailahun"      "Kailahun"     
##  [79] "Kenema"        "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kailahun"     
##  [85] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
"Kailahun"      "Kenema"       
##  [91] "Kailahun"      "Kailahun"      "Kailahun"      "Kono"         
"Port Loko"     "Kenema"       
##  [97] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"     
```

```
"Kenema"          "Kailahun"
## [103] "Kailahun"       "Kenema"         "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [109] "Kailahun"       "Kailahun"       "Kenema"         "Western Urban"
"Kailahun"        "Kailahun"
## [115] "Kailahun"       "Kailahun"       "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [121] "Kailahun"       "Kailahun"       "Kenema"         "Kailahun"
"Kailahun"        "Kenema"
## [127] "Kailahun"       "Port Loko"      "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [133] "Kailahun"       "Kailahun"       "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [139] "Kailahun"       "Kailahun"       "Kailahun"       "Kailahun"
"Kailahun"        "Kenema"
## [145] "Kenema"         "Kailahun"       "Kenema"         "Kailahun"
"Kailahun"        "Kailahun"
## [151] "Kailahun"       "Kailahun"       "Kenema"         "Kailahun"
"Kailahun"        "Kenema"
## [157] "Kailahun"       "Kenema"         "Kailahun"       "Kailahun"
"Kenema"          "Kailahun"
## [163] "Kailahun"       "Kailahun"       "Kailahun"       "Bo"
"Kailahun"        "Kailahun"
## [169] "Kailahun"       "Kailahun"       "Kailahun"       "Kailahun"
"Kenema"          "Kailahun"
## [175] "Kailahun"       "Kenema"         "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [181] "Kailahun"       "Kailahun"       "Kailahun"       "Western Urban"
"Kailahun"        "Kailahun"
## [187] "Kenema"         "Kailahun"       "Kailahun"       "Kailahun"
"Kailahun"        "Kailahun"
## [193] "Kailahun"       "Kenema"         "Kenema"         "Kailahun"
"Kailahun"        "Kailahun"
## [199] "Kailahun"       "Kenema"
```

Sorry for printing that very long vector!

Frequency tables

You can use the `table()` function to create a frequency table of a categorical variable:

```
table(ebola_sierra_leone$district)
```

```
##
##             Bo      Kailahun         Kambia         Kenema            Kono
Port Loko Western Urban
##              2           155              1             34               2
2             4
```

You can see that most cases are in Kailahun and Kenema.

`table()` is auseful “base” function. But there is a better function for creating frequency tables, called `tabyl()`, from the {janitor} package.

To use it, you supply the name of your data frame as the first argument, then the name of variable to be tabulated:

```
tabyl(ebola_sierra_leone, district)
```

```
##        district   n percent
##              Bo   2   0.010
##        Kailahun 155   0.775
##          Kambia   1   0.005
##         Kenema   34   0.170
##            Kono   2   0.010
##       Port Loko   2   0.010
##   Western Urban   4   0.020
```

As you can see, `tabyl()` gives you both the counts and the percentage proportions of each value. It also has some other attractive features you will see later.

PRO TIP

You can also easily make cross-tabulations with `tabyl()`. Simply add additional variables separated by a comma. For example, to create a cross-tabulation by district and sex, run:

```
tabyl(ebola_sierra_leone, district, sex)
```

```
##        district  F  M
##              Bo  0  2
##        Kailahun 91 64
##          Kambia  0  1
##          Kenema 20 14
##            Kono  0  2
##       Port Loko  1  1
##   Western Urban  2  2
```

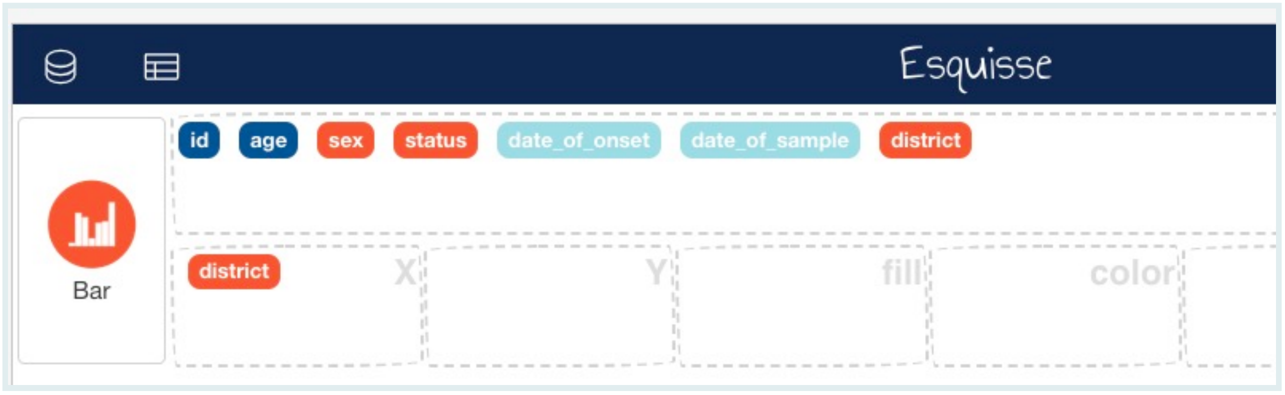
The output shows us that there were 0 women in the Bo district, 2 men in the Bo district, 91 women in the Kailahun district, and so on.

Visualizing a categorical variable

Now, let’s try to visualize the `district` variable. As before, the best way to do this is with the `esquisser()` function from {esquisse}. Run this code again:

```
esquisser(ebola_sierra_leone)
```

Then drag the `district` variable to the X axis box:



You should get a bar chart showing the count of individuals across districts. Copy the generated code and paste it into your script.

Answering questions about the outbreak

With the functions you have just learned, you have the tools to answer the questions about the Ebola outbreak that were listed at the top. Give it a go. Attempt these questions on your own, then look at the solutions below.

- **When was the first case reported? (Hint: look at the date of sample)**
- **As at the end of June 2014, which 10-year age group had had the most cases?**
- **What was the median age of those affected?**
- **Had there been more cases in men or women?**
- **What district had had the most reported cases?**
- **By the end of June 2014, was the outbreak growing or receding?**

Solutions

- **When was the first case reported?**

```
min(ebola_sierra_leone$date_of_sample)
```

```
## [1] "2014-05-23"
```

We don't have the date of report, but the first "date_of_sample" (when the Ebola test sample was taken from the patient) is May 23rd. We can use this as a proxy for the date of first report.

- **What was the median age of cases?**

```
median(ebola_sierra_leone$age, na.rm = T)
```

```
## [1] 35
```

The median age of cases was 35.

- **Are there more cases in men or women?**

```
tabyl(ebola_sierra_leone$sex)
```

```
##  ebola_sierra_leone$sex   n percent
##                       F 114    0.57
##                       M  86    0.43
```

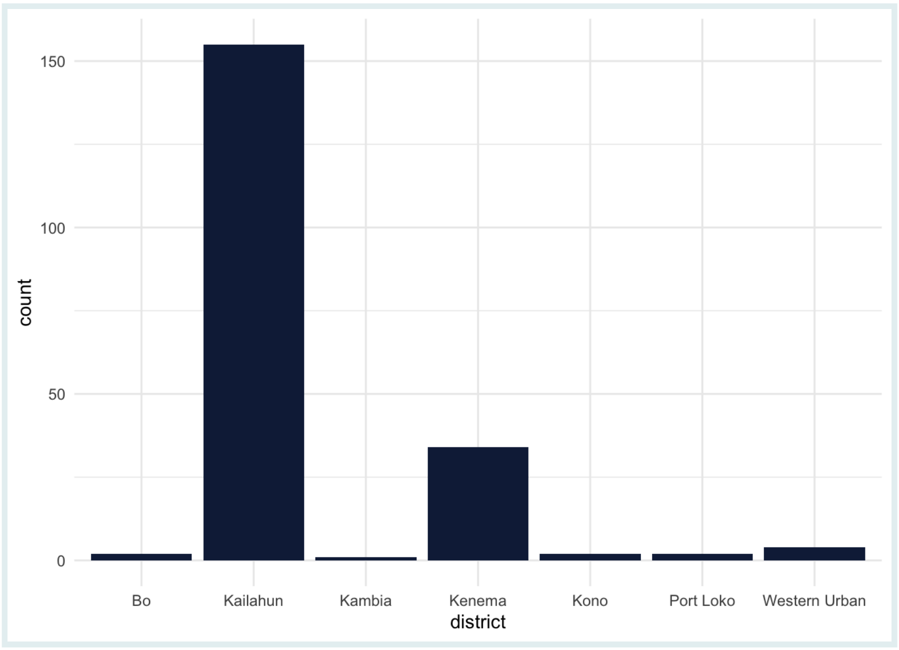
As seen in the table, there were more cases in women. Specifically, 57% of cases are of women.

- **What district has had the most reported cases?**

```
tabyl(ebola_sierra_leone$district)
```

```
##  ebola_sierra_leone$district   n percent
##                           Bo   2   0.010
##                     Kailahun 155   0.775
##                       Kambia   1   0.005
##                       Kenema  34   0.170
##                         Kono   2   0.010
##                    Port Loko   2   0.010
##                Western Urban   4   0.020
```

```
# We can also plot the following chart (generated with esquisse)
ggplot(ebola_sierra_leone) +
  aes(x = district) +
  geom_bar(fill = "#112446") +
  theme_minimal()
```

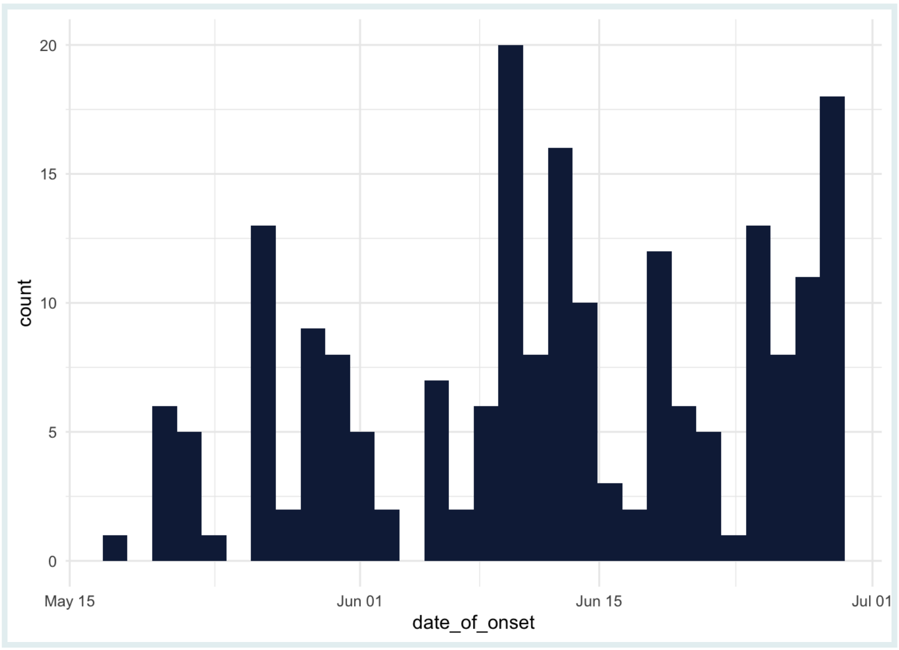


As seen, the Kailahun district had the majority of cases.

- **By the end of June 2014, was the outbreak growing or receding?**

For this, we can use esquisse to generate a bar chart that shows a count of cases in each day. Simply drag the `date_of_onset` variable to the x axis. The output code from esquisse should resemble the below:

