



Introduction to data analysis with R

With Examples from Public
Health and Epidemiology

Table of contents

Introduction	9
Contributors	9
Partners & Funders	9
1 Setting up R and RStudio	10
1.1 Introduction	10
1.2 Working locally vs. on the cloud	10
1.3 RStudio on the cloud	11
1.4 Set up on Windows	12
1.4.1 Download and install R	12
1.4.2 Download, install & run RStudio	13
1.5 Set up on macOS	15
1.5.1 Download and install R	15
1.5.2 Download, install & run RStudio	16
1.6 Wrap up	17
2 Setting up R and RStudio	19
2.1 Introduction	19
2.2 Working locally vs. on the cloud	19
2.3 RStudio on the cloud	20
2.4 Set up on Windows	21
2.4.1 Download and install R	21
2.4.2 Download, install & run RStudio	22
2.5 Set up on macOS	24
2.5.1 Download and install R	24
2.5.2 Download, install & run RStudio	25
2.6 Wrap up	26
3 Using RStudio	28
3.1 Learning objectives	28
3.2 Introduction	28
3.3 The RStudio panes	29
3.3.1 Source/Editor	31
3.3.2 Console	33
3.3.3 Environment	34
3.3.4 History	35
3.3.5 Files	35
3.3.6 Plots	36
3.3.7 Packages	37
3.3.8 Viewer	39
3.3.9 Help	39
3.4 RStudio options	39

3.5	Command palette	41
3.6	Wrapping up	41
3.7	Further resources	42
3.8	References	42
4	Coding basics	43
4.1	Introduction	43
4.2	Comments	44
4.3	R s a calculator	45
4.4	Formatting code	46
4.5	Objects in R	47
4.5.1	Create an object	47
4.5.2	What is an object?	48
4.5.3	Datasets are objects too	50
4.5.4	Rename an object	51
4.5.5	Overwrite an object	51
4.5.6	Working with objects	51
4.5.7	Some errors with objects	52
4.5.8	Naming objects	54
4.6	Functions	55
4.6.1	Basic function syntax	55
4.6.2	Nesting functions	58
4.7	Packages	59
4.7.1	A first example: the {tableone} package	59
4.7.2	Full signifiers	62
4.7.3	pacman::p_load()	63
4.8	Wrapping up	64
4.9	Answers	64
5	Data dive: Ebola in Sierra Leone	66
5.1	Introduction	66
5.2	Script setup	67
5.2.1	Header	67
5.2.2	Packages	68
5.3	Importing data into R	68
5.4	Intro to reproducibility	71
5.5	Quick data exploration	73
5.5.1	vis_dat()	75
5.5.2	inspect_cat() and inspect_num()	76
5.6	Analyzing a single numeric variable	78
5.6.1	Extract a column vector with \$	78
5.6.2	Basic operations on a numeric variable	79
5.6.3	Visualizing a numeric variable	80
5.7	Analyzing a single categorical variable	84
5.7.1	Frequency tables	85
5.7.2	Visualizing a categorical variable	86
5.8	Answering questions about the outbreak	87
5.9	Haven't had enough?	89
5.10	Wrapping up	89
6	RStudio projects	91
6.1	Learning objectives	91
6.2	Introduction	91

6.3	Creating a new RStudio Project	91
6.3.1	On RStudio Cloud	92
6.3.2	On a local computer	92
6.4	Creating Project subfolders	95
6.5	Adding a dataset to the “data” folder	96
6.5.1	On RStudio Cloud	96
6.5.2	On a local computer	96
6.6	Creating a script in the “scripts” folder	97
6.7	Importing data from the “data” folder	98
6.7.1	Using <code>here::here()</code>	101
6.8	Exporting data to the “outputs” folder	102
6.8.1	Overwriting data	104
6.9	Exporting plots to the “outputs” folder	105
6.10	Sharing a Project	107
6.11	Wrapping up	107
7	R Markdown	109
7.1	Introduction	109
7.2	Project setup	109
7.3	Create a new document	110
7.4	R Markdown Header (YAML)	111
7.4.1	Word Document	112
7.4.2	PowerPoint Document	112
7.4.3	PDF Document	112
7.4.4	Prettydoc	114
7.4.5	Flexdashboard	114
7.5	Visual vs Source mode	114
7.6	Markdown syntax	116
7.6.1	Customizing the generated document	117
7.7	R code chunks	119
7.7.1	Chunk output inline vs in condole	120
7.7.2	R code chunk options	120
7.7.3	Block name	120
7.7.4	Options	121
7.7.5	Change options	121
7.7.6	Global Options	122
7.8	Inline Code	122
7.9	Display tables	123
7.10	Document Templates	126
7.10.1	Slides	126
7.10.2	Templates	127
7.11	Resources	129
8	Data structures	130
8.1	Intros	130
8.2	Learning objectives	130
8.3	Packages	130
8.4	Introducing vectors	130
8.5	Creating vectors	130
8.6	Manipulating vectors	131
8.7	From vectors to data frames	132
8.8	Tibbles	133
8.8.1	<code>read_csv()</code> creates tibbles	135

8.9	Wrap-up	135
8.10	Solutions	135
9	Using ChatGPT for Data Analysis	137
9.1	Introduction	137
9.2	Models	137
9.3	1. Explain Unfamiliar Code	138
9.4	2. Debug Simple Errors	138
9.5	3. Add Code Comments	139
9.6	4. Reformat Code	139
9.7	5. Make Plots	140
9.8	6. Simple Data Wrangling	143
9.9	7. Translate Language	144
9.10	8. Translate Programming Languages	144
9.11	9. Fluid Find and Replace	145
9.12	Limitations of ChatGPT	147
10	Selecting and renaming columns	148
10.1	Introduction	148
10.2	Learning objectives	148
10.3	The Yaounde COVID-19 dataset	149
10.4	Introducing <code>select()</code>	150
10.4.1	Selecting column ranges with <code>:</code>	152
10.4.2	Excluding columns with <code>!</code>	152
10.5	Helper functions for <code>select()</code>	153
10.5.1	<code>starts_with()</code> and <code>ends_with()</code>	153
10.5.2	<code>contains()</code>	154
10.5.3	<code>everything()</code>	154
10.6	Change column names with <code>rename()</code>	156
10.6.1	Rename within <code>select()</code>	158
10.7	Wrap up	158
10.8	Solutions	159
11	Filtering rows	160
11.1	Intro	160
11.2	Learning objectives	160
11.3	The Yaounde COVID-19 dataset	161
11.4	Introducing <code>filter()</code>	161
11.5	Relational operators	162
11.6	Combining conditions with <code>&</code> and <code> </code>	164
11.7	Negating conditions with <code>!</code>	165
11.8	<code>NA</code> values	166
11.9	Wrap up	168
11.10	Solutions	169
12	Mutating columns	171
12.1	Intro	171
12.2	Learning objectives	171
12.3	Packages	171
12.4	Datasets	172
12.5	Introducing <code>mutate()</code>	173
12.6	Creating a Boolean variable	175
12.7	Creating a numeric variable based on a formula	177

12.8 Changing a variable's type	178
12.8.1 Integer: <code>as.integer</code>	178
12.9 Wrap up	179
12.10 Solutions	180
13 Conditional mutating	182
13.1 Introduction	182
13.2 Learning objectives	182
13.3 Packages	183
13.4 Datasets	183
13.5 Reminder: relational operators (comparators) in R	184
13.6 Introduction to <code>case_when()</code>	184
13.7 The <code>TRUE</code> default argument	187
13.8 Matching NA's with <code>is.na()</code>	188
13.9 Keeping default values of a variable	189
13.10 Multiple conditions on a single variable	192
13.11 Multiple conditions on multiple variables	194
13.12 Order of priority of conditions in <code>case_when()</code>	196
13.12.1 Overlapping conditions within <code>case_when()</code>	199
13.13 Binary conditions: <code>dplyr::if_else()</code>	202
13.14 Wrap up	204
13.15 Solutions	205
14 Grouping and summarizing data	208
14.1 Introduction	208
14.2 Learning objectives	208
14.3 The Yaounde COVID-19 dataset	208
14.4 What are summary statistics?	209
14.5 Introducing <code>dplyr::summarize()</code>	211
14.6 Grouped summaries with <code>dplyr::group_by()</code>	213
14.7 Grouping by multiple variables (nested grouping)	217
14.8 Ungrouping with <code>dplyr::ungroup()</code> (why and how)	219
14.9 Counting rows	222
14.9.1 Counting rows that meet a condition	224
14.9.2 <code>dplyr::count()</code>	227
14.10 Including missing combinations in summaries	230
14.11 Wrap up	236
14.12 Solutions	237
15 Grouped filter, mutate and arrange	241
15.1 Introduction	241
15.2 Learning objectives	241
15.3 Packages	241
15.4 Datasets	241
15.5 Arranging by group	243
<code>arrange()</code> can group automatically	244
15.6 Filtering by group	245
Filtering with nested groupings	246
15.7 Mutating by group	248
Mutating with nested groupings	250
15.8 Wrap up	252
15.9 Solutions	253