```
ggplot(ebola_sierra_leone) +
  aes(x = date_of_onset) +
  geom_histogram(bins = 30L, fill = "#112446") +
  theme_minimal()
```



Great! But it is debatable whether the outbreak was growing or receding at the end of June 2014; a precise trend is not really clear!

Haven't had enough?

If you would like to practice some of the methods and functions you learned on a similar dataset, try downloading the data that is stored on this page: https://bit.ly/view-yaounde-covid-data

That dataset is in the form of an Excel spreadsheet, so when you are importing the dataset with RStudio, you should use the "From Excel" option (File > Import Dataset > From Excel).

This dataset contains the results of a COVID-19 serological survey conducted in Yaounde, Cameroon in late 2020. The survey estimated how many people had been infected with COVID-19 in the region, by testing for IgG and IgM antibodies. The full dataset can be obtained from here: go.nature.com/3R866wx

Wrapping up

Congratulations! You have now taken your first baby steps in analyzing data with R: you imported a dataset, explored its structure, performed basic univariate analysis and visualization on its numeric and categorical variables, and you were able to answer important questions about the outbreak based on this.

Of course, this was only a *sneak peek* of the data analysis process—a lot was left out. Hopefully, though, this sneak peek has gotten you a bit excited about what you can do with R. And hopefully, you can already start to apply some of these to your own datasets. The journey is only beginning! See you soon.

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU
Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Barnier, Julien. "Introduction à R Et Au Tidyverse." Partie 13 Diffuser et publier avec rmarkdown, May 24, 2022. https://juba.github.io/tidyverse/13-rmarkdown.html.
- Yihui Xie, J. J. Allaire, and Garrett Grolemund. "R Markdown: The Definitive Guide." Home, April 11, 2022. https://bookdown.org/yihui/rmarkdown/.

Lesson notes | RStudio projects

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Getting started: RStudio projects

Learning objectives

1. You can set up an RStudio Project and create sub-directories for input data, scripts and analytic outputs.

2. You can import and export data within an RStudio Project.

3. You understand the difference between relative and absolute file paths.

4. You recognize the value of Projects for organizing and sharing your analyses.

Introduction

Previously, you walked through some of the essential steps of data analysis, from importing data to calculating basic statistics. But you skipped over one crucial step: setting up a data analysis *project*.

Experienced data analysts keep all the files associated with a specific analysis—input data, R scripts and analytic outputs—together in a single folder. These folders are called *projects* (small p), and RStudio has built-in support for them via RStudio *Projects* (capital P).

In this lesson you will learn how to use these RStudio Projects to organize your data analysis coherently, and improve the reproducibility of your work. You will replicate some of the analysis you did in the last data dive lesson, but in the context of an RStudio Project.

Let’s get started.

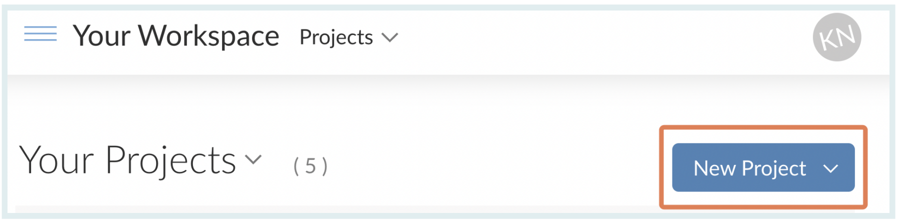
Creating a new RStudio Project

Creating a new RStudio Project looks different if you are on a local computer and if you are on RStudio Cloud. Jump to the section that is relevant for you.

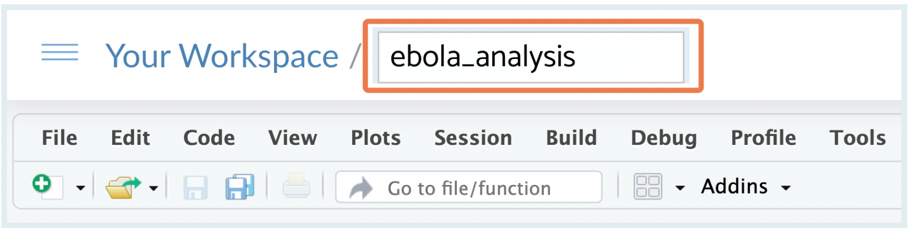
On RStudio Cloud

If you are using RStudio Cloud, you have probably *already* created a project, because you can't do any analysis without projects.

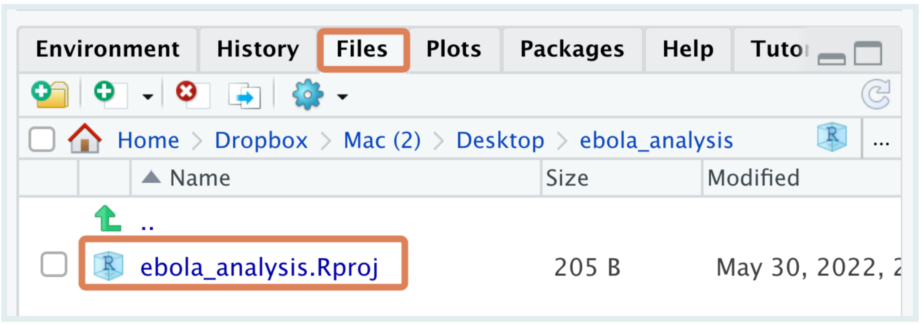
The steps are pretty simple: go to your Cloud homepage, rstudio.cloud, and click on the "New Project" button.



Name your Project something like `ebola_analysis` or `ebola_analysis_proj` if you already have a project named `ebola_analysis`.


Your Workspace / ebola_analysis
File Edit Code View Plots Session Build Debug Profile Tools
Go to file/function
Addins

The RStudio Project you have now created is just a folder on a virtual computer, which has a .Rproj file within it (and maybe a .RHistory file). You should be able to see this .Rproj file in the Files pane of RStudio:

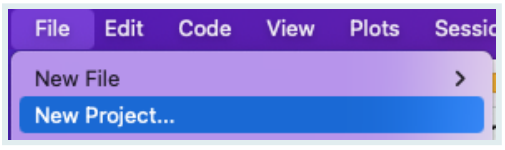

Environment History Files Plots Packages Help Tuto
Home > Dropbox > Mac (2) > Desktop > ebola_analysis
Name Size Modified
..
ebola_analysis.Rproj 205 B May 30, 2022,

KEY POINT

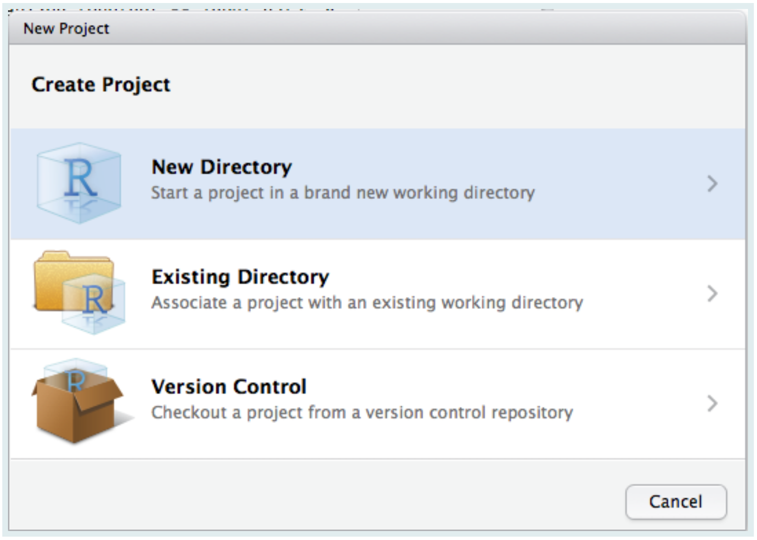
The .RProj file is what turns a regular computer folder into an "RStudio Project".

On a local computer

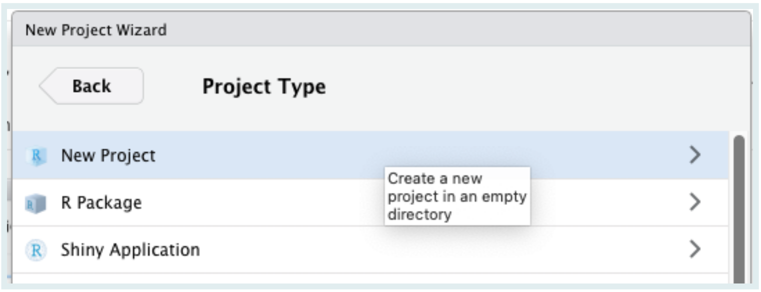
If you are on a local computer, open RStudio, then on the RStudio menu, go to `File > New Project`. Your options may look a little different from the screenshots below depending on your operating system.



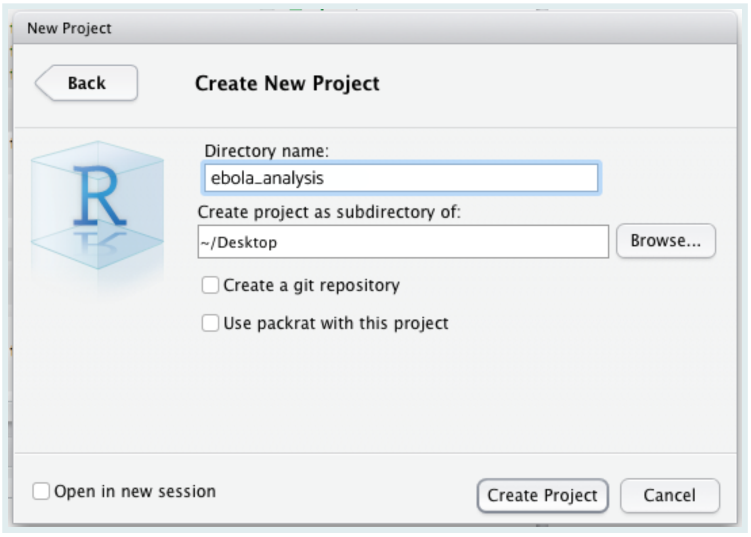
Choose "New directory"


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

Then choose "New Project":



You can call your Project something like "ebola_analysis" and make it a "subdirectory" of a folder that is easy to find, such as your desktop. (The phrase "Create project as subdirectory of" sounds scary, but it's not; RStudio is simply asking: "where should I put the project folder"?)



Directory name: ebola_analysis

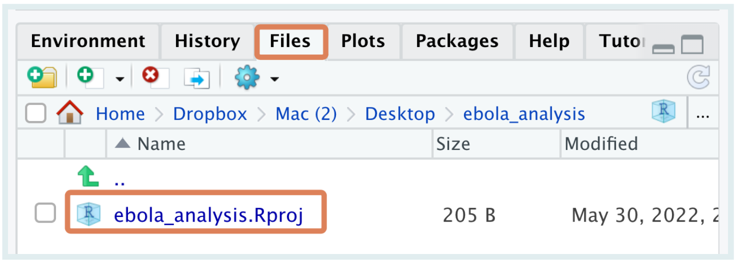
Create project as subdirectory of: ~/Desktop

Create a git repository

Use packrat with this project

Open in new session

The RStudio Project you have created is just a folder with a .Rproj file within it (and maybe a .RHistory file). You should be able to see this .Rproj file in the Files pane of RStudio:



Home > Dropbox > Mac (2) > Desktop > ebola_analysis

Name	Size	Modified
..		
ebola_analysis.Rproj	205 B	May 30, 2022, 2

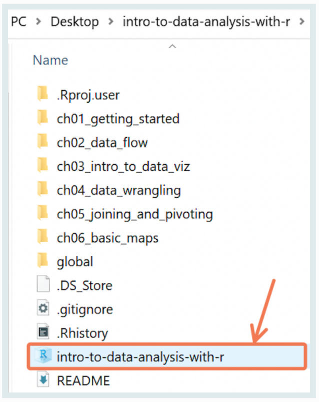
KEY POINT

Click on the .Rproj file to open your project

The .RProj file is what turns a regular computer folder into an "RStudio Project".

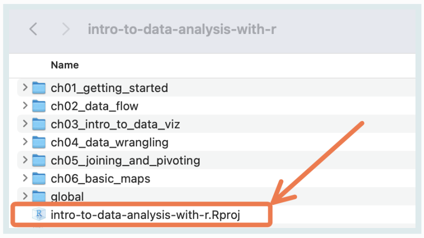
From now on, to open your project, you should double click on this .RProj file from your computer's Finder/File Explorer.

On Windows, here is an example of what a .Rproj file will look like from the File Explorer:

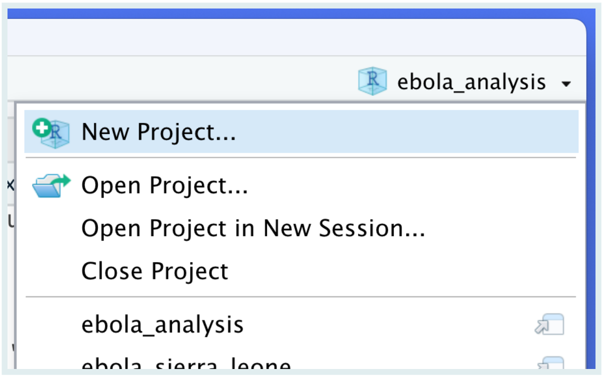


KEY POINT

On macOS, here is an example of what a .Rproj file will look like from Finder:

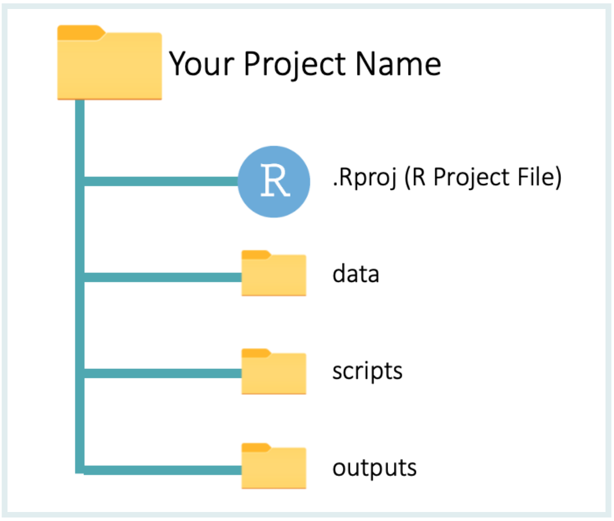


Note also that there is a header at the top right of RStudio window that tells you which Project you currently have open. Clicking on this gives you some additional Project options. You can create a new project, close a project and open recent projects, among other options.



Creating Project subfolders

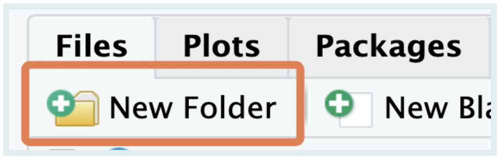
Data analysis projects usually have at least three sub-folders: one for data, another for scripts, and a third for outputs, as seen below:



Let's look at the sub-folders one by one:

- **data:** This contains the source (raw) data files that you will use in the analysis. These could be CSV or Excel files, for example.
- **scripts:** This sub-folder is where you keep your R scripts. You can also save RMarkdown files in this folder. (You will learn about RMarkdown files soon.)
- **outputs:** Here, you save the outputs of your analysis, like plots and summary tables. These outputs should be *disposable* and *reproducible*. That is, you should be able to regenerate the outputs by running the code in your scripts. You will understand this better soon.

Now go ahead and create these three sub-folders, "data", "scripts" and "outputs". within your RStudio Project folder. You should use the "New Folder" button on the RStudio Files pane to do this:



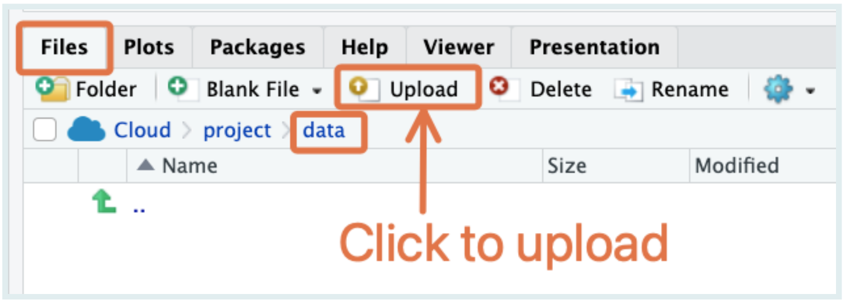
Adding a dataset to the "data" folder

Next, you should move the Ebola dataset you downloaded in the previous lesson to the newly-created "data" sub-folder (you can re-download that dataset at bit.ly/ebola-data if you can't find where you stored it).

The procedure for moving this dataset to the "data" folder is different for RStudio Cloud users and those using a local computer. Jump to the section that is relevant for you.

On RStudio Cloud

If you are on RStudio Cloud, adding the dataset to your "data" folder is straightfoward. Simply navigate to the folder within the Files pane, then click the "Upload" button:

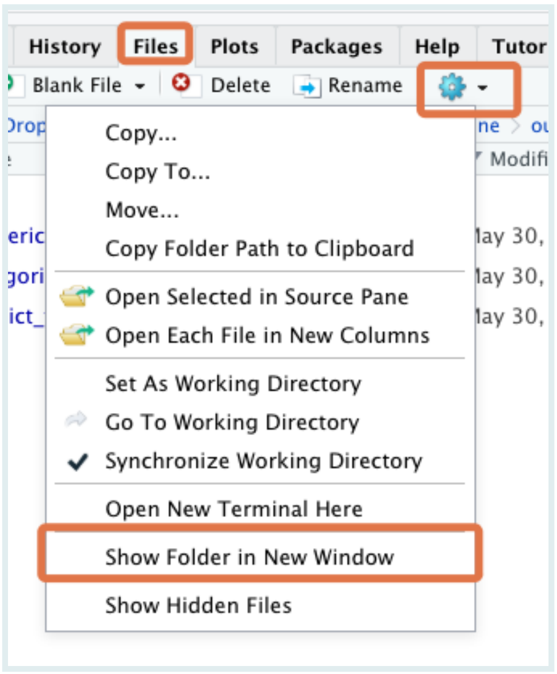


This will bring up a dialog box where you can select the file for upload.

On a local computer

On a local computer, this step has to be done with your computer's File Explorer/Finder.

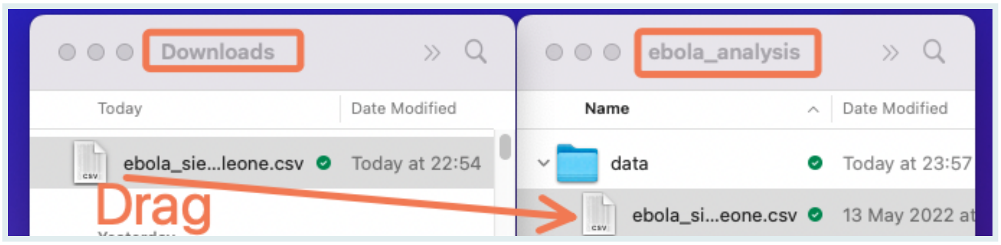
- First, locate the Project folder with your computer's File Explorer/Finder. If you're having trouble locating this, RStudio can help: go to the "Files" tab, click on "More" (the gear icon), then click "Show Folder in New Window".



  This will bring you to the Project folder in your computer's File Explorer/Finder.

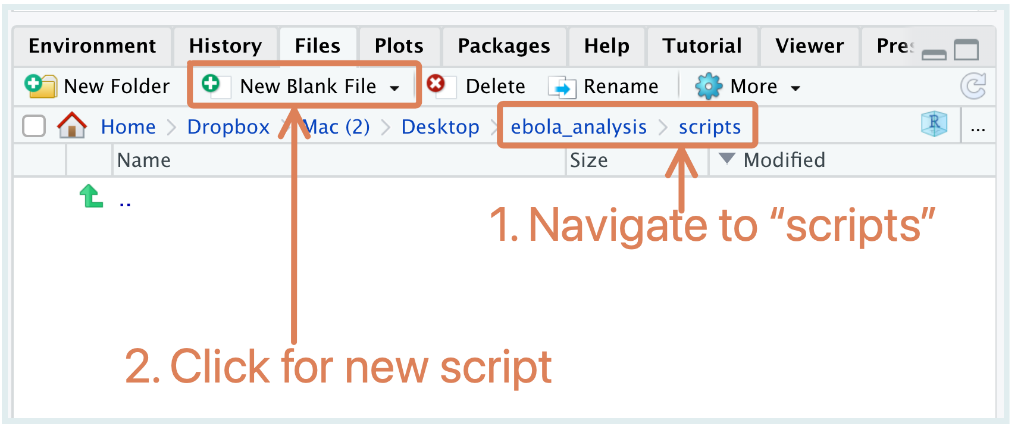
- Now, move the Ebola dataset you downloaded in the previous lesson to the newly-created "data" sub-folder.

Here is what moving the file might look like on macOS:



Creating a script in the "scripts" folder

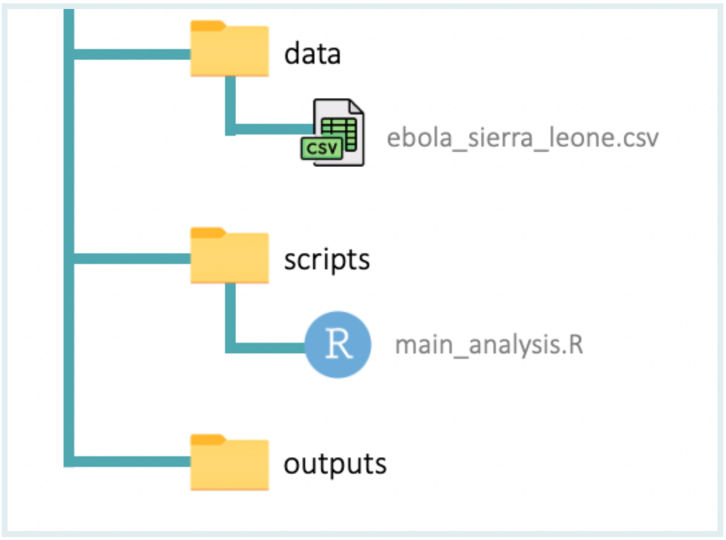
Next, create and save a new R script within the "scripts" folder. You can call this "main_analysis" or something similar. To create a new R script within a folder, first navigate to that folder in the Files pane, then click the "New Blank File" button and select "R script" in the dropdown:



SIDE NOTE

Note that this is different from what you have done so far when creating a new script (before, you used the menu option, `File > New File > New Script`). The old way is still valid; but this "New Blank File" button will probably be faster for you.

Great work so far! Now your Project folder should have the structure shown below, with the "ebola_sierra_leone.csv" dataset in the "data" folder and the "main_analysis.R" script (still empty) in the "scripts" folder:



This is a process you should go through at the start of every data analysis project: set up an RStudio Project, create the needed sub-folders, and put your datasets and scripts in the appropriate sub-folders. It can be a bit painful, but it will pay off in the long run.

The rest of this lesson will teach you how to conduct your analysis in the context of this folder setup. At the end, you will have an overall flow of data and outputs that resembles the diagram below:

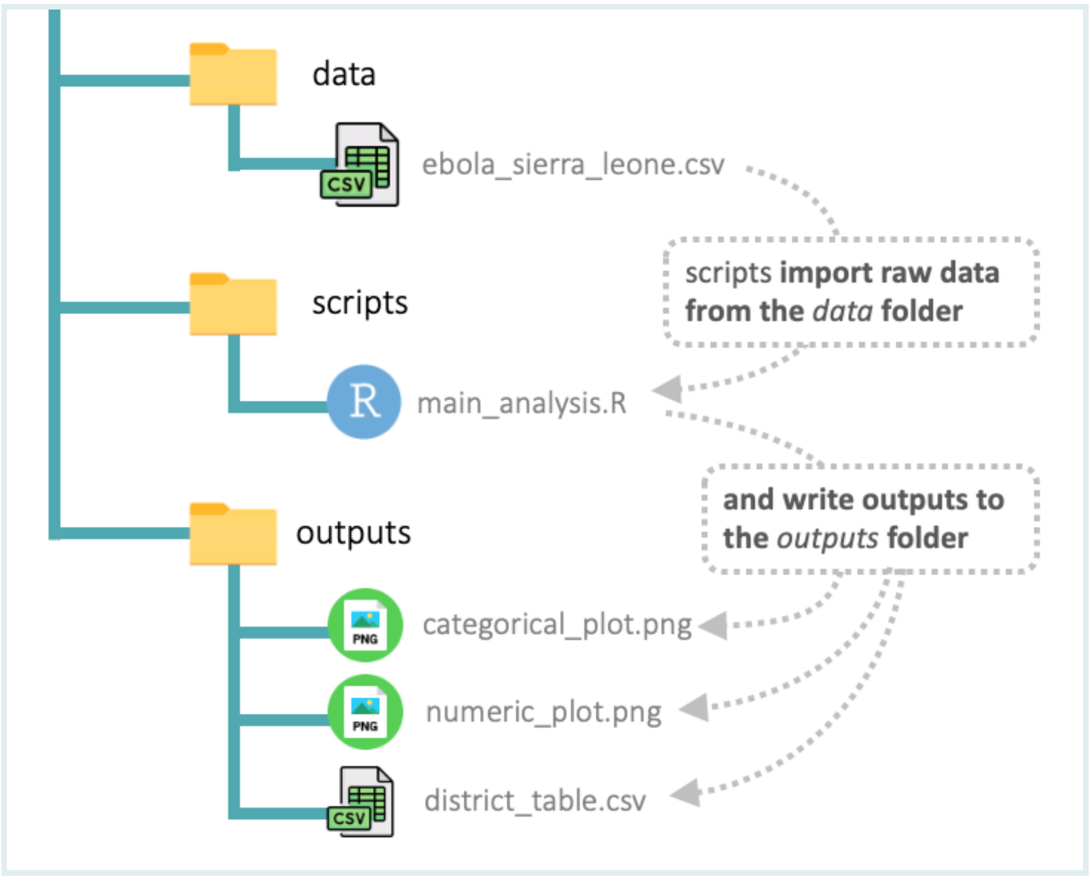


Figure: Data flow in an R project. Scripts in the “scripts” folder import data from “data” folder and export data and plots to the “outputs” folder

You should refer back to this diagram as you proceed through the sections below to help orient yourself.

Importing data from the “data” folder

We will use the code snippet below to demonstrate the flow of data through a Project. Copy and paste this snippet into your “main_analysis.R” script (but don’t run it yet). The code replicates parts of the analysis from the data dive lesson.