16 Pivoting data	255
16.1 Intro	255
16.2 Learning Objectives	255
16.3 Packages	255
16.4 What do wide and long mean?	255
16.5 When should you use wide vs long data?	258
16.6 Pivoting wide to long	259
16.7 Pivoting long to wide	261
16.8 Why is long data better for analysis?	263
16.8.1 Filtering grouped data	263
16.8.2 Summarizing grouped data	265
16.8.3 Plotting	266
16.9 Pivoting can be hard	268
16.10 Wrap up	268
16.11 Solutions	268
17 Advanced pivoting	270
17.1 Intro	270
17.2 Learning Objectives	270
17.3 Packages	270
17.4 Datasets	270
17.5 Wide to long	270
17.5.1 Understanding <code>names_sep</code> and <code>value</code>	273
17.5.2 Value type <i>before</i> the separator	277
17.5.3 A non-time-series example	279
17.5.4 Escaping the dot separator	280
17.5.5 What to do when you don't have a neat separator?	281
17.6 Long to wide	284
17.7 Wrap up	287
17.8 Solutions	287
18 Intro to ggplot2	291
18.1 Introduction	291
18.2 Learning objectives	291
18.3 Packages	292
18.4 Measles outbreaks in Niger	292
18.4.1 The <code>nigerm</code> dataset	293
18.4.2 The layered Grammar of Graphics	296
18.5 Working through the essential layers	299
18.5.1 Building a <code>ggplot()</code> in steps	300
18.6 Modifying the layers	306
18.6.1 Changing aesthetic mappings	306
18.6.2 Changing <code>geom_*</code> functions	307
18.6.3 Additional aesthetic mappings inside <code>aes()</code>	309
18.6.4 Fixed aesthetics outside <code>aes()</code>	313
18.7 Additional GG layers	317
18.8 Learning outcomes	318
18.9 Solutions	319
19 Scatter plots and smoothing lines	321
19.1 Introduction	321
19.2 Learning Objectives	321
19.3 Childhood diarrheal diseases in Mali	321

19.4 Scatter plots via <code>geom_point()</code>	323
19.5 Aesthetic modifications	324
19.5.1 Mapping data to aesthetics	324
19.5.2 Setting fixed aesthetics	327
19.6 Adding a trend line	334
19.7 Summary	339
19.8 Solutions	340
20 Lines, scales, and labels	342
20.1 Learning Objectives	342
20.2 Introduction	343
20.3 Packages	343
20.4 The <code>gapminder</code> data frame	343
20.5 Line graphs via <code>geom_line()</code>	346
20.5.1 Fixed aesthetics in <code>geom_line()</code>	347
20.6 Combining compatible geoms	349
20.7 Mapping data to multiple lines	351
20.8 Modifying continuous x & y scales	358
20.8.1 Scale breaks	360
20.8.2 Logarithmic scaling	363
20.9 Labeling with <code>labs()</code>	367
20.10 Preview: Themes	375
20.11 Wrap up	375
20.12 Solutions	376
21 Histograms with {ggplot2}	378
21.1 Histograms with {ggplot2}	378
21.2 Learning Objectives	378
21.3 Introduction	378
21.4 Packages	378
21.5 Childhood diarrheal diseases in Mali	378
21.6 Basic histograms with <code>geom_histogram()</code>	381
21.7 Adjusting bins in a histogram	383
21.7.1 Set the number of bins with <code>bins</code>	384
21.7.2 Set the width of bins with <code>binwidth</code>	386
21.7.3 Modify bin boundaries with <code>breaks</code>	389
21.8 Summary	389
21.9 Solutions	390
22 Boxplots with {ggplot2}	392
22.1 Boxplots with {ggplot2}	392
22.1.1 Learning Objectives	392
22.1.2 Introduction	392
22.1.3 Load packages	394
22.1.4 The <code>gapminder</code> dataset	394
22.1.5 Basic boxplots with <code>geom_boxplot()</code>	396
22.1.6 Reordering boxes with <code>reorder()</code>	401
22.1.7 Adding data points with <code>geom_jitter()</code>	405
22.1.8 Wrap up	409
22.1.9 Learning Outcomes	409
22.2 Solutions	410

Introduction

This book is a compilation of lesson notes for a 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.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit organization dedicated to making code and data skills accessible through affordable live bootcamps and free self-paced courses.

Contributors

We are extremely grateful to the following individuals who have contributed to the development of these materials over several years:

Amanda McKinley, Andree Valle Campos, Aziza Merzouki, Benedict Nguimbis, Bennour Hsin, Camille Beatrice Valera, Daniel Camara, Eduardo Araujo, Elton Mukonda, Guy Wafeu, Imad El Badisy, Imane Bensouda Korachi, Joy Vaz, Kene David Nwosu, Lameck Agasa, Laure Nguemo, Laure Vancauwenberghe, Matteo Franza, Michal Shrestha, Olivia Keiser, Sabina Rodriguez Velasquez, Sara Botero Mesa.

Partners & Funders

- University of Geneva
- University of Oxford
- World Health Organization
- Global Fund
- Ernst Goehner Foundation

Chapter 1

Setting up R and RStudio

Learning objective

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

1.1 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.

1.2 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!

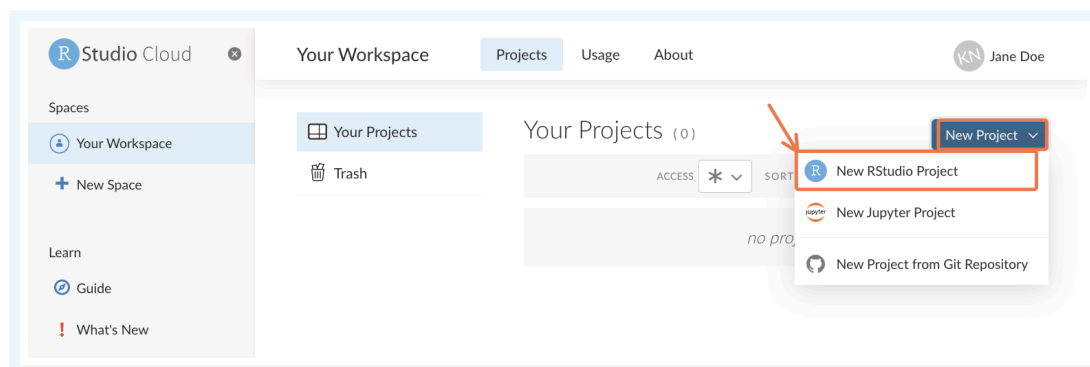
Watch Out

RStudio cloud will only give you 25 free project hours per month. After that, you will need to upgrade to a paid plan. If you think you'll need more than 25 hours per month, you may want to avoid this option.

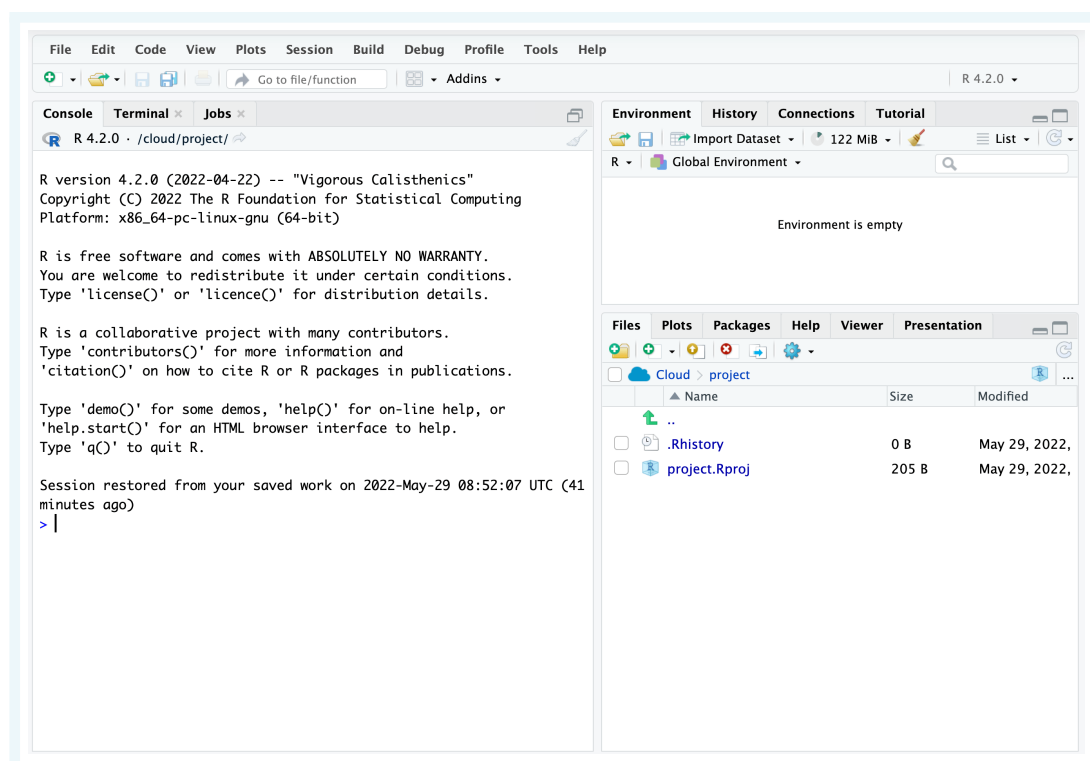
1.3 RStudio on the cloud

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".

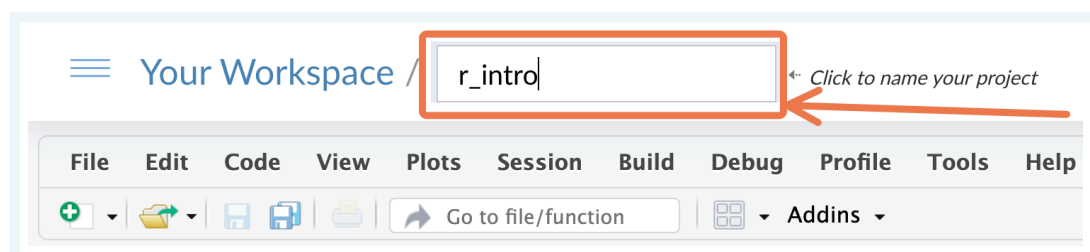


You should see a screen like this:

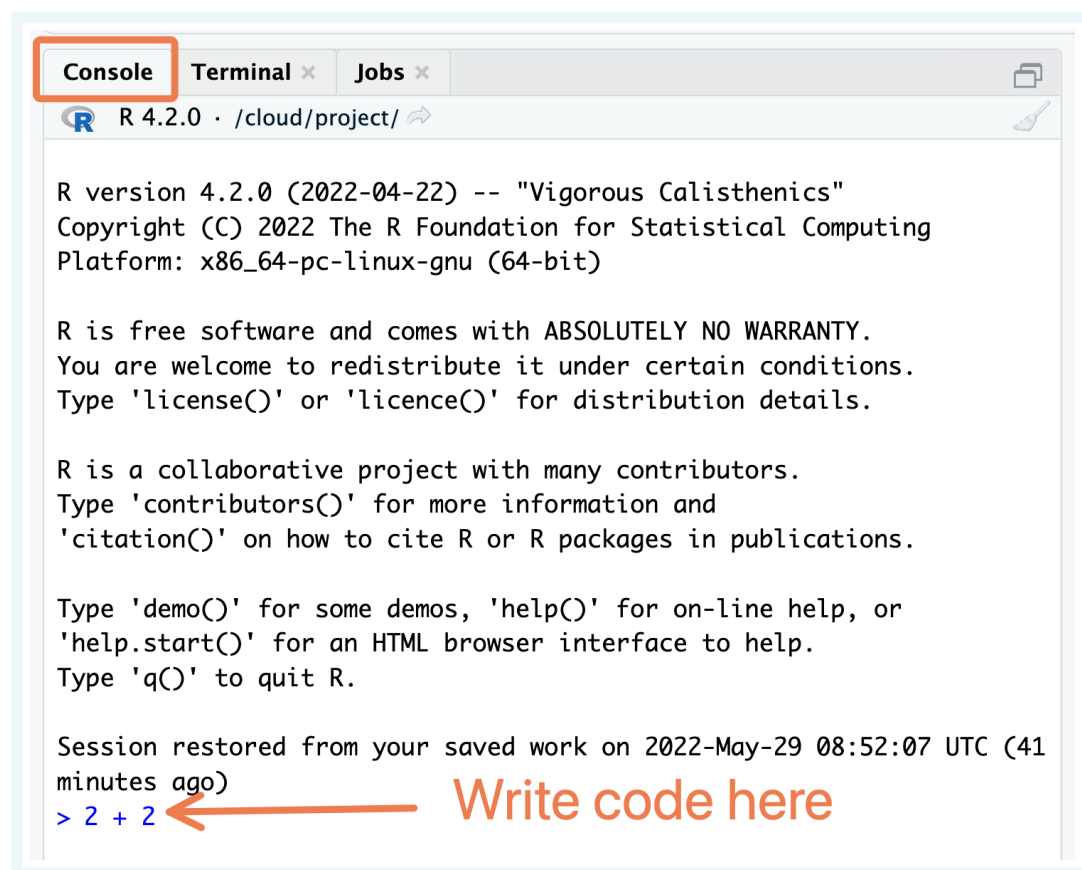


This is RStudio, your new home for a long time to come!

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



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,

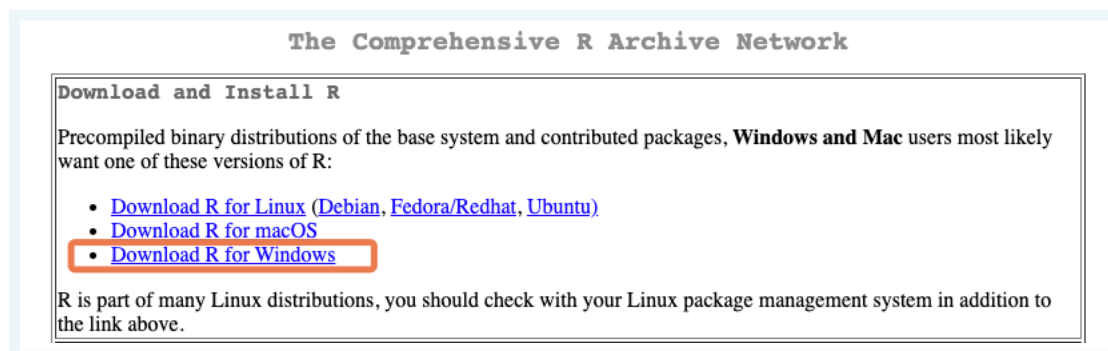
Proceed to the “wrapping up” section of the lesson.

1.4 Set up on Windows

1.4.1 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

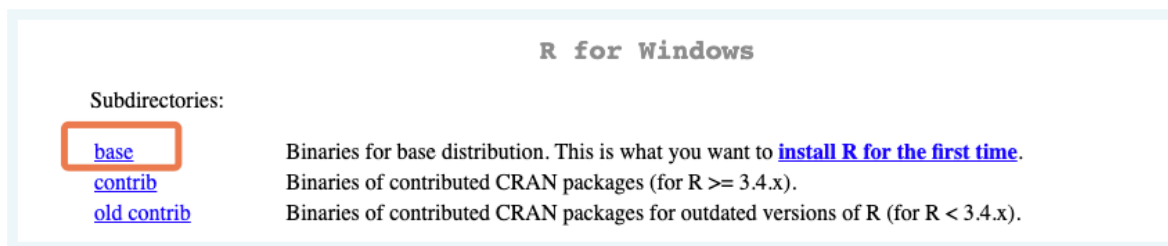
Download and Install R

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

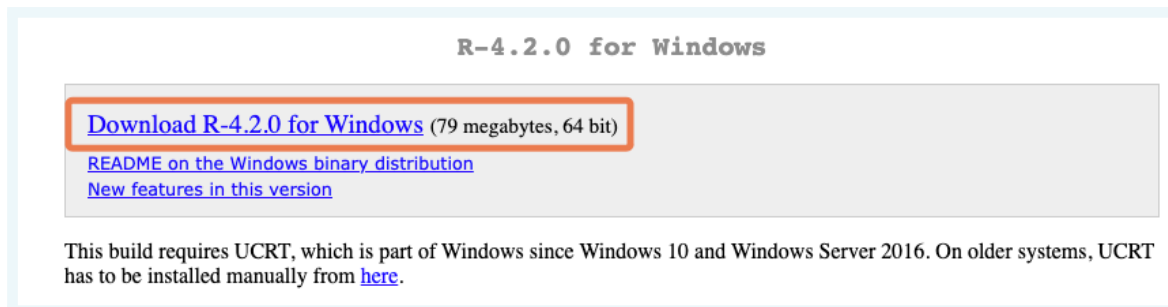
- [Download R for Linux \(Debian, Fedora/Redhat, Ubuntu\)](#)
- [Download R for macOS](#)
- [Download R for Windows](#)

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.