```r
# Ebola Sierra Leone analysis
# John Sample-Name Doe
# 2024-01-01


# Load packages ----
if(!require(pacman)) install.packages("pacman")
pacman::p_load(
  tidyverse,
  janitor,
  inspectdf,
  here # new package we will use soon
)


# Load data ----
ebola_sierra_leone <- read_csv("") # DATA PENDING! WE WILL UPDATE THIS BELOW.

# Cases by district ----
district_tab <- tabyl(ebola_sierra_leone, district)
district_tab

# Visualize categorical variables ----
categ_vars_plot<- show_plot(inspect_cat(ebola_sierra_leone))
categ_vars_plot

# Visualize numeric variables ----
num_vars_plot <- show_plot(inspect_num(ebola_sierra_leone))
num_vars_plot
```

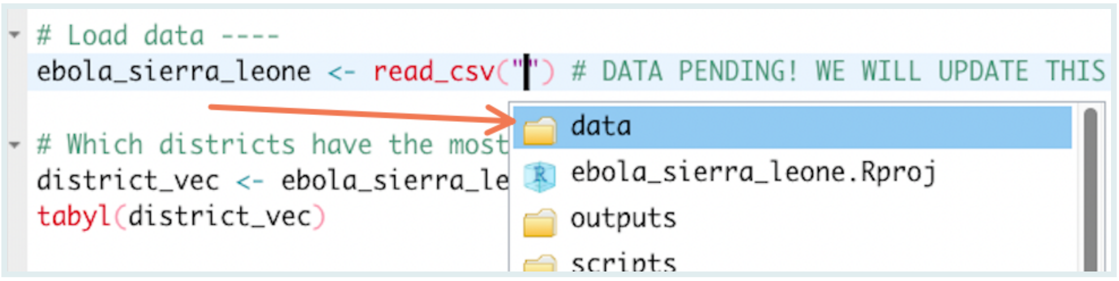
First run the "Load packages" section to install and/or load any needed packages.

Then proceed to the "Load data" section, which looks like this:

```r
# Load data ----
ebola_sierra_leone <- read_csv("") # DATA PENDING! WE WILL UPDATE THIS BELOW.
```

Here you want to import the Ebola dataset that you previously placed inside the Project's "data" folder. To do this, you need to supply the file path of that dataset as the first argument of `read_csv()`.

Because you are using an RStudio Project, this path can be obtained very easily: place your cursor inside the quotation marks within the `read_csv()` function, and press the `Tab` key on your keyboard. You should see a list of the sub-folders available in your Project. Something like this:

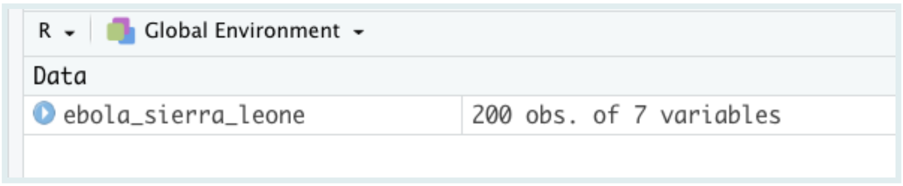

```
# Load data ----
ebola_sierra_leone <- read_csv("") # DATA PENDING! WE WILL UPDATE THIS

# Which districts have the most
district_vec <- ebola_sierra_le
tabyl(district_vec)
```

Click on the "data" folder, then press `Tab` again. Since you only have one file in the "data" folder, RStudio should automatically fill in it's name. You should now see:

```
ebola_sierra_leone <- read_csv("data/ebola_sierra_leone.csv")
```

Wonderful! Run this line of code now to import the data.

If this is successful, you should see the data appear in the Environment tab of RStudio:



```
ebola_sierra_leone    200 obs. of 7 variables
```

KEY POINT

Relative paths

The path you have used here, "data/ebola_sierra_leone.csv", is called a *relative* path, because it is relative to the *root* (or the *base*) of your Project.

How does R know where the root of your Project is? That's where the .RProj file comes in. This file, which lives in the "ebola_analysis" folder tells R "here! Here! I am in the 'ebola_analysis' folder so this must be the root!". Thus, you only need to specify path components that are *deeper* than this root.

RStudio Projects, and the relative paths they allow you to use, are important for reproducibility. Projects that use relative paths can be run on anyone's computer, and the importing and exporting code should work without any hiccups. This means that you can send someone an RStudio Project folder and the code should run on their machine just as it ran on yours!

This would not be the case if you were to use an *absolute* path, something like "~/Desktop/my_data_analysis/learning_r/ebola_sierra_leone.csv", in your

KEY POINT

script. Absolute paths give the full address of a file, and will not usually work on someone else's computer, where files and folders will be arranged differently.

RSTUDIO CLOUD

Note that if you are using RStudio Cloud, you are *forced* to use relative paths, because you cannot access the general file system of the virtual computer; you can only work within specific Project folders.

Using `here::here()`

As you have now seen, RStudio Projects simplify the data import process and improve the reproducibility of your analysis, primarily because they allow you to use relative paths.

But there is one more step we recommend when using relative paths: rather than leave your path *naked*, wrap it in the `here()` function from the {here} package.

So, in the data import section of your script, change `read_csv()`'s input from `"data/ebola_sierra_leone.csv"` to `here("data/ebola_sierra_leone.csv")`:

```
ebola_sierra_leone <- read_csv(here("data/ebola_sierra_leone.csv"))
```

What is the point of wrapping the path in `here()`? Well, technically, this is no real point in doing this in an *R* script; the importing code works fine without it. But it *will* be necessary when you start using *RMarkdown* scripts (which you will soon be introduced to), because paths not wrapped in `here()` are problematic in the RMarkdown context.

So to keep things consistent, we always recommend you use `here()` when pointing to paths, whether in an R script or an RMarkdown script

Exporting data to the "outputs" folder

Importing data is not the only benefit of RStudio Projects; data export is also streamlined when you use Projects. Let's look at this now.

In the "Cases by district" section of your script, you should have:

```
# Cases by district ----
district_tab <- tabyl(ebola_sierra_leone, district)
district_tab
```

Run this code now; you should get the following tabular output:

```
##        district   n percent
##              Bo   2   0.010
##        Kailahun 155   0.775
##          Kambia   1   0.005
##          Kenema  34   0.170
##            Kono   2   0.010
##       Port Loko   2   0.010
##   Western Urban   4   0.020
```

Now, imagine that you want to export this table as a CSV. It would be nice if there was a specific folder designated for such exports. Well, there is! It's the "outputs" folder you created earlier. Let's export your table there now. Type out the code below (but don't run it yet):

```
write_csv(x = district_tab, file = "")
```

With the `write_csv()` function, you are going to "write" (or "save") the `district_tab` table as a CSV file.

The `x` argument of `write_csv()` takes in the object to be saved (in this case `district_tab`). And the `file` argument takes in the target file path. This target file path can be a simple relative path: "outputs/district_table.csv". (And, as mentioned before, we should wrap the path in `here()`.) Type this up and run it now:

```
write_csv(x = district_tab, file = here("outputs/district_table.csv"))
```

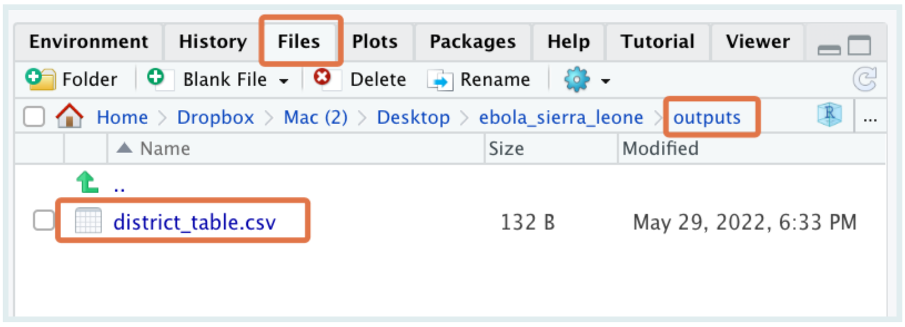
The path "outputs/district_table.csv" tells `write_csv()` to save the plot as a CSV file named "districts_table" in the "outputs" folder of the Project.

SIDE NOTE

You can replace "district_table.csv" with any other appropriate name, for example "freq table across districts.csv":

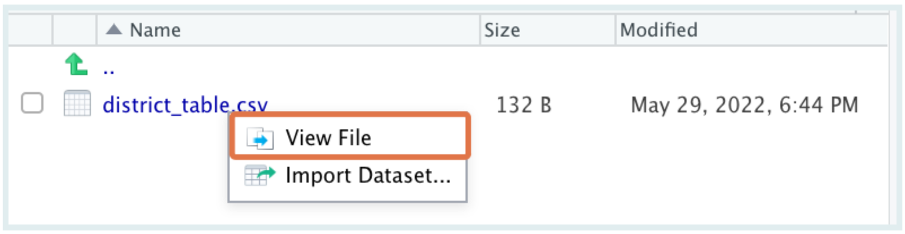
```
write_csv(x = district_tab, file = here("outputs/freq table
         across districts.csv"))
```

Great work! Now, if you go to the Files tab and navigate to the outputs folder of your Project, you should see this newly created file:



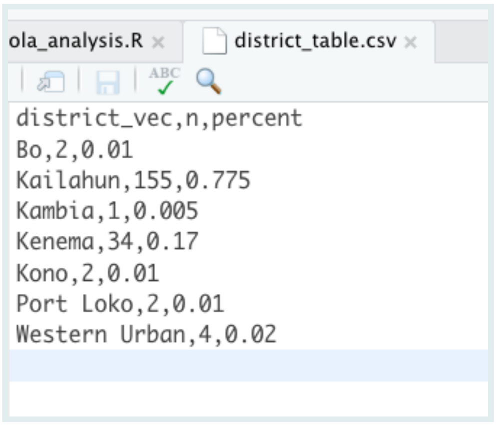
district_table.csv 132 B May 29, 2022, 6:33 PM

You can click on the file to view it within RStudio as a raw CSV:



district_table.csv 132 B May 29, 2022, 6:44 PM

This should bring up an RStudio viewer window:



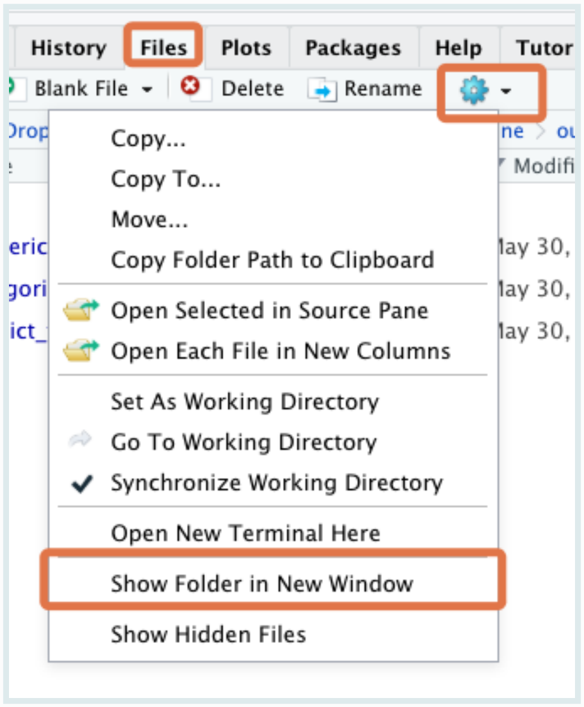
```
district_vec,n,percent
Bo,2,0.01
Kailahun,155,0.775
Kambia,1,0.005
Kenema,34,0.17
Kono,2,0.01
Port Loko,2,0.01
Western Urban,4,0.02
```

If you instead want to view the CSV in Microsoft Excel, you can navigate to the same file in your computer's Finder/File Explorer and double-click on it from there.

REMINDER

To locate your Project folder in your computer's Finder/File Explorer, go the "Files" tab, click on the gear icon, then click "Show Folder in New Window".

REMINDER



RSTUDIO CLOUD

If you are on RStudio cloud, then you won't be able to view the CSV in Microsoft Excel until you have "exported" it. Use the "Export" menu option in the Files tab. If this is not immediately visible, click on the gear icon to bring up "More" options, then scroll through to find the "Export" option.

Overwriting data

If you need to update the output CSV, you can simply rerun the `write_csv()` function with the updated data object.

To test this, replace the "Cases by district" section of your script with the following code. It uses the `arrange()` function to arrange the table in order of the number of cases, `n`:

```
# Cases by district ----
district_tab <- tabyl(ebola_sierra_leone, district)
district_tab_arranged <- arrange(district_tab, -n)
district_tab_arranged
```

(`-n` means "sort in descending order of the `n` variable"; we will introduce you to the arrange function properly later on.)

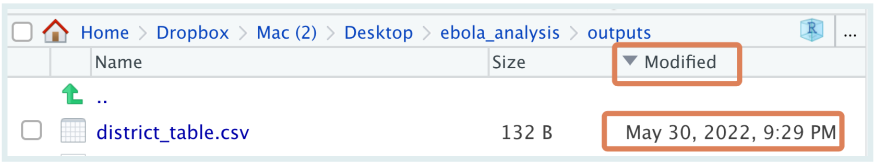
The output should be:

```
##        district   n percent
##        Kailahun 155   0.775
##          Kenema  34   0.170
##   Western Urban   4   0.020
##              Bo   2   0.010
##            Kono   2   0.010
##       Port Loko   2   0.010
##          Kambia   1   0.005
```

You can now overwrite the old “district_table.csv” file by re-running the write_csv function with the `district_tab` object:

```
write_csv(x = district_tab_arranged, file =
        here("outputs/district_table.csv"))
```

To verify that the dataset was actually updated, observe the “Modified” time stamp in the RStudio Files pane:



Home > Dropbox > Mac (2) > Desktop > ebola_analysis > outputs

Name	Size	Modified
..		
district_table.csv	132 B	May 30, 2022, 9:29 PM

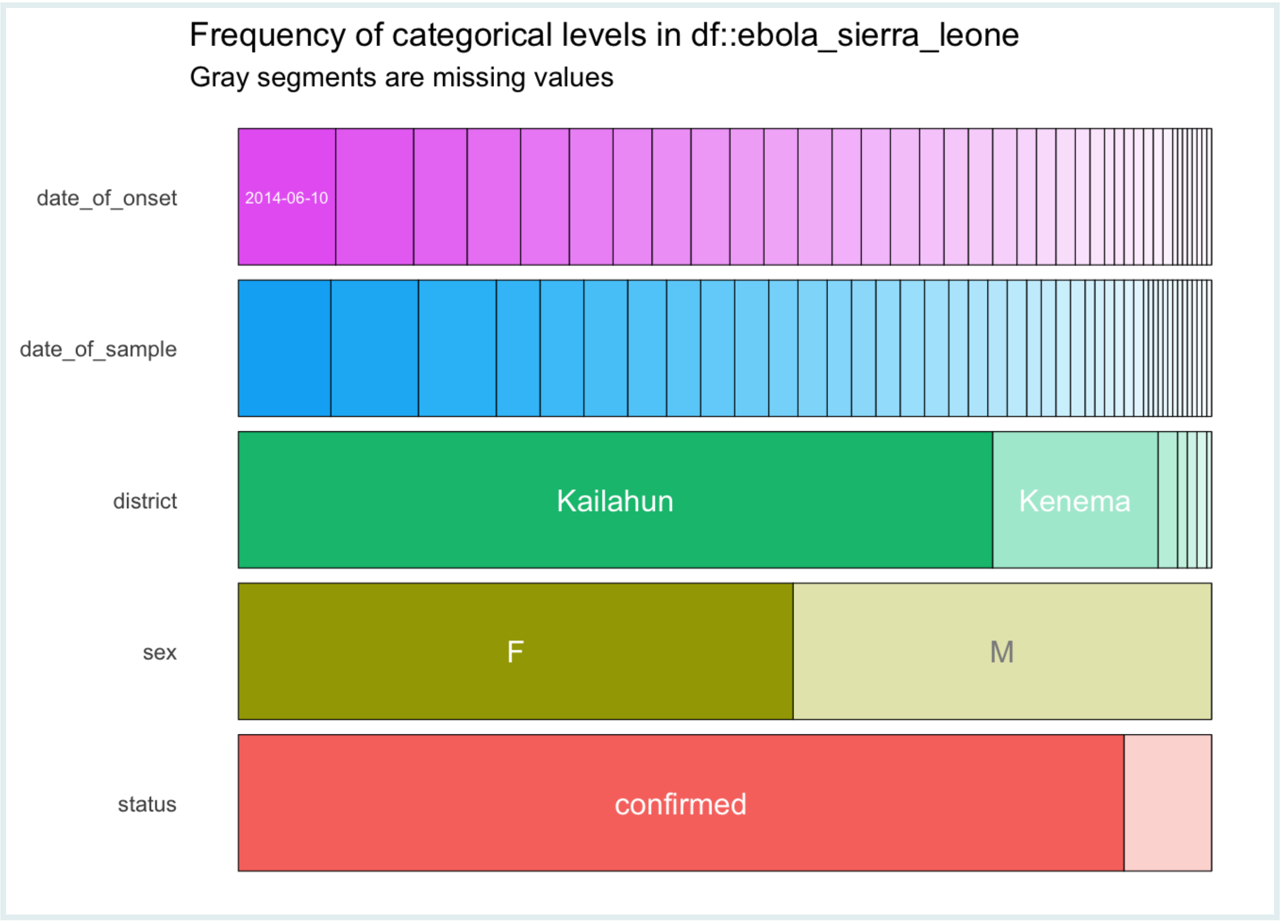
Exporting plots to the “outputs” folder

Finally, let’s look at plot exporting in the context of an RStudio Project.

In the “Visualize categorical variables” section of your script, you should have:

```
# Visualize categorical variables ----
categ_vars_plot<- show_plot(inspect_cat(ebola_sierra_leone))
categ_vars_plot
```

Running these code lines should give you this output:



Below these lines, type up the `ggsave()` command below (but don't run it yet):

```
ggsave(filename = "", plot = categ_vars_plot)
```

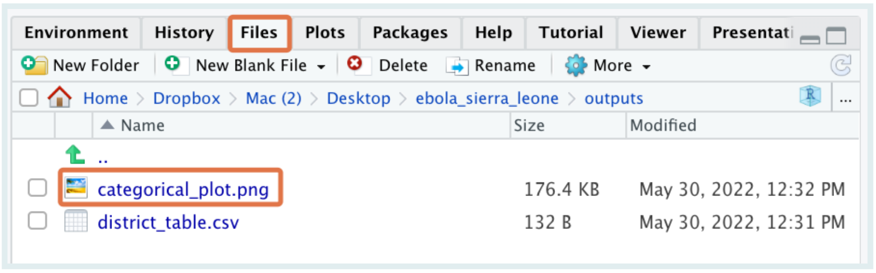
This command uses the `ggsave()` function to export the `categ_vars_plot` figure. The `plot` argument of `ggsave()` takes in the object to be saved (in this case `categ_vars_plot`), and the `filename` argument takes in the target file path for the plot.

As you saw when exporting data, this target file path is quite simple because you are working in an RStudio Project. In this case, you have:

```
ggsave(filename = "outputs/categorical_plot.png", plot = categ_vars_plot)
```

Run this `ggsave()` command now. The path "outputs/categorical_plot.png" tells `ggsave()` to save the plot as a PNG file named "categorical_plot" in the "outputs" folder of the Project.

To see this newly-saved plot, navigate to the Files tab. You can click on it to open it with your computer's default image viewer:



Name	Size	Modified
..		
categorical_plot.png	176.4 KB	May 30, 2022, 12:32 PM
district_table.csv	132 B	May 30, 2022, 12:31 PM

Also note that the the `ggsave()` function lets you save plots to multiple image formats. For example, you could instead write:

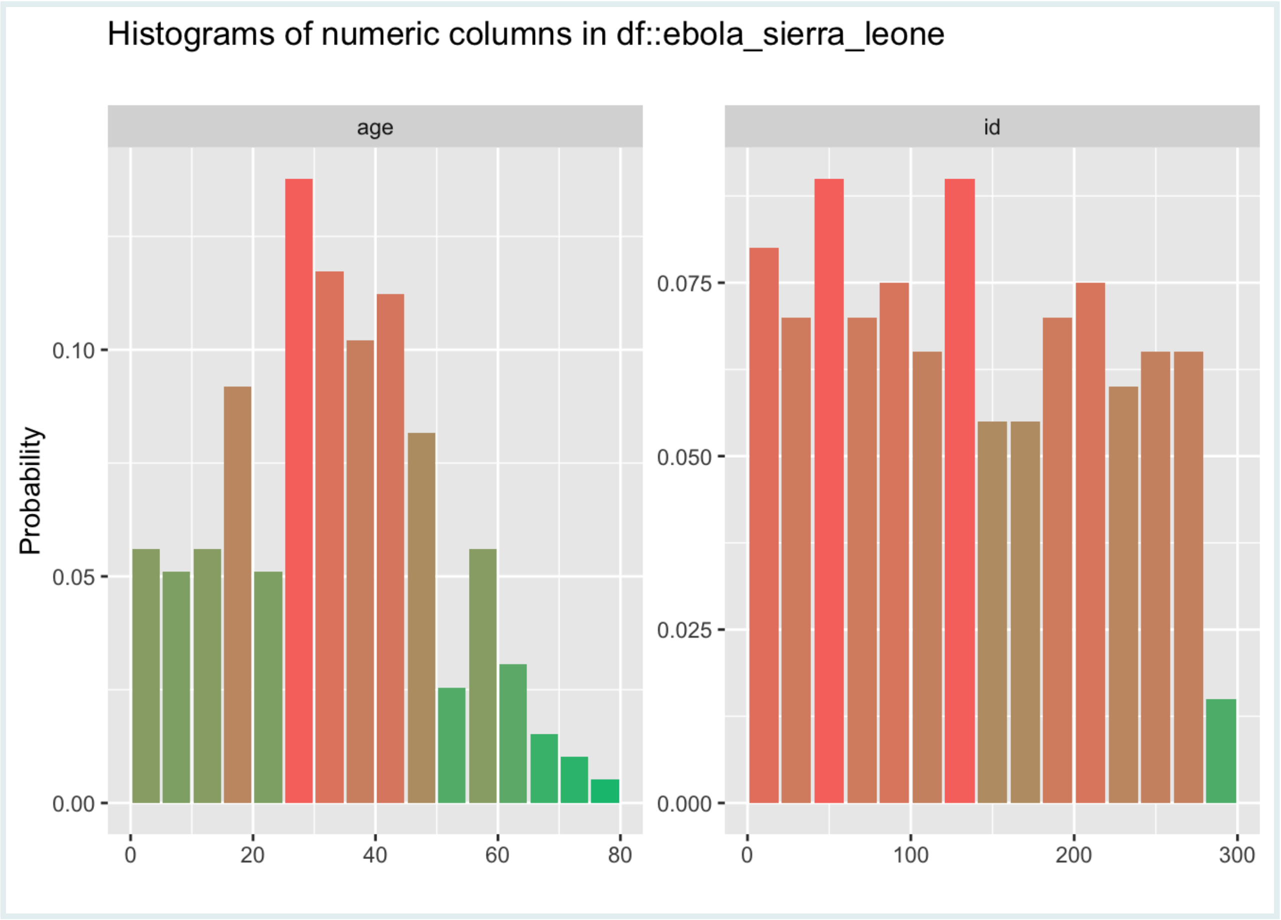
```
ggsave(filename = "outputs/categorical_plot.pdf", plot = categ_vars_plot)
```

to save the plot as a PDF. Run `?ggsave` to see what other formats are possible.

Now let's export the second plot, the numerical summary. In the section of your script called "Visualize numeric variables", you should have:

```
# Visualize numeric variables ----
num_vars_plot <- show_plot(inspect_num(ebola_sierra_leone))
num_vars_plot
```

Running these code lines should give you this output:



To export this plot, type up and run the following code:

```
ggsave(filename = "outputs/numeric_plot.png", plot = num_vars_plot)
```

Wonderful!

Sharing a Project

Projects are also great for sharing your analysis with collaborators.

You can zip up your Project folder and send it to a colleague through email or through a file sharing service like Dropbox. The colleague can then unzip the folder, click on the .Rproj file to open the Project in RStudio, and re-do and edit all your analysis steps.

This is a decent setup, but sending projects back and forth may not be ideal for long-term collaboration. So experienced analysts use a technology called *git* to collaborate on projects. But this topic is a bit too advanced for this course; we will cover it in detail in a future course. If you are impatient, you can check out this book chapter: https://intro2r.com/github_r.html

Wrapping up

Congratulations! You now know how to set up and use RStudio Projects!

Hopefully you see the value of organizing your analysis scripts, data and outputs in this way. Projects are a coherent way to structure your analyses, and make it easy to revisit, revise and share your work. They will be the foundation for much of your work as a data analyst going forward.

That's it for now. See you in the next lesson.

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Wickham, H., & Grolemund, G. (n.d.). *R for data science*. 8 Workflow: projects | R for Data Science. Retrieved May 31, 2022, from https://r4ds.had.co.nz/workflow-projects.html

Lesson notes | R Markdown

Created by the GRAPH Courses team

April 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Introduction

The {rmarkdown} package allows you to dynamically generate documents by mixing formatted text and results produced by R code. The generated documents can be in HTML, PDF, Word, and many other formats. It is therefore a very practical tool for exporting, communicating and disseminating analysis results.

There is a whole book on Rmarkdown, so we can only cover some of the essentials here.

This document was itself generated from R Markdown files.

Learning objectives

- You can create and knit an Rmarkdown document containing code and free text.
- You can output documents to multiple formats including HTML, PDF, Word, Powerpoint and flexdashboards.
- You understand basic markdown syntax.
- You can use R chunk options, including *eval*, *echo*, and *message*.
- You know the syntax for in-line R code.
- You recognize some useful packages for table formatting in Rmarkdown.
- You understand how to use the {here} package to force Rmarkdown files to use the project folder as the working directory.

Project setup

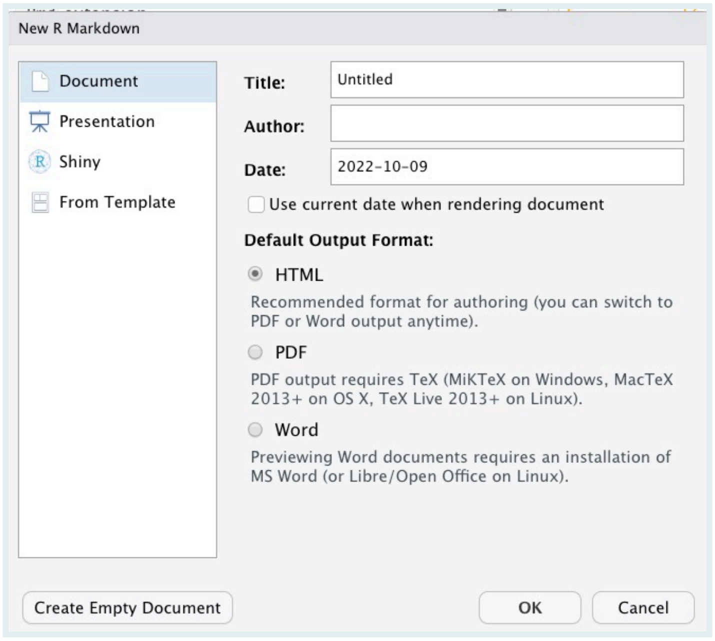
In RStudio, click on the *File* menu, and select *New Project...*. Then click on *New Directory*. Give your project a name, and select a directory into which to place it. (Make sure you remember where you put it!) Once you have these fields filled out, click *Create Project*.

Next we're going to set up some folders inside the project. Go to the *Files* pane, and click on *New Folder*. Name this one "data", and click *OK*. This is where you will put the data related to this project. Create one more called "rmd". R Markdown documents will go here.

Create a new document

An R Markdown document is a simple text file saved with the `.Rmd` extension.

In RStudio, you can create a new document by going to the *File* menu then choosing *New file* then *R Markdown...*. The first time you create an R Markdown document, you may be asked to install several packages. Go ahead and install those. Once RStudio has the appropriate packages, the following dialog box appears :



Title: Untitled
Author:
Date: 2022-10-09
☐ Use current date when rendering document

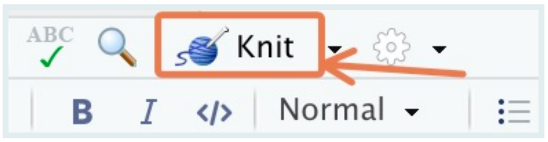
Default Output Format:
● HTML
Recommended format for authoring (you can switch to PDF or Word output anytime).
○ PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).
○ Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

For now, you can leave all the default values and click OK. A file with sample content is then displayed.

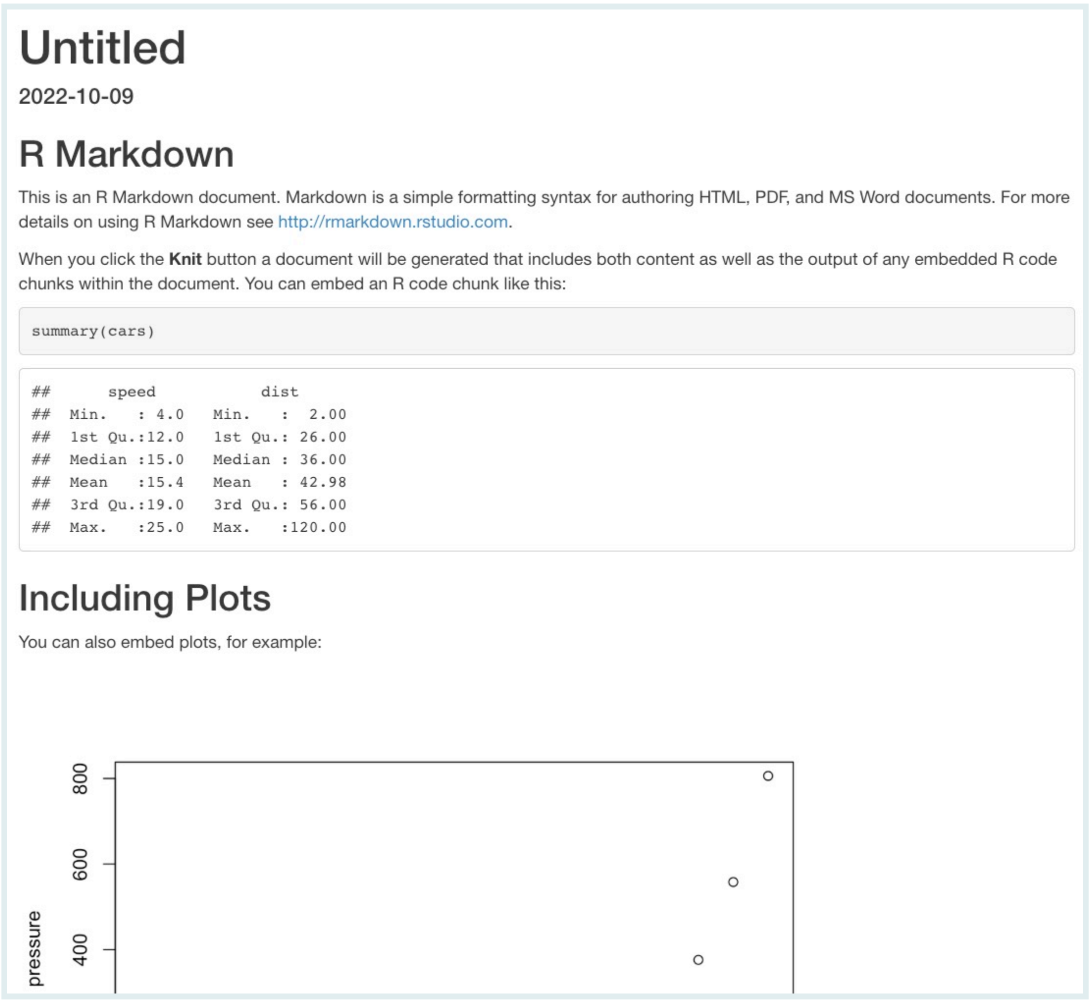
Try editing some of the text in the file. Notice that is made up of some free text and some code sections.

Save your file with `Cmd/Ctrl + S`, remembering to give it the extension ".Rmd". E.g. "ebola_analysis.Rmd". Be sure to save it in the "rmd" folder you just created.

You can now try rendering the document by clicking on the "knit" button at the top right:



This will create an HTML output that looks like this:



Untitled

2022-10-09

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00  
```

Including Plots

You can also embed plots, for example:

This new rendered file is stored in the same directory as your Rmd. It has the same name, except it ends with ".html" instead of ".rmd".

VOCAB

HTML stands for Hyper text markup language and is the format that is used for most documents on the web.

Rmarkdown Header (YAML)

Now let's return to the rest of the Rmd to consider it part by part.

The first part of the document is its *header*. (It is also called "YAML", which stands for "Yet another markup language".) (The name is intended to be humorous.)

```
---
title: "Untitled"
output: html_document
date: "2022-10-09"
---
```

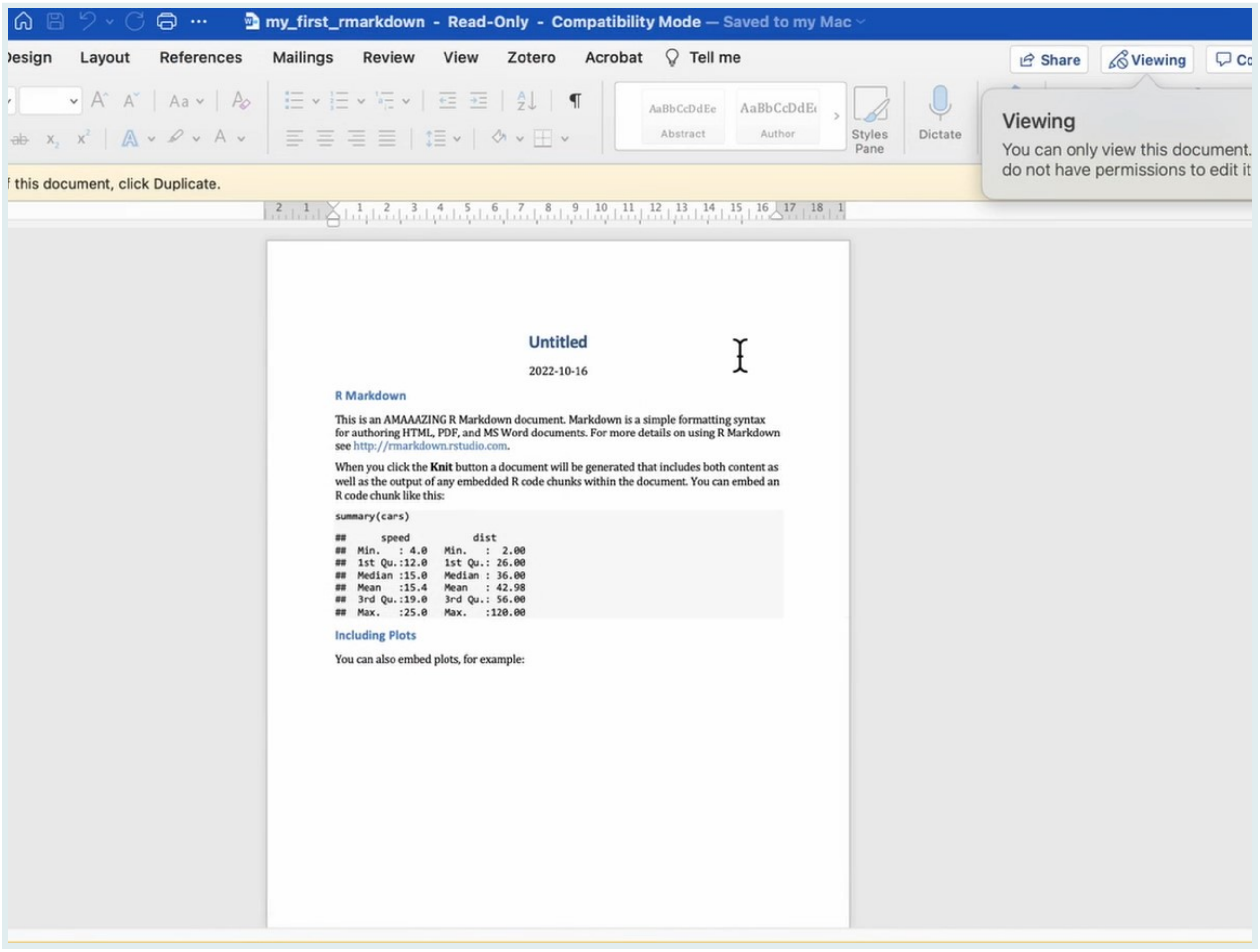
The YAML header must be located at the very beginning of the document, delimited by three dashes (`---`) before and after.

This header contains the document's metadata, such as its title, author, date, plus a whole host of possible options that will allow you to configure or customize the entire document and its rendering. Here, for example, the line `output: html_document` indicates that the generated document must be in HTML format.

We can change the `html_document` text to try out some other formats.

First you can make so

With the output set to "word_document", we get something like this:


my_first_rmarkdown - Read-Only - Compatibility Mode — Saved to my Mac
esign
Layout
References
Mailings
Review
View
Zotero
Acrobat
Tell me
Share
Viewing
Styles Pane
Dictate
Viewing
You can only view this document. do not have permissions to edit it
this document, click Duplicate.