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



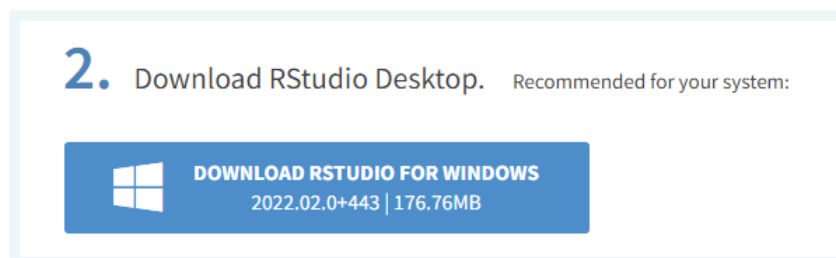
Note that the screenshot above may not show the latest version.

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.

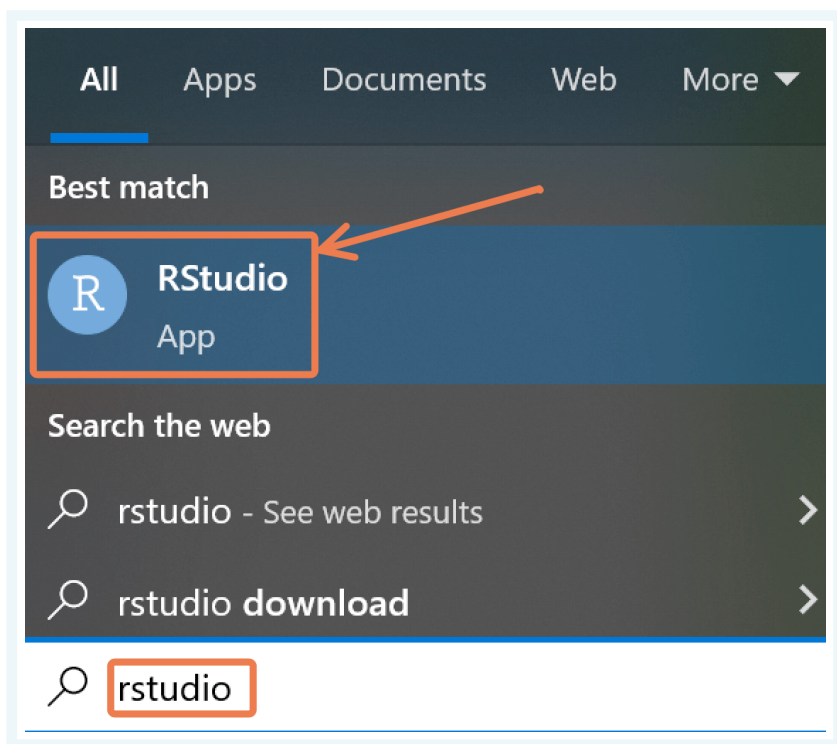
1.4.2 Download, install & run RStudio

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

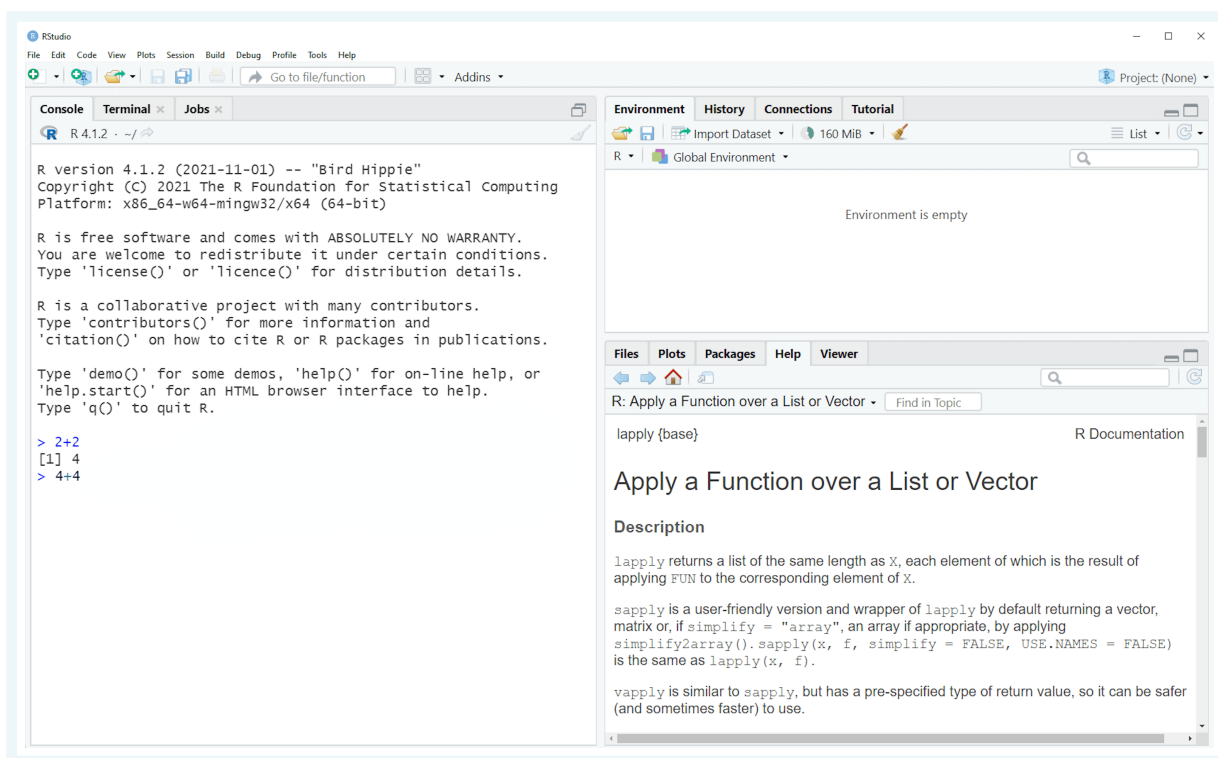


After the download is finished, click on the downloaded file and follow the installation instructions.

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:

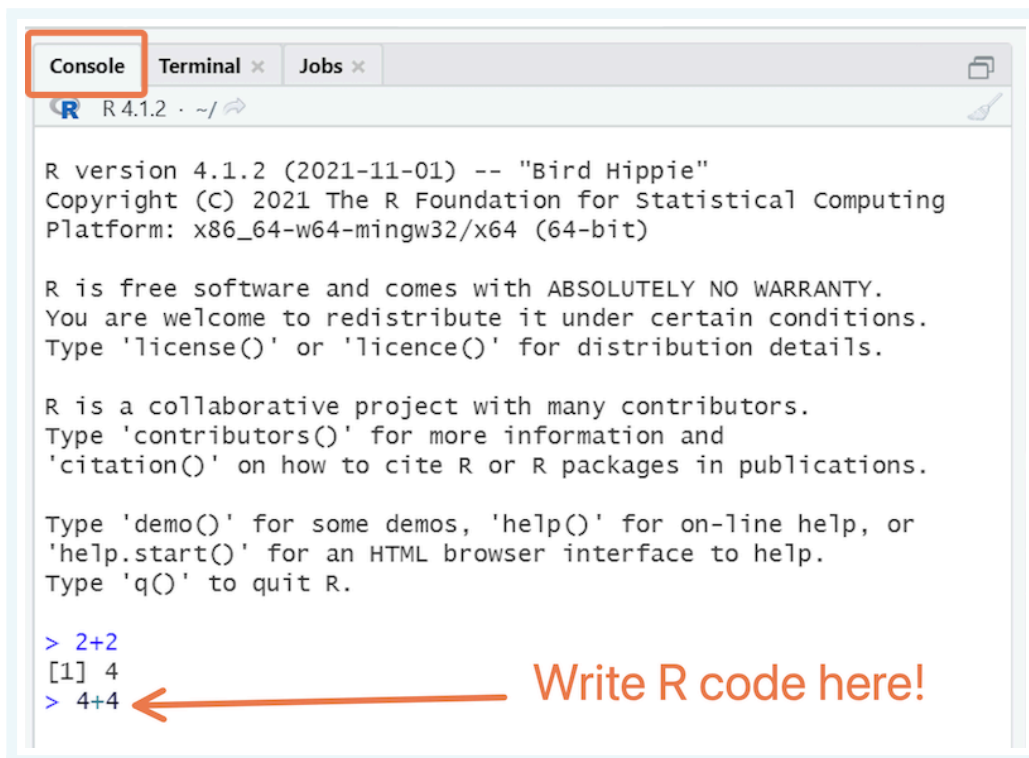


You should see a window like this:



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
```

Write R code here!

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

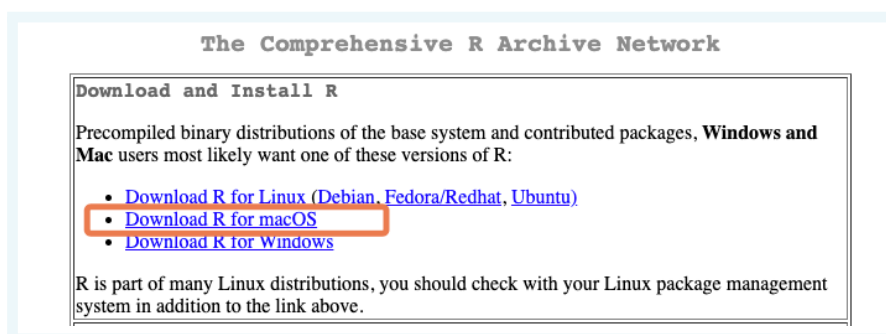
That's it; you're ready to roll. Proceed to the "wrapping up" section of the lesson.

1.5 Set up on macOS

1.5.1 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:



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: 2d9058826e447270494a0a09a0715638747
(ca. 90MB) for Intel Macs

Latest version for Intel Macs

R 4.2.0-arm64.pkg (notarized and signed)
SHA1 hash: 23b3c41167b97716486589a75657827923d4db2bc
(ca. 89MB) for M1 Macs only!

Latest version for M1 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 omitted when choosing "custom install", they are only needed if you want to use the `tecltk` R package or build package documentation from sources.

Note: the use of X11 (including `tecltk`) 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 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 `tecltk`) 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)

Mac-GUI-1.78.tar.gz
SHA1 hash: 23b3c41167b97716486589a75657827923d4db2bc

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.

R-3.6.3.pkg (signed)
SHA1 hash: c4632961f9b45d77801058d9a225d912363557c4
(ca. 77MB)

For older macs

Binaries for legacy OS X systems:

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 omitted when choosing "custom install", they are only needed if you want to use the `tecltk` R package or build package documentation from sources.


- 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.

1.5.2 Download, install & run RStudio

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

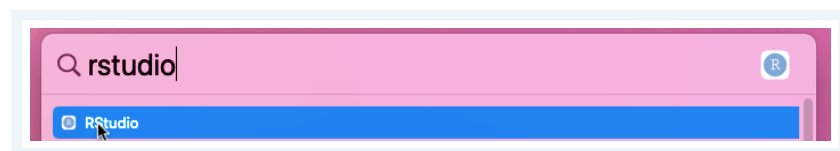
2. Download RStudio Desktop. Recommended for your system:



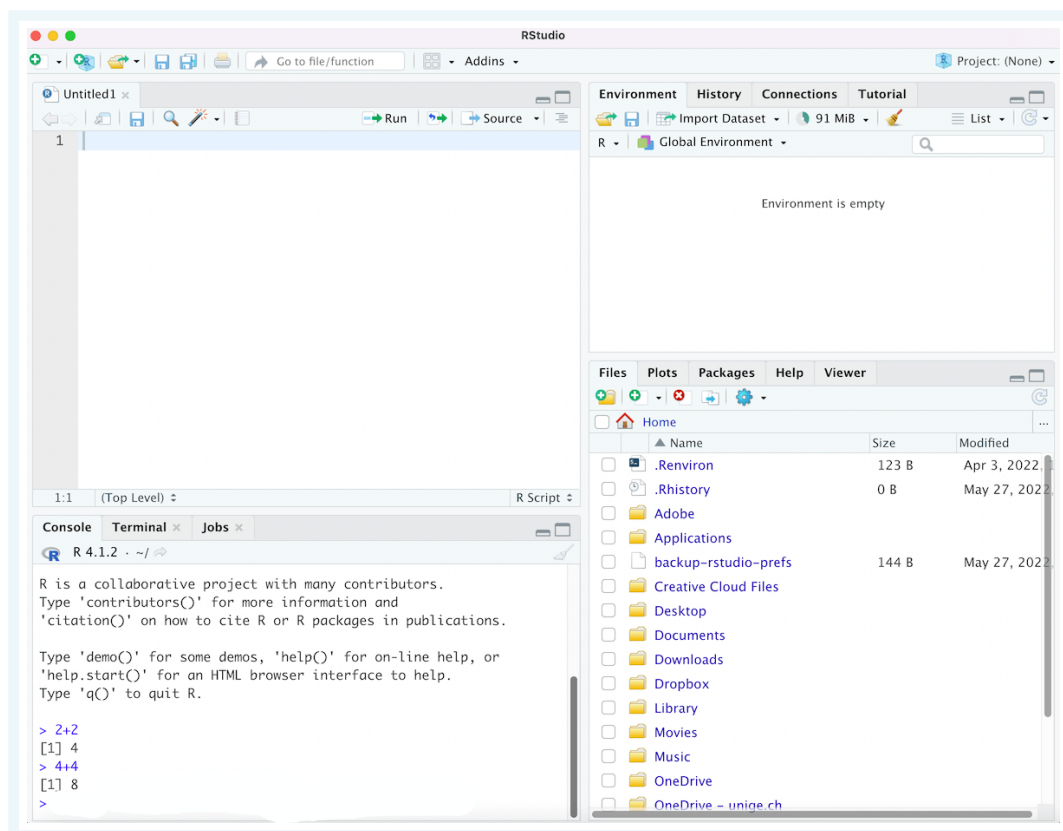
DOWNLOAD RSTUDIO FOR MAC
2022.02.0+443 | 217.18MB

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.

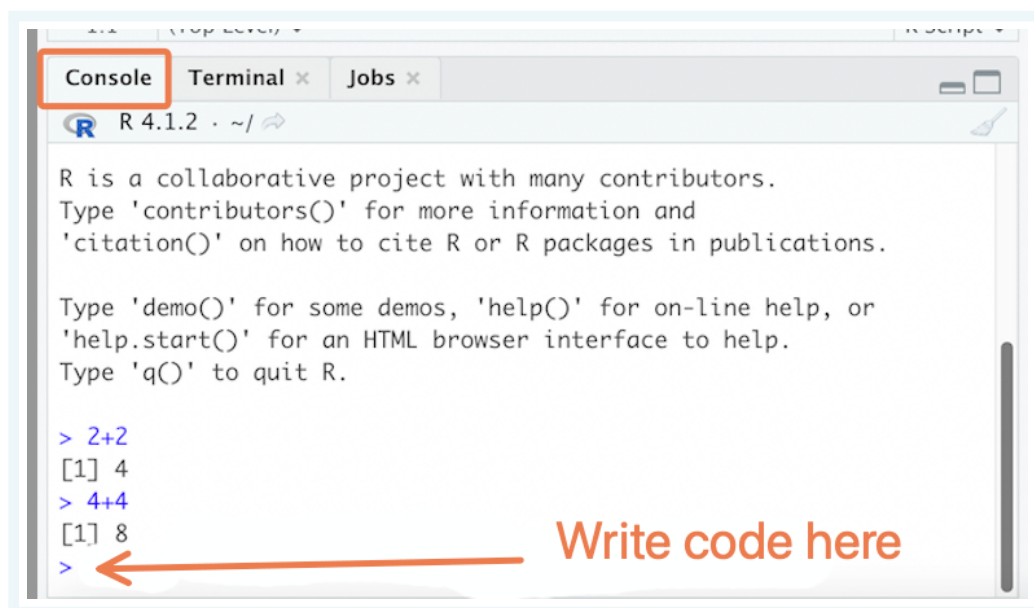


You should see a window like this:



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:



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

1.6 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!

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.

Chapter 2

Setting up R and RStudio

Learning objective

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

2.1 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.

2.2 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!

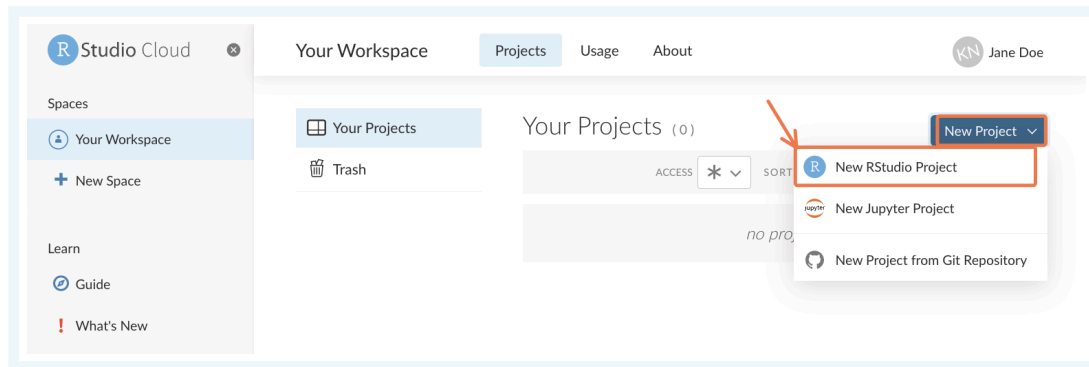
Watch Out

RStudio cloud will only give you 25 free project hours per month. After that, you will need to upgrade to a paid plan. If you think you'll need more than 25 hours per month, you may want to avoid this option.

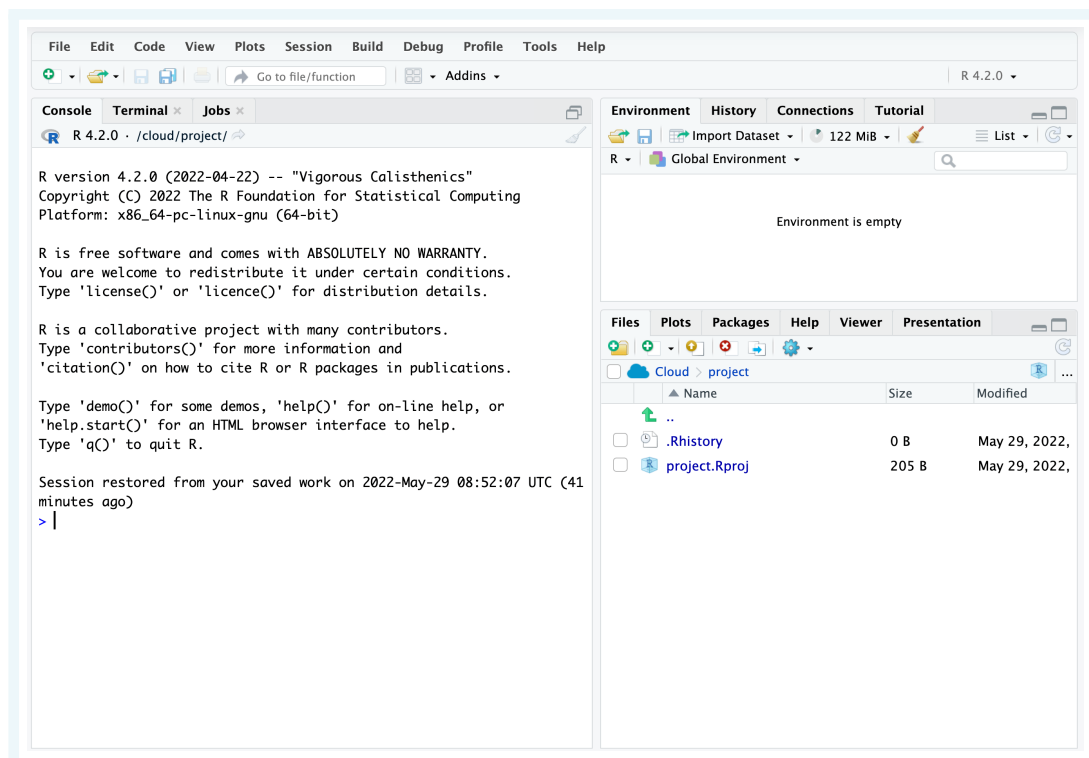
2.3 RStudio on the cloud

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".

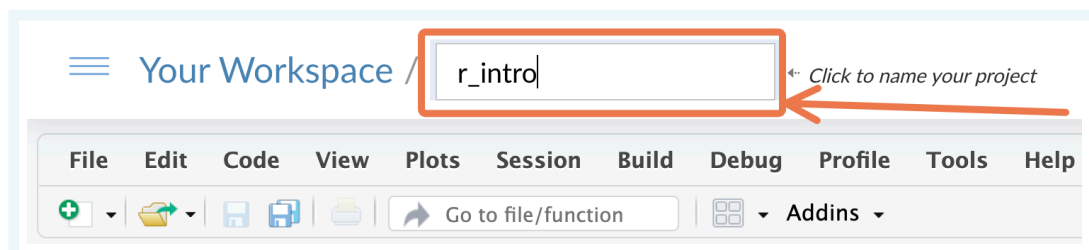


You should see a screen like this:

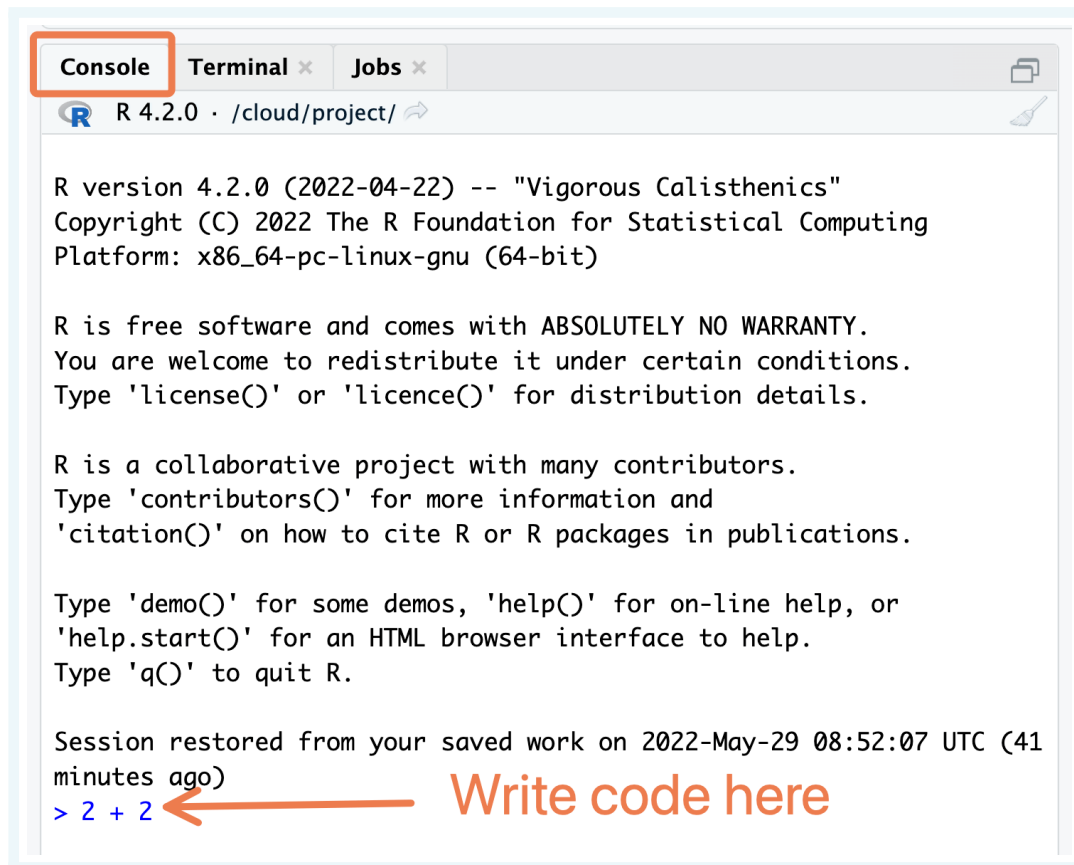


This is RStudio, your new home for a long time to come!

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



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



The screenshot shows the RStudio interface with the 'Console' pane selected. The console displays the R version 4.2.0 startup message, including copyright information and instructions on how to use R. At the bottom, the prompt '>' is followed by the code '2 + 2'. An orange arrow points to the prompt, and the text 'Write code here' is written in orange.