Untitled

2022-10-16

R Markdown

This is an AMAAAZING R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00  
```

Including Plots

You can also embed plots, for example:

Image of the r markdown document open in the Microsoft Word program

Note that this creates a ".docx" version of our document in the "rmd" folder.

With the output set to "powerpoint_document", it comes out like this:

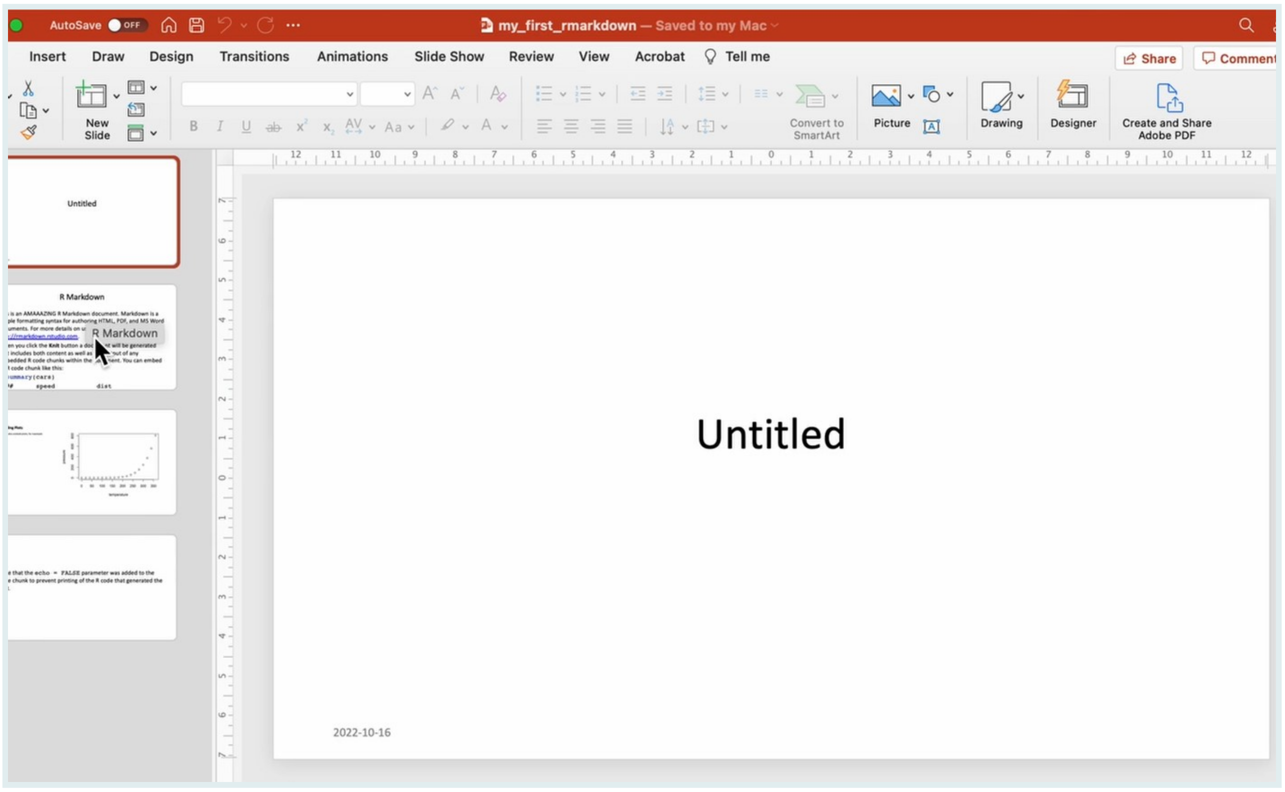
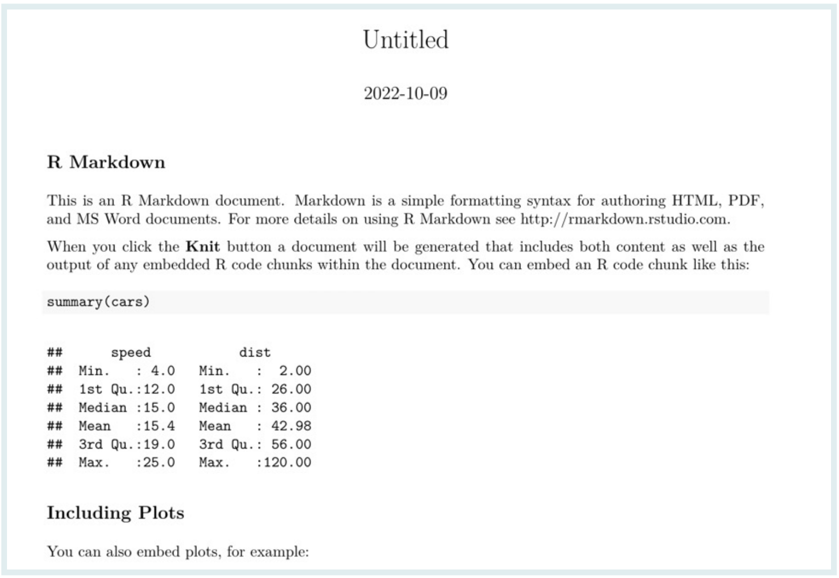


Image of the r markdown document open in the Microsoft Powerpoint program

If we change the output setting to "pdf_document", we can get the same document in PDF format (for this you may be prompted to install tinytex on your computer, see below):



Untitled

2022-10-09

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Including Plots

You can also embed plots, for example:

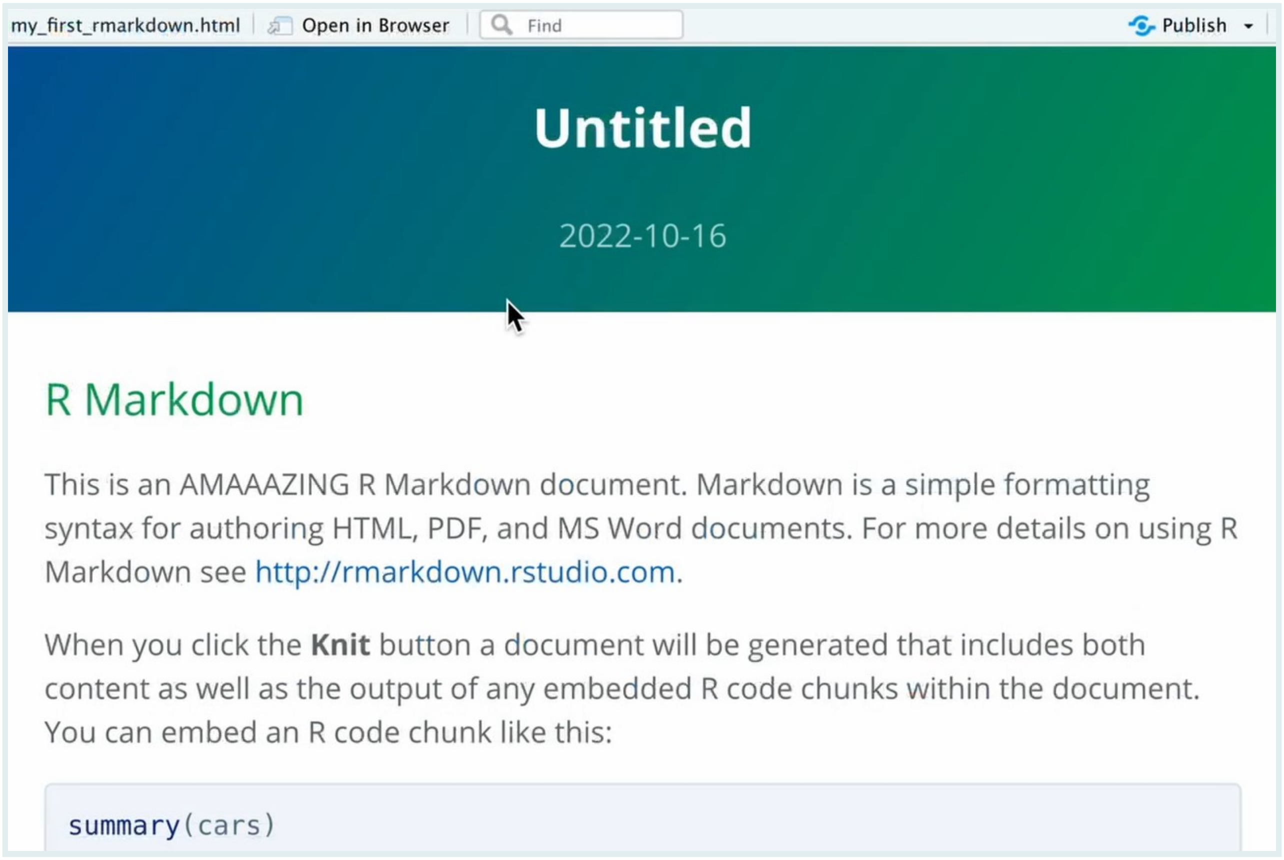
KEY POINT

For PDF generation, you must have a working `LaTeX` installation on your system. If not, Yihui Xie's `tinytex` extension aims to make it easier to install a minimal `LaTeX` distribution regardless of your machine's

KEY POINT

operating system. To use it, you must first install the extension with `install.packages('tinytex')`, then run the following command in the console (expect a download of about 200MB): `tinytex::install_tinytex()` More information on the tinytex website.

There is also a file format called "prettydoc". To try this out, type `install.packages('prettydoc')` into the console and hit *enter*. The output format for prettydoc is a little different than the previous three we've seen, you need to type `prettydoc::html_pretty` in the `output` section. When you knit a prettydoc, you should see something like this:



Untitled

2022-10-16

R Markdown

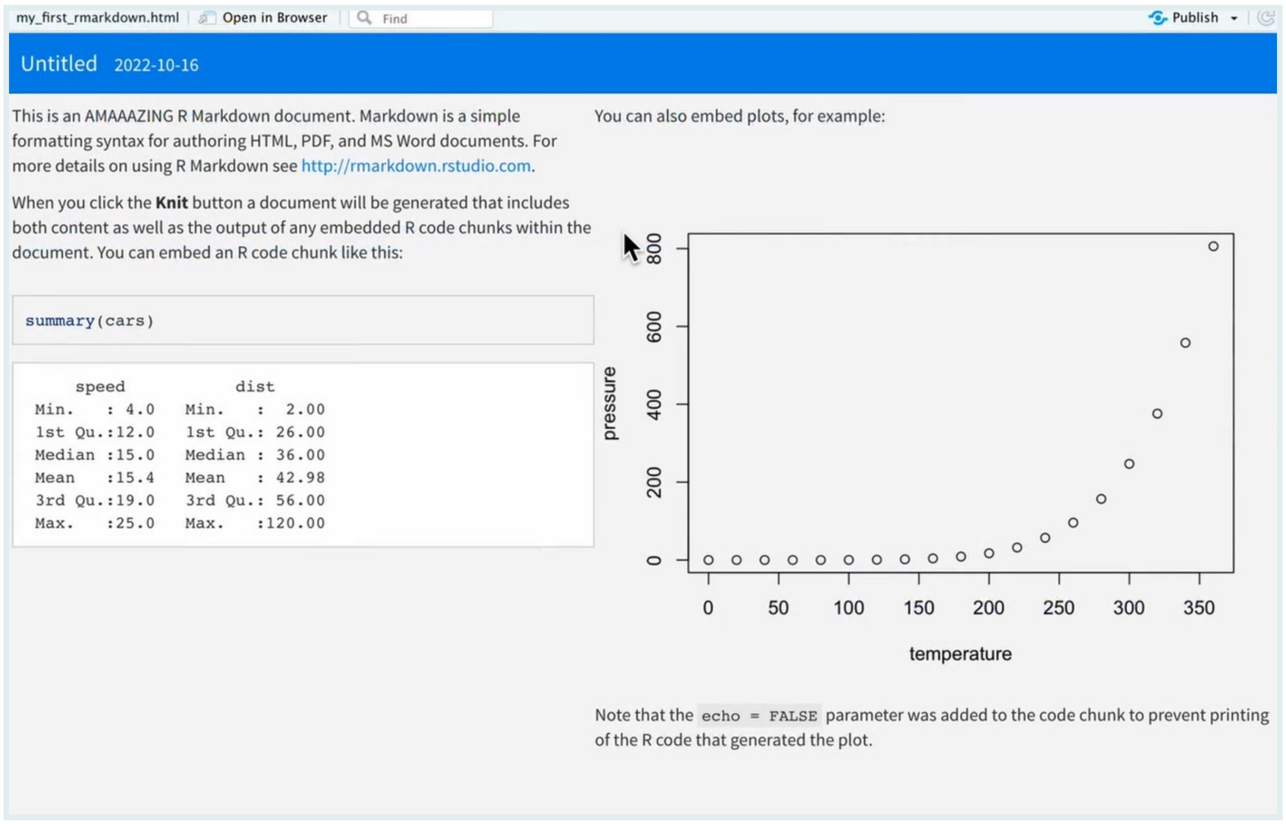
This is an AMAAAZING R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

Image of the r markdown document as a prettydoc

We can even get a simple dashboard format. First we need to `install.packages('flexdashboard')` . Then if we set the `output` to `flexdashboard::flex_dashboard`, and knit, we get something like the following:



This is an AMAAAZING R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
     speed           dist
 Min.   : 4.0   Min.   :  2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   : 42.98
 3rd Qu.:19.0   3rd Qu.: 56.00
 Max.   :25.0   Max.   :120.00
```

You can also embed plots, for example:

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Image of the r markdown document as a flexdashboard

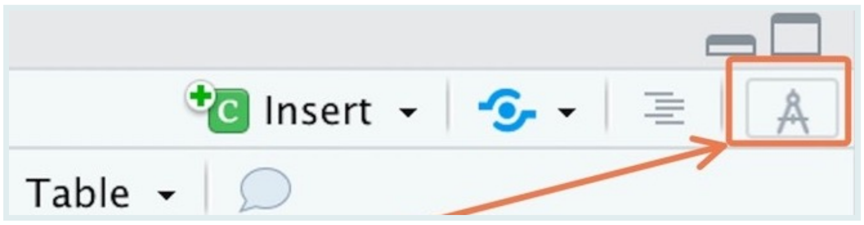
Note that it does not yet have tabs. To create tabs in a flexdashboard, change some of your double hashtags `##` to single hashtags `#`. This will change the header style for those sections, and get flexdashboard to render those headers as tabs instead.

Many other formats are possible, and we encourage you to explore on your own!

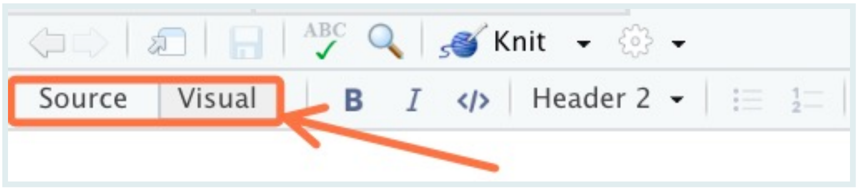
Visual vs Source mode

Rmarkdown documents can be edited in either a “Source” mode or a “Visual” mode.

You can switch into visual mode for a given document using the toolbars. For older RStudio versions, you may have an `A` button at the top-right of the document toolbar

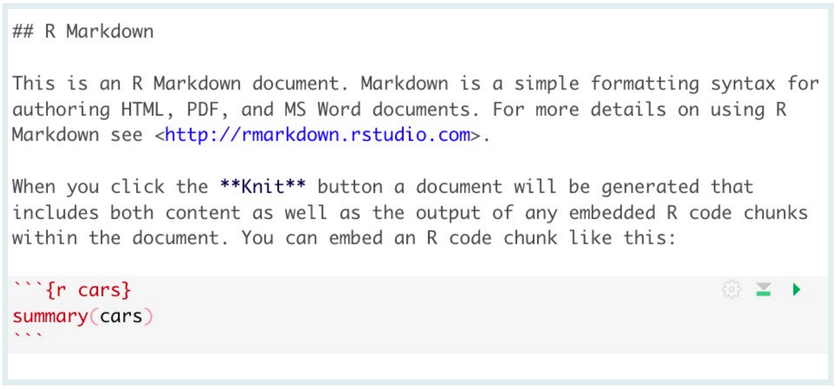


For newer RStudio versions, there is a pair of buttons to toggle between the modes:



What's the difference between these two modes?

In source mode you see the raw markdown syntax.

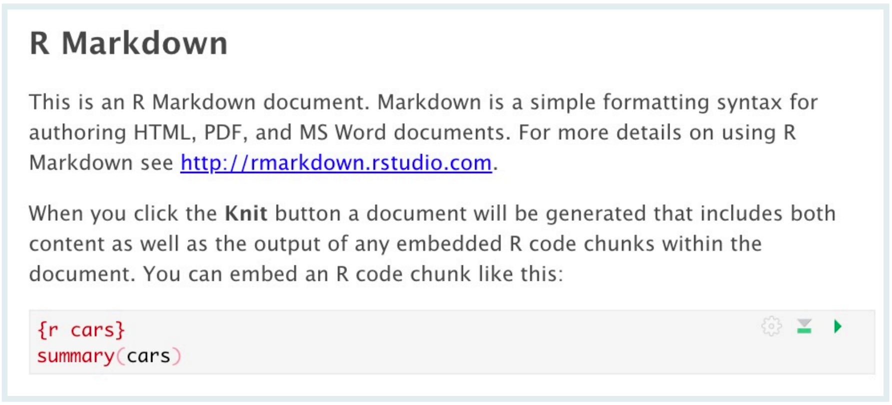


~~~
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for
authoring HTML, PDF, and MS Word documents. For more details on using R
Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that
includes both content as well as the output of any embedded R code chunks
within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```
~~~

VOCAB

Markdown is a simple set of conventions for adding formatting to plain text. For example, to italicize text, you wrap it in as asterisk `*text here*`, and to start a new header, you use the pound sign `#` . We will learn these in detail below

But in visual mode, you instead see a Microsoft-word like WYSIWIG view:



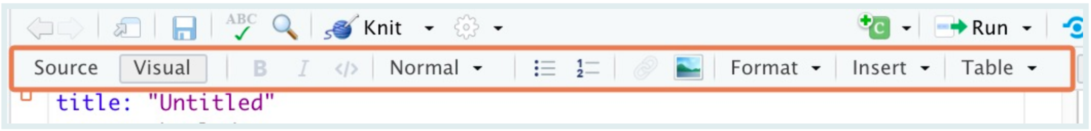
R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
{r cars}
summary(cars)
```

with a toolbar for easy formatting:



```
title: "Untitled"
```

That means you do not have remember the syntax for markdown elements. For example, if you want to make a section of text bold, you can simply highlight that piece of text and click on the bold button in the toolbar.

Now, while visual mode is much easier to use, we will teach you markdown syntax here for three reasons:

- Visual mode is sometimes a buggy experience, and to debug this you'll need to switch to source mode
- Understandin markdown syntax is useful outside of Rmarkdown
- Visual mode is not available in RStudio's collaborative mode, which you may make use of

Markdown syntax

In the "Help" tab, if you look up "Markdown Quick Reference", you will be able to find a wide variety of RMD options available to you.

You can define titles of different levels by starting a line with one or more `#`:

```
# Level 1 title
## Level 2 Title
### Level 3 Title
```

The body of the document consists of text that follows the *Markdown* syntax. A Markdown file is a text file that contains lightweight markup that helps set heading levels or format text. For example, the following text:

```
This is text with *italics* and **bold**.

You can define bulleted lists:

- first element
- second element
```

Will generate the following formatted text:

> This is text with *italics* and **bold**.
>
> You can define bulleted lists:
>
> - first element
> - second element

Note that you need spaces before and after lists, as well as keeping the listed items on separate lines, or else they will all crunch together rather than making a list.

We see that words placed between asterisks are italicized, lines that begin with a dash are transformed into a bulleted list, etc.

The Markdown syntax allows for other formatting, such as the ability to insert links or images. For example, the following code:

```
[Example Link](https://example.com)
```

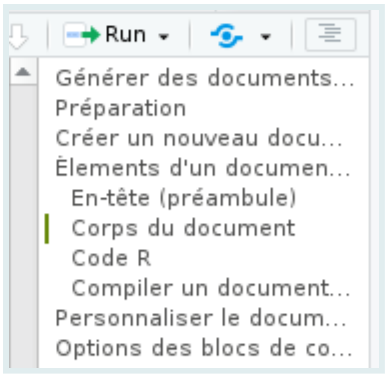
… will give the following link:

Example Link

We can also embed images. If you're in *Source* mode, type:

`[what you want the subtitle to say](images/picture_name.jpg)`, replacing "what you want the subtitle to say" (it can also be blank), "images" with the name of the image folder in your project, and "picture_name.jpg" with the name of the image you want to use. Of course, it is easier to do in *Visual* mode. From here, you can just open the folder that holds your image on your computer and drag-and-drop the image from the folder onto the page you're building. Or you can place the cursor where you want the image, click the button above marked with a "picture" icon, follow the prompts, and insert your image where the cursor is. Note that this will also create an "images" folder in your project (if it doesn't already exist) and put the image file into the "images" folder.

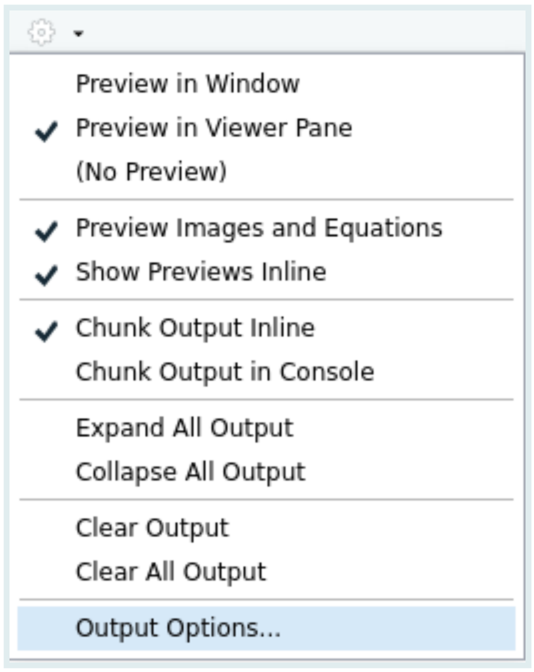
When titles have been defined, if you click on the *Show document outline* icon completely to the right of the toolbar associated with the R Markdown file, a table of contents automatically generated from the titles is displayed and allows you to navigate easily in the document:



Dynamic TOC

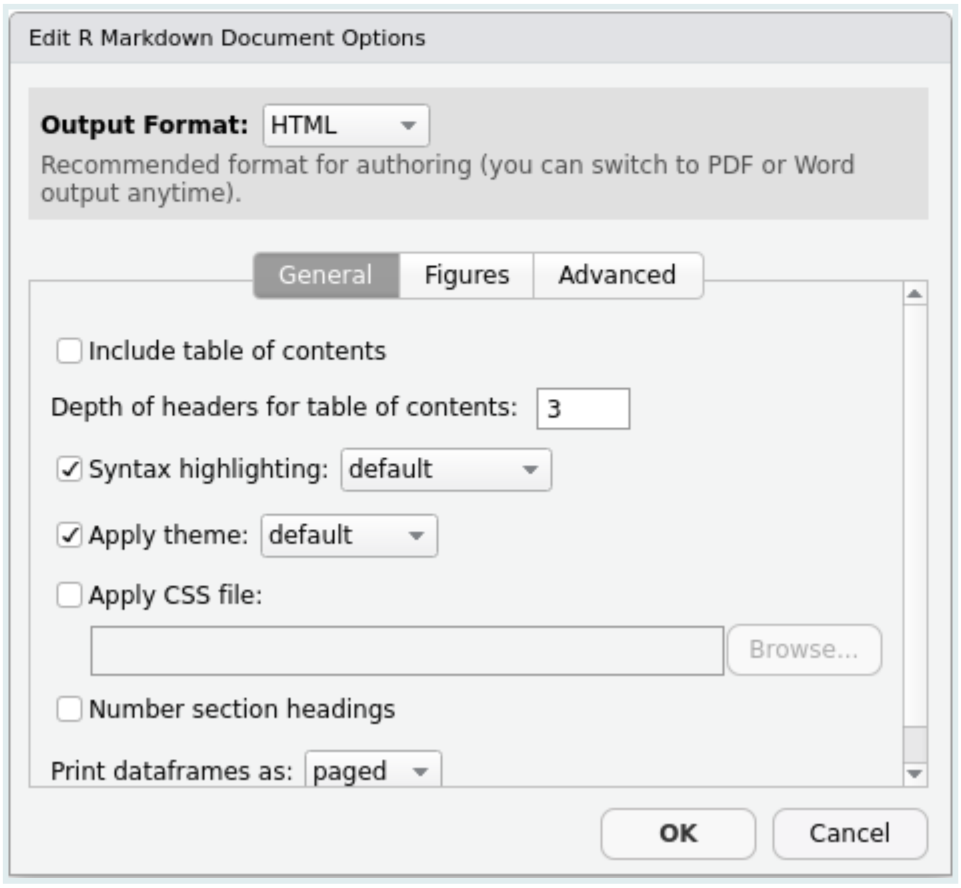
Customizing the generated document

The customization of the generated document is done by modifying options in the preamble of the document. However, RStudio offers a small graphical interface to change these options more easily. To do this, click on the gear icon to the right of the *Knit* button and choose *Output Options…*



R Markdown Output Options

A dialog box appears allowing you to select the desired output format and, depending on the format, different options:


Edit R Markdown Document Options

Output Format: HTML

Recommended format for authoring (you can switch to PDF or Word output anytime).

☐ Include table of contents

Depth of headers for table of contents: 3

✓ Syntax highlighting: default

✓ Apply theme: default

☐ Apply CSS file:

☐ Number section headings

Print dataframes as: paged

R Markdown Output Options Dialog

For the HTML format for example, the *General* tab allows you to specify if you want a table of contents, its depth, the themes to apply for the document and the syntax highlighting of the R blocks, etc. The *Figures* tab allows you to change the default dimensions of the graphics generated.

When you change options, RStudio will actually change the preamble of your document. So if you choose to show a table of contents and change the syntax highlighting theme, your header will become something like:

```
---
title: "R Markdown Review"
output:
   html_document:
     highlight: kate
     knock: yes
---
```

You can modify the options directly by editing the preamble.

Note that it is possible to specify different options depending on the format, for example:

```
---
title: "R Markdown Review"
output:
  html_document:
    highlight: kate
    knock: yes
  pdf_document:
    fig_caption: yes
    highlight: kate
---
```

The complete list of possible options is present on the official documentation site (very complete and well done) and on the cheat sheet and the reference guide, accessible from RStudio via the *Help* menu then *Cheatsheets*.

R code chunks

In addition to free text in Markdown format, an R Markdown document contains, as its name suggests, R code. This is included in blocks (*chunks*) written the following way in *Source* mode:

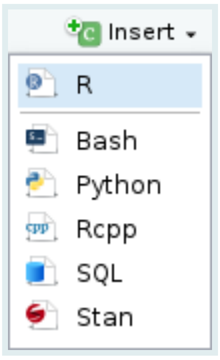
```{r}
r_code <- 2+2
```

Which will produce the following in *Visual* mode:

```
r_code <- 2+2
```

As this sequence of characters is not very easy to enter, you can use the *Insert* menu of RStudio and choose *R*[^3], or use the keyboard shortcut `Command+Option+i` on Mac or `Ctrl+Alt+i` on Windows.

Note that it is possible to use other languages in code chunks.



Code block insertion menu

In RStudio blocks of R code are usually displayed with a slightly different background color to distinguish them from the rest of the document.

When your cursor is in a block, you can enter the R code you want and execute it with Command + Enter. You can also execute all the code contained in a block by clicking on the green "play" button at the top right of the code chunk.

Chunk output inline vs in condole

In RStudio, by default, the results of a block of code (text, table or graphic) are displayed directly *in* the document editing window, allowing them to be easily viewed and kept for the duration of the session.

This behavior can be changed by clicking the gear icon on the toolbar and choosing *Chunk Output in Console.*

R code chunk options

It is also possible to pass options to each block of R code to modify its behavior.

Remember that a block of code looks like this:

~~~
```{r}
x <- 1:5
~~~

The options of a code block are to be placed inside the braces `{r}`, with a comma separating each option.

Block name

The first possibility is to give a *name* to the block. This is indicated directly after the `r`:

`{r block_name}`

It is not mandatory to name a block, but it can be useful in the event of a compilation error, to identify the block that caused the problem. Be careful, you cannot have two blocks with the same name.

Options

In addition to a name, a block can be passed a series of options in the form `option=value`. Here is an example of a block with a name and options:

~~~
```{r blockName, echo = FALSE, warning = TRUE}
x <- 1:5
~~~

And an example of an unnamed block with options:

~~~
```{r echo = FALSE, warning = FALSE}
x <- 1:5
~~~

One of the useful options is the `echo` option. By default `echo` is `TRUE`, and the block of R code is inserted into the generated document, like this:

```
x <- 1:5
print(x)
```

```
## [1] 1 2 3 4 5
```

But if we set the `echo=FALSE` option, then the R code is no longer inserted into the document, and only the result is visible:

```
## [1] 1 2 3 4 5
```

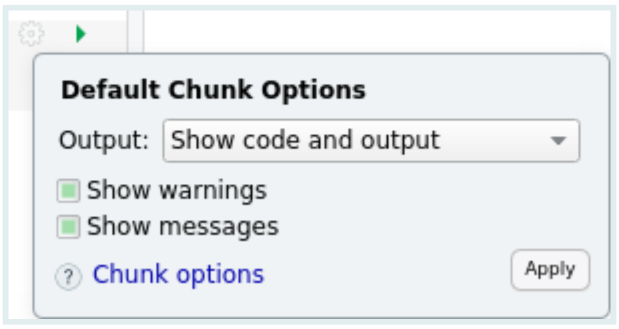
Here is a list of some of the available options:

Option	Values	Description
echo	TRUE/FALSE	Show (or hide) this R code chunk in the resulting knitted document
eval	TRUE/FALSE	Run (or not) the code in this code chunk in the resulting knitted document
include	TRUE/FALSE	Combines the options "echo and eval"; either show and run, or hide and don't run
message	TRUE/FALSE	Show (or hide) any system messages generated by running this code chunk in the resulting knitted document
warning	TRUE/FALSE	Show (or hide) any warnings generated by running this code chunk in the resulting knitted document

There are many other options described in particular in R Markdown reference guide{target = "_blank"} (PDF in English).

Change options

It is possible to modify the options manually by editing the header of the code block, but you can also use a small graphical interface offered by RStudio. To do this, simply click on the gear icon located to the right of the header line of each block:



Code Block Options Menu

You can then modify the most common options, and click on *Apply* to apply them.

Global Options

You may want to apply an option to all the blocks in a document. For example, one may wish by default not to display the R code of each block in the final document.

You can set an option globally using the `knitr::opts_chunk$set()` function. For example, inserting `knitr::opts_chunk$set(echo = FALSE)` into a code block will set the `echo = FALSE` option to default for all subsequent blocks.

In general, we place all these global modifications in a special block called `setup` and which is the first block of the document:

~~~
```{r, include=FALSE}
knitr::opts_chunk$set(echo = FALSE)
~~~

Inline Code

It is also possible to write code chunks embedded in the text. If you go to *Source* mode and type

"The sum of a pair of 2s is ` r 2+2 `"

and then knit the RMD, the resulting document will evaluate the r code between the backticks. Note that you have to include the "r" at the beginning of your inline code chunk to get it to recognize it as R code.

You could also pass variables around your document just like in a regular R program. For example, on one line you could run,

``` {r} max_height <- max(women$height) ```

"The maximum height in the women data set is ` r max_height ` ."

The advantages of such a system are numerous:

- a single document can show your entire analysis workflow, since the code, results and text explanations are included
- the document can be very easily regenerated and updated, for example if the source data has been modified.
- the variety of output formats (HTML, PDF, Word, slides, dashboards, etc.) makes it easy to present your work to others.

Display tables

There are a number of ways for R Markdown Documents to show data tables. To start, you can see how our RMD displays a table with no formatting:

```
women
```

```
##    height weight
## 1      58    115
## 2      59    117
## 3      60    120
## 4      61    123
## 5      62    126
## 6      63    129
## 7      64    132
## 8      65    135
## 9      66    139
## 10     67    142
## 11     68    146
## 12     69    150
## 13     70    154
## 14     71    159
## 15     72    164
```

It looks pretty basic. Next, to follow along you'll want to load the following packages:

```
pacman::p_load(flextable, gt, reactable)
```

Flextable is better for showing simple tables supported by many formats. GT is better for showing complex tables in HTML documents. Reactable is better for showing very large tables in HTML by giving your audience the option to scroll through the tables.

```
"This is a flextable"
```

```
## [1] "This is a flextable"
```

```
flextable::flextable(women)
```

height	weight
58	115
59	117
60	120
61	123
62	126
63	129
64	132
65	135
66	139
67	142
68	146
69	150
70	154
71	159
72	164

```
"This is a GT table"
```

```
## [1] "This is a GT table"
```

```
gt::gt(women)
```

height	weight
58	115
59	117
60	120
61	123
62	126
63	129
64	132
65	135
66	139
67	142
68	146
69	150
70	154
71	159
72	164

```
"This is a reactable"
```

```
## [1] "This is a reactable"
```

```
reactable::reactable(women)
```

You can see many other types of table formats people have created at https://www.rstudio.com/blog/rstudio-table-contest-2022/

Document Templates

We have seen here the production of "classic" documents, but R Markdown allows you to create many other things.

The extension's documentation site offers a gallery of the different possible outputs. You can create slides, websites or even entire books, like this document.

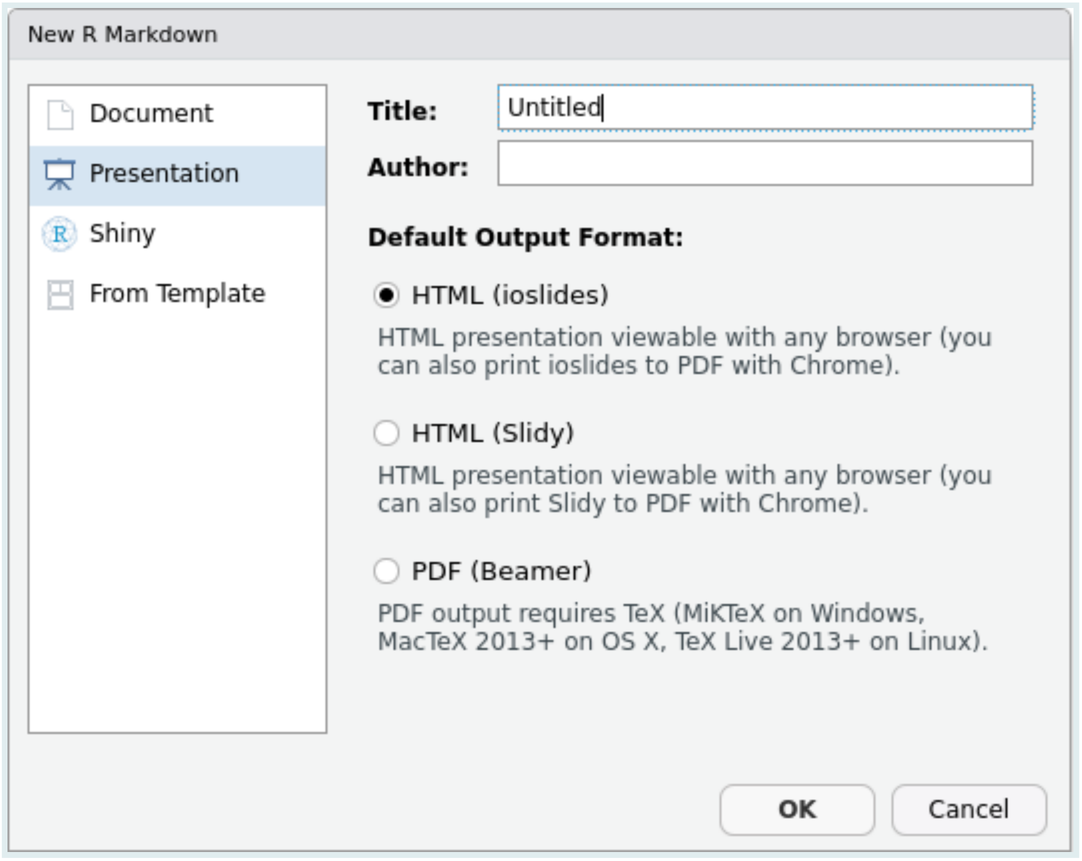
Slides

An interesting use is the creation of slideshows for presentations in the form of slides. The principle remains the same: we mix text in Markdown format and R code, and R Markdown transforms everything into presentations in HTML or PDF format. In general, the different slides are separated at certain heading levels.

Some slide templates are included with R Markdown, including:

- `ioslides` and `Slidy` for HTML presentations
- `beamer` for PDF presentations via `LaTeX`

When you create a new document in RStudio, these templates are accessible via the *Presentation* entry:



New R Markdown

Title: Untitled
Author:

Default Output Format:

(●) HTML (ioslides)
HTML presentation viewable with any browser (you can also print ioslides to PDF with Chrome).

( ) HTML (Slidy)
HTML presentation viewable with any browser (you can also print Slidy to PDF with Chrome).