```
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,

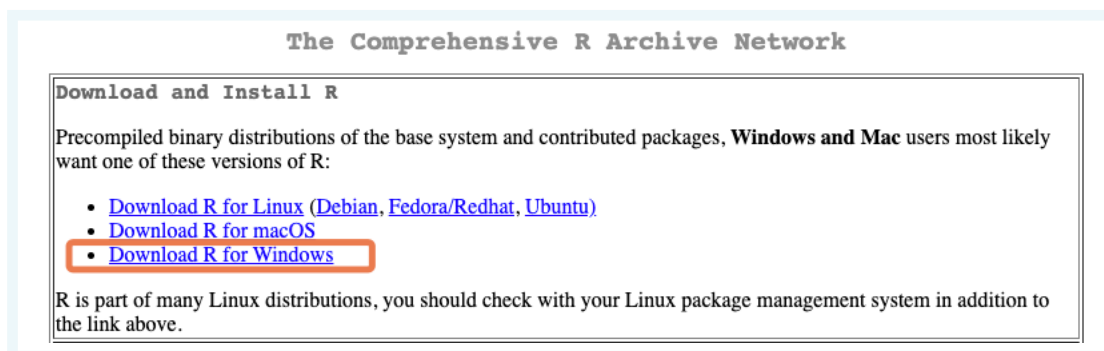
Proceed to the “wrapping up” section of the lesson.