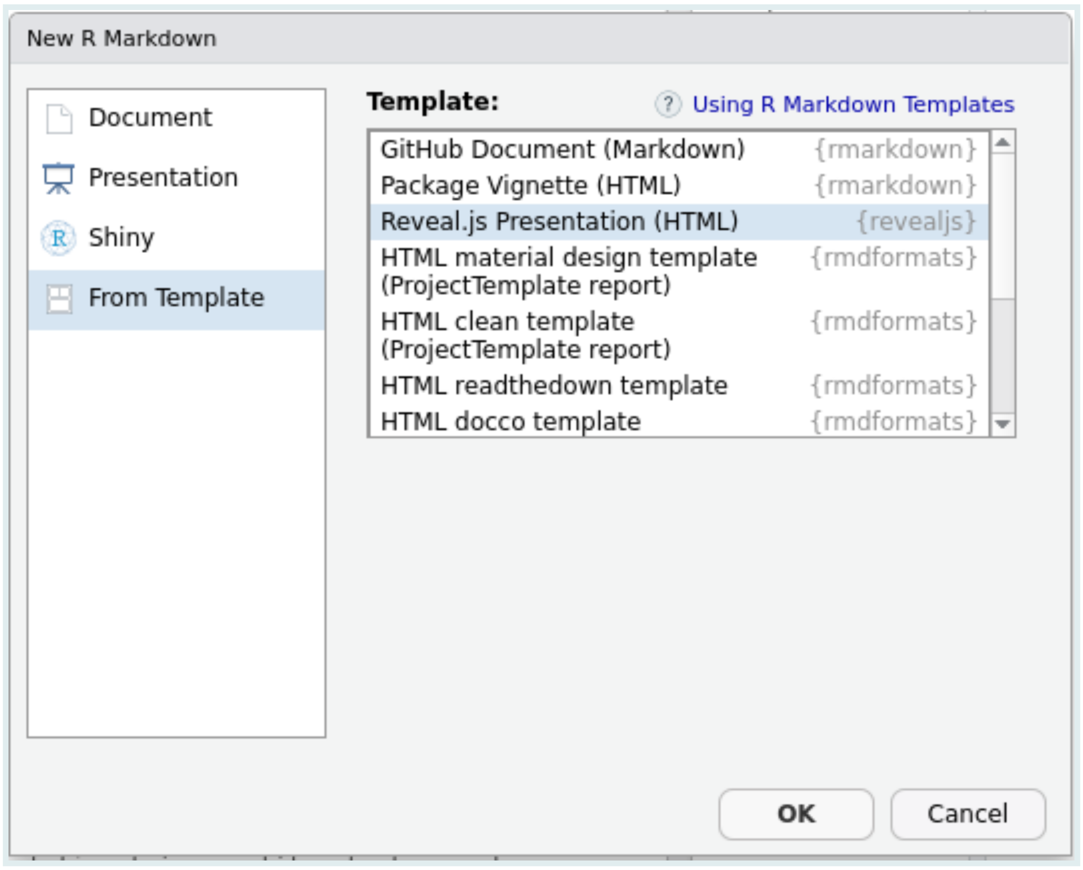
( ) PDF (Beamer)
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

Create an R Markdown presentation

Other extensions, which must be installed separately, also allow slideshows in various formats. These include in particular:

- xaringan for HTML presentations based on remark.js
- revealjs for HTML presentations based on reveal.js
- rmdshower for HTML slideshows based on shower

Once the extension is installed, it generally offers a starting *template* when creating a new document in RStudio. These are accessible from the *From Template* entry.



GitHub Document (Markdown) {rmarkdown}
Package Vignette (HTML) {rmarkdown}
Reveal.js Presentation (HTML) {revealjs}
HTML material design template (ProjectTemplate report) {rmdformats}
HTML clean template (ProjectTemplate report) {rmdformats}
HTML readthedown template {rmdformats}
HTML docco template {rmdformats}

Create a presentation from a template

Templates

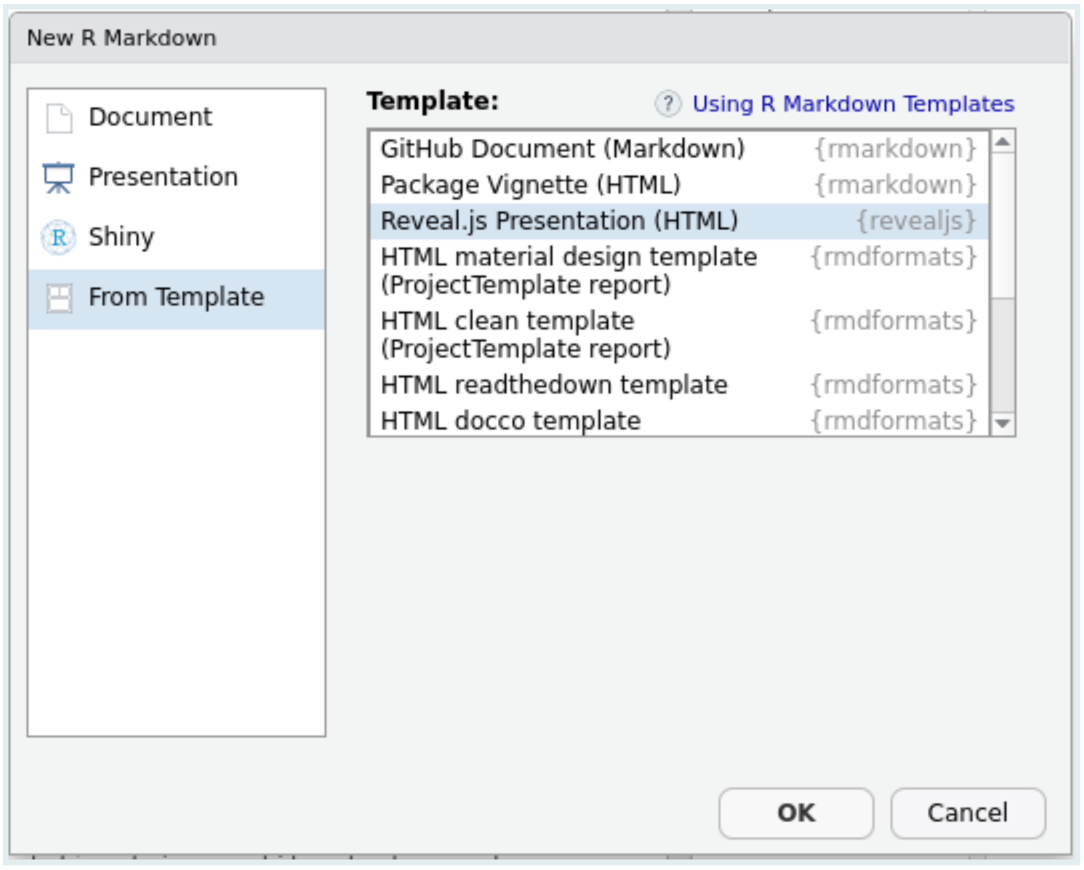
There are also different *templates* allowing you to change the format and presentation of the generated documents. A list of these formats and their associated documentation can be accessed from the formats documentation page.

Note in particular:

- the Distill format, suitable for scientific or technical publications on the Web
- the Tufte Handouts format which allows you to produce PDF or HTML documents in a format similar to that used by Edward Tufte for some of his publications
- rticles, package that offers LaTeX templates for several scientific journals

Finally, the rmdformats extension offers several HTML templates particularly suitable for long documents.

Again, most of the time, these document templates offer a starting *template* when creating a new document in RStudio (entry *From Template*):



GitHub Document (Markdown) {rmarkdown}
Package Vignette (HTML) {rmarkdown}
Reveal.js Presentation (HTML) {revealjs}
HTML material design template (ProjectTemplate report) {rmdformats}
HTML clean template (ProjectTemplate report) {rmdformats}
HTML readthedown template {rmdformats}
HTML docco template {rmdformats}

Create a document from a template

Resources

The following resources are all in English…

The book *R for data science*, available online, contains a chapter dedicated to R Markdown.

The extension's official site contains very complete documentation, both for beginners and for advanced users.

Finally, the RStudio help (*Help* menu then *Cheatsheets*) provides access to two summary documents: a synthetic "cheat sheet" (*R Markdown Cheat Sheet*) and a more complete "reference guide" (*R Markdown Reference Guide*).

Example analysis in Rmarkdown

Now we can put the tools we just used to work!

First, create a new R Markdown Project in RStudio.

Then open a web browser, go to https://bit.ly/view-ebola-data , and download the CSV. Here are the data you need.

Open your "downloads" folder, and copy or move the CSV to the "data" folder of your new project.

Create a new R Markdown Document in this project.

Open a web browser, go to https://tinyurl.com/ebola-script , highlight the code (under "ebola-script", starting with `1 # Ebola Sierra Leone analysis` and ending with `29 num_vars_plot`), and copy that text.

Paste the text into your new RMD under the `setup` chunk.

We're going to need to find the data, so paste "data/ebola_sierra_leone.csv" inside the readcsv function.

Next we'll try running through the code to make sure it works. Click the *Knit* button above.

It didn't work! One small difficulty with R Markdown is that it has a very limited vision, and only looks in its own folder. You need to tell it more explicitly where to look. See the new package in the `# Load packages` section, `here`? Here will help us change the frame of reference so we can access our data. Change our previous attempt to `readcsv` to the following:

```
ebola_sierra_leone <- readcsv(here("data/ebola_sierra_leone.csv"))
```

Now if you *Knit*, everything should run nicely. However, the document still looks disorganized, with visible code fragments and very basic displays. But we have the tools to change it!

Move the `# Load packages ----` line and all the package loading below it up into the setup code chunk. If you try *Knit*ting again you'll see a bunch of messages spitting out from all these load statements. We don't want everyone to see that, so add `, message=FALSE` to your setup code chunk.
It's also common to do all your data loading in one place, so move your `# Load data ----` and the `readcsv` line up into the setup code chunk as well.

Next let's break this document apart into various sections.

Cut the `# Cases by district —-` line, and paste it into the white space between the code chunks. We probably don't want to show the parts where we're adding data to the tabyl, so in this code chunk add `echo = false`.

Remember how we inserted code chunks earlier? We can use that same command to break code chunks apart, too. Break this code block apart into three individual chunks by using Command+Option+i or CTRL+Alt+i between each section. Pull the other two headers outside of the code.

Knit again and see what happens! You can play around with options here to see how they change the results.

As an example you can go back to the source document and use one of the new tables you learned about. Add `flextable,` inside the `p_load(` function, and try to display `district_tab` as a flextable.

Knit once more. You did it!

Lesson notes | Data structures

Created by the GRAPH Courses team

May 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Intro

In this lesson, we'll take a brief look at data structures in R. Understanding data structures is crucial for data manipulation and analysis. We will start by exploring vectors, the basic data structure in R. Then, we will learn how to combine vectors into data frames, the most common structure for organizing and analyzing data.

Learning objectives

1. You can create vectors with the `c()` function.
2. You can combine vectors into data frames.
3. You understand the difference between a tibble and a data frame.

Packages

Please load the packages needed for this lesson with the code below:

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse)
```

Introducing vectors

The most basic data structures in R are vectors. Vectors are a collection of values that all share the same class (e.g., all numeric or all character). It may be helpful to think of a vector as a column in an Excel spreadsheet.

Creating vectors

Vectors can be created using the `c()` function, with the components of the vector separated by commas. For example, the code `c(1, 2, 3)` defines a vector with the elements `1`, `2` and `3`.

In your script, define the following vectors:

```
age <- c(18, 25, 46)
sex <- c('M', 'F', 'F')
positive_test <- c(T, T, F)
id <- 1:3 # the colon creates a sequence of numbers
```

You can also check the classes of these vectors:

```
class(age)
```

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

```
class(sex)
```

```
## [1] "character"
```

```
class(positive_test)
```

```
## [1] "logical"
```

Each line of code below tries to define a vector with three elements but has a mistake. Fix the mistakes and perform the assignment.

```
my_vec_1 <- (1,2,3)
my_vec_2 <- c("Obi", "Chika" "Nonso")
```

VOCAB

The individual values within a vector are called *components* or elements. So the vector `c(1, 2, 3)` has three components/elements.

Manipulating vectors

Many of the functions and operations you have encountered so far in the course can be applied to vectors.

For example, we can multiply our `age` object by 2:

```
age
```

```
## [1] 18 25 46
```

```
age * 2
```

```
## [1] 36 50 92
```

Notice that every element in the vector was multiplied by 2.

Or, below we take the square root of `age`:

```
age
```

```
## [1] 18 25 46
```

```
sqrt(age)
```

```
## [1] 4.242641 5.000000 6.782330
```

You can also can add (numeric) vectors to each other:

```
age + id
```

```
## [1] 19 27 49
```

Note that the first element of `age` is added to the first element of `id` and the second element of `age` is added to the second element of `id` and so on.

From vectors to data frames

Now that we have a handle on creating vectors, let's move on to the most commonly used object in R: data frames. A data frame is just a collection of vectors of the same length with some helpful metadata. We can create one using the `data.frame()` function.

We previously created vector variables (id, age, sex and positive_test) for three individuals:

We can now use the `data.frame()` function to combine these into a single tabular structure:

```
data_epi <- data.frame(id, age, sex, positive_test)
data_epi
```

```
##   id age sex positive_test
## 1  1  18   M          TRUE
## 2  2  25   F          TRUE
## 3  3  46   F         FALSE
```

Note that instead of creating each vector separately, you can create your data frame defining each of the vectors inside the `data.frame()` function.

```
data_epi_2 <- data.frame(age = c(18, 25, 46),
                         sex = c('M', 'F', 'F'))

data_epi_2
```

```
##   age sex
## 1  18   M
## 2  25   F
## 3  46   F
```

SIDE NOTE Most of the time you work with data in R, you will be importing it from external contexts. But it is sometimes useful to create datasets *within* R itself. It is in such cases that the `data.frame()` function will come in handy.

To extract the vectors back out of the data frame, use the `$` syntax. Run the following lines of code in your console to observe this.

```
data_epi$age
is.vector(data_epi$age) # verify that this column is indeed a vector
class(data_epi$age) # check the class of the vector
```

Combine the vectors below into a data frame, with the following column names: "name" for the character vector, "number_of_children" for the numeric vector and "is_married" for the logical vector.

```
character_vec <- c("Bob", "Jane", "Joe")
numeric_vec <- c(1, 2, 3)
logical_vec <- c(T, F, F)
```

Use the `data.frame()` function to define a data frame in R that resembles the following table:

room	num_windows
dining	3
kitchen	2
bedroom	5

Tibbles

The default version of tabular data in R is called a data frame, but there is another representation of tabular data provided by the *tidyverse* package. It's called a `tibble`, and it is an improved version of the data frame.

You can convert from a data frame to a tibble with the `as_tibble()` function:

```
data_epi
```

```
##   id age sex positive_test
## 1  1  18   M          TRUE
## 2  2  25   F          TRUE
## 3  3  46   F         FALSE
```

```
tibble_epi <- as_tibble(data_epi)
tibble_epi
```

```
## # A tibble: 3 × 4
##      id   age sex   positive_test
##   <int> <dbl> <chr> <lgl>
## 1     1    18 M     TRUE
## 2      2     25  F      TRUE
## 3      3     46  F      FALSE
```

Notice that the tibble gives the data dimensions in the first line:

```
👉# A tibble: 3 × 4👈
     id   age sex   positive_test
  <int> <dbl> <chr> <lgl>
1      1    18  M      TRUE
2      2    25  F      TRUE
3      3    46  F      FALSE
```

And also tells you the data types, at the top of each column:

```
# A tibble: 3 × 4
     id   age sex   positive_test
👉  <int> <dbl> <chr> <lgl> 👈
1      1    18  M      TRUE
2      2    25  F      TRUE
3      3    46  F      FALSE
```

There, "int" stands for integer, dbl" stands for double (which is a kind of numeric class), "chr" stands for character, and "lgl" for logical.

The other benefit of tibbles is they avoid flooding your console when you print a long table.

Consider the console output of the lines below, for example:

```
# print the infert data frame (a built in R dataset)
infert # Veryyy long print
as_tibble(infert) # more manageable print
```

For your most of your data analysis needs, you should prefer tibbles over regular data frames.

`read_csv()` creates tibbles

When you import data with the `read_csv()` function from {readr}, you get a tibble:

```
ebola_tib <- read_csv("https://tinyurl.com/ebola-data-sample") # Needs
        internet to run
class(ebola_tib)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

But when you import data with the base `read.csv()` function, you get a data.frame:

```
ebola_df <- read.csv("https://tinyurl.com/ebola-data-sample") # Needs internet
         to run
class(ebola_df)
```

```
## [1] "data.frame"
```

Try printing `ebola_tib` and `ebola_df` to your console to observe the different printing behavior of tibbles and data frames.

This is one reason we recommend using `read_csv()` instead of `read.csv()`.

Wrap-up

With your understanding of data classes and structures, you are now well-equipped to perform data manipulation tasks in R. In the upcoming lessons, we will explore the powerful data transformation capabilities of the dplyr package, which will further enhance your data analysis skills.

Congratulations on making it this far! You have covered a lot and should be proud of yourself.

Solutions

Solution to the first r-practice block:

```
my_vec_1 <- c(1,2,3) # Use 'c' function to create a vector
my_vec_2 <- c("Obi", "Chika", "Nonso") # Separate each string with a comma
```

Solution to the second r-practice block:

```
df <- data.frame(name = character_vec,
                 number_of_children = numeric_vec,
                 is_married = logical_vec)
```

Solution to the third r-practice block:

```
# Solution to the third r-practice block
rooms <- data.frame(room = c("dining", "kitchen", "bedroom"),
                    num_windows = c(3, 2, 5))
```

Contributors

The following team members contributed to this lesson:

DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health researcher at Fiocruz, Brazil
Passionate about lots of things, especially when it involves people leading lives with more equality and freedom

EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education

LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Wickham, H., & Grolemund, G. (n.d.). *R for data science*. 15 Factors | R for Data Science. Accessed October 26, 2022. https://r4ds.had.co.nz/factors.html.