2.4 Set up on Windows

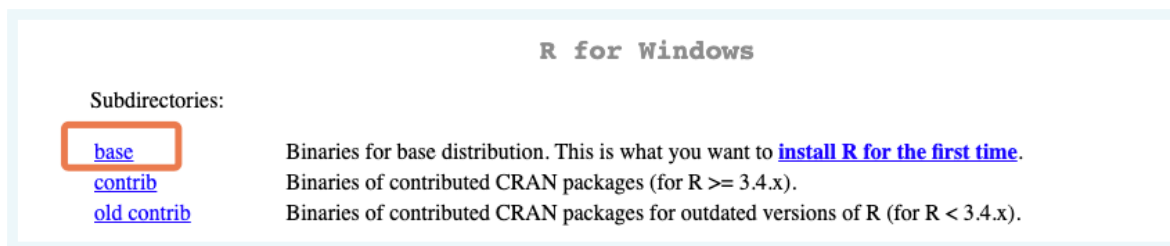
2.4.1 Download and install R

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

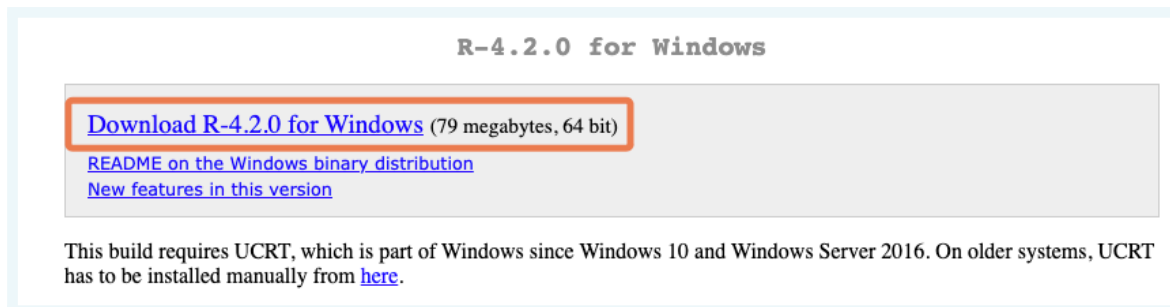
1. Go to cran.rstudio.com to access the R installation page. Then click the download link for Windows:



2. Choose the “base” sub-directory.



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



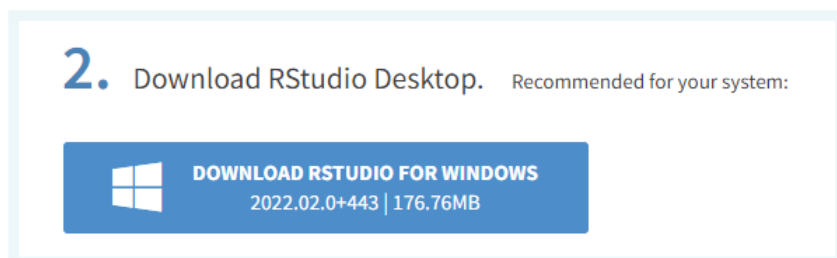
Note that the screenshot above may not show the latest version.

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.

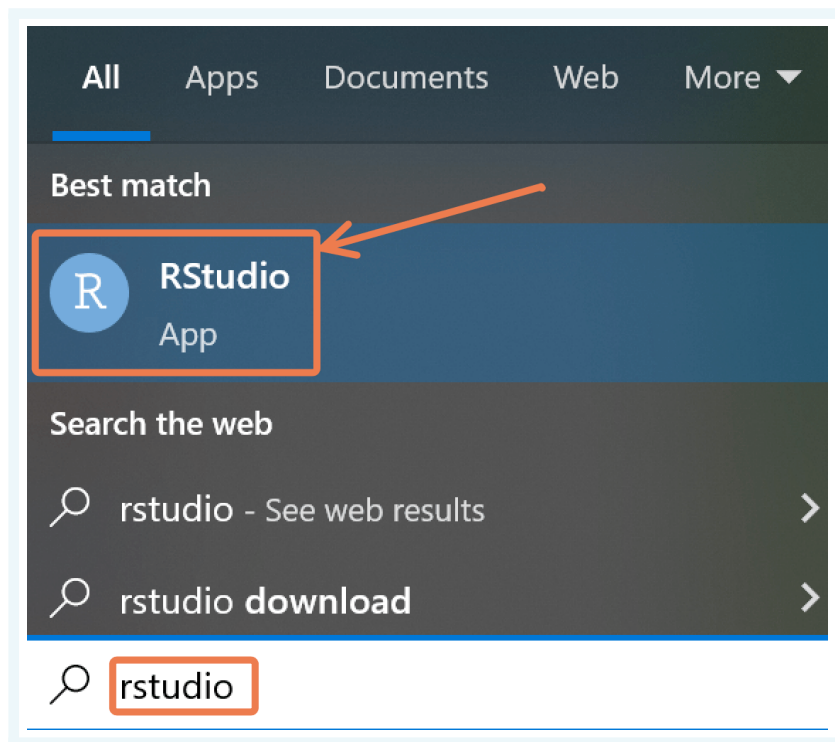
2.4.2 Download, install & run RStudio

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

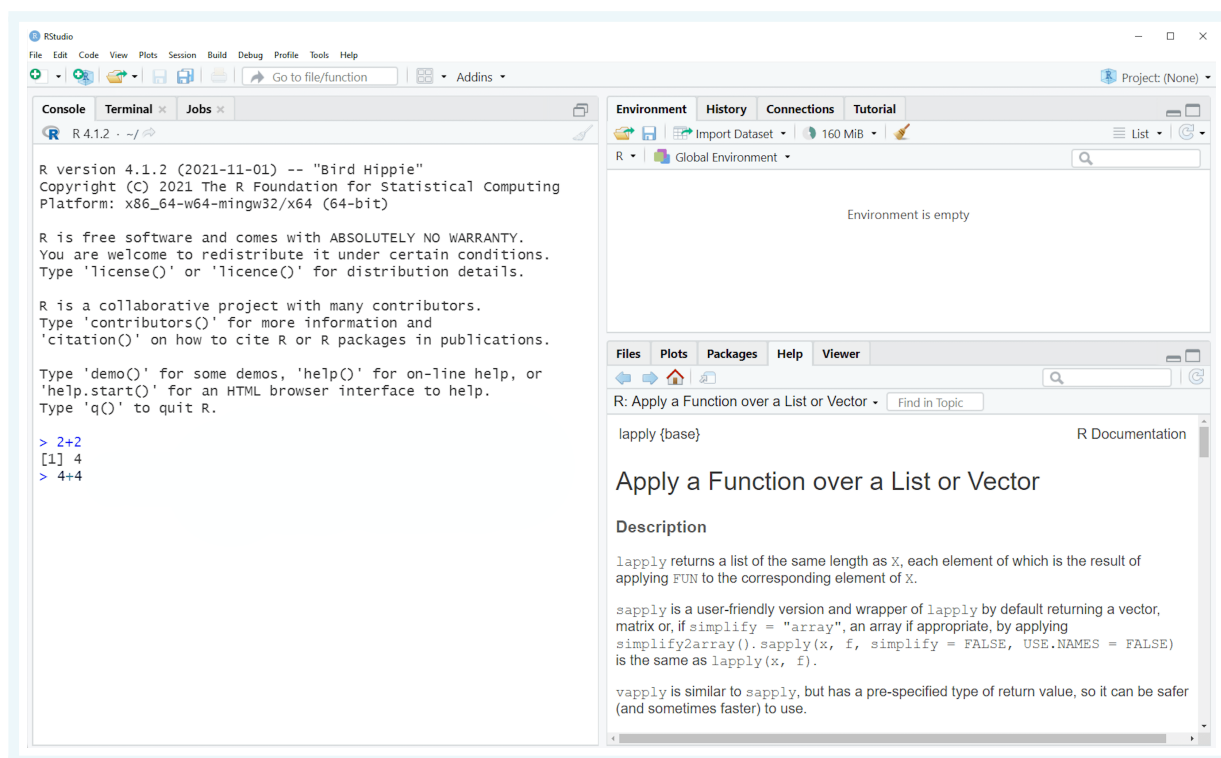


After the download is finished, click on the downloaded file and follow the installation instructions.

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:

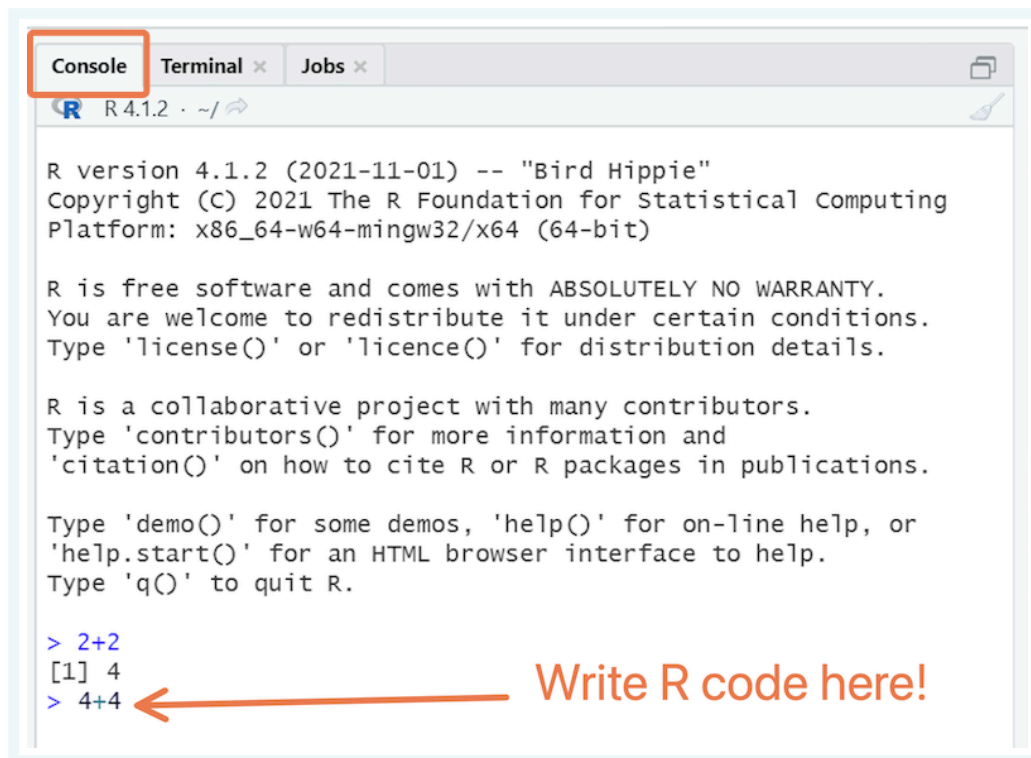


You should see a window like this:



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
```

Write R code here!

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

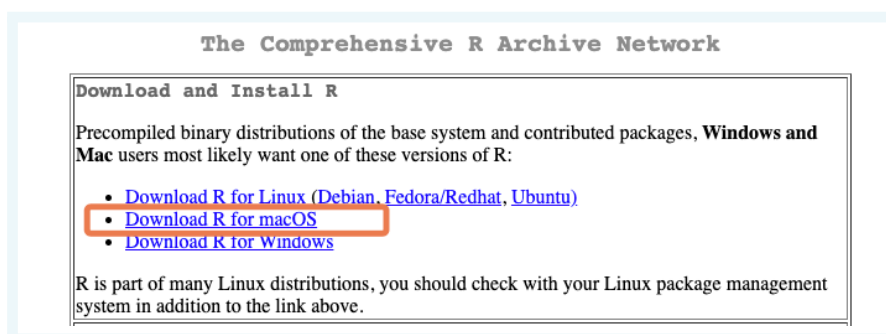
That's it; you're ready to roll. Proceed to the "wrapping up" section of the lesson.

2.5 Set up on macOS

2.5.1 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:



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: 2d9058826e4472794894a09a0b7156a58747
(ca. 90MB) for Intel Macs

Latest version 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 omitted when choosing "custom install", they are only needed if you want to use the `tecltk` R package or build package documentation from sources.

Note: the use of X11 (including `tecltk`) 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: 23b3c41167b97716485d59a75657827923d4db2bc
(ca. 89MB) for M1 Macs only!

Latest version for M1 Macs

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 `tecltk`) 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)

[Mac-GUI-1.78.tar.gz](#)
SHA1 hash: 23b3c41167b97716485d59a75657827923d4db2bc

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

R-3.6.3.pkg (signed)
SHA1 hash: c4632961f9b45d77801058d9a225a9123b3557c4
(ca. 77MB)

For older macs

Binaries for legacy OS X systems:

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 omitted when choosing "custom install", they are only needed if you want to use the `tecltk` R package or build package documentation from sources.

- 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.

2.5.2 Download, install & run RStudio

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

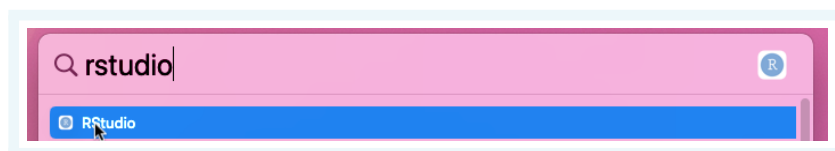
2.

Download RStudio Desktop. Recommended for your system:

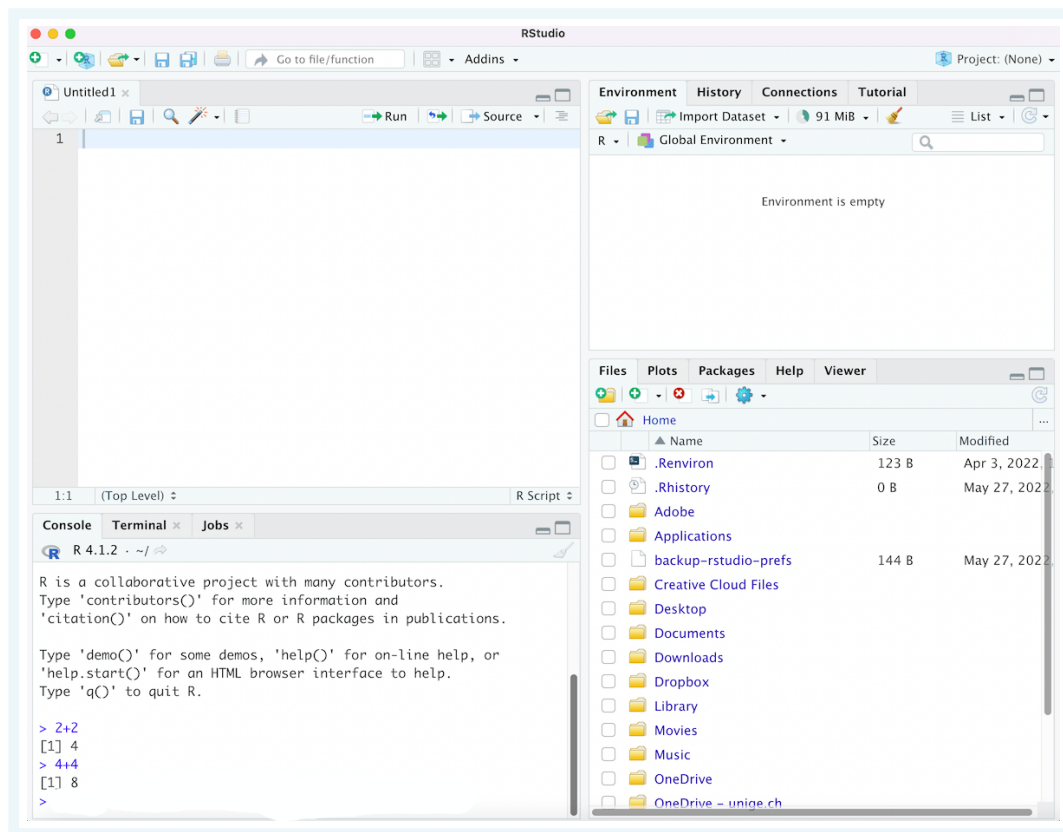
DOWNLOAD RSTUDIO FOR MAC
2022.02.0+443 | 217.18MB

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.

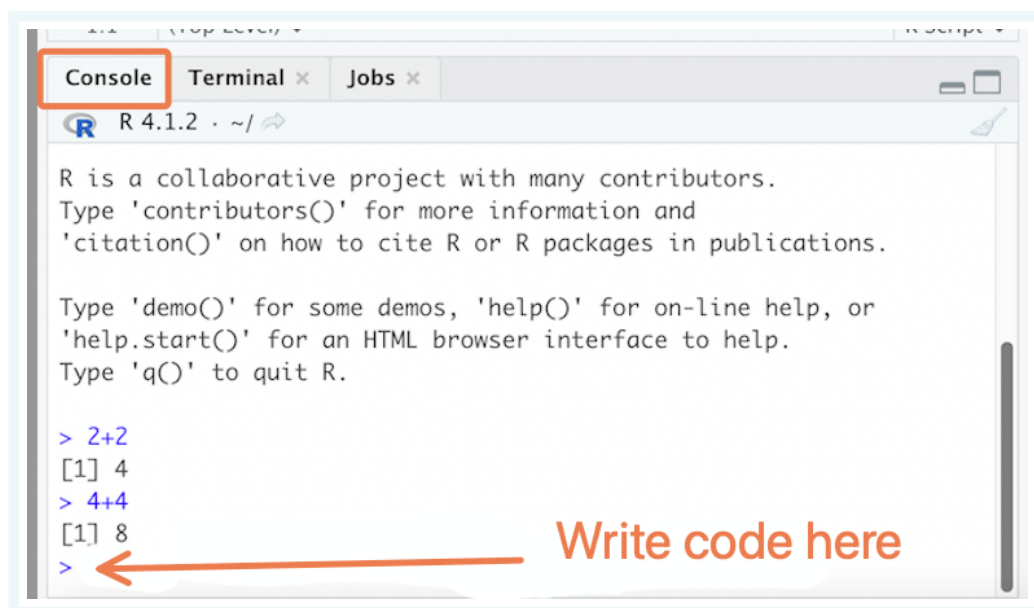


You should see a window like this:



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:



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

2.6 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!

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.

Chapter 3

Using RStudio

3.1 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.

3.2 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).

3.3 The RStudio panes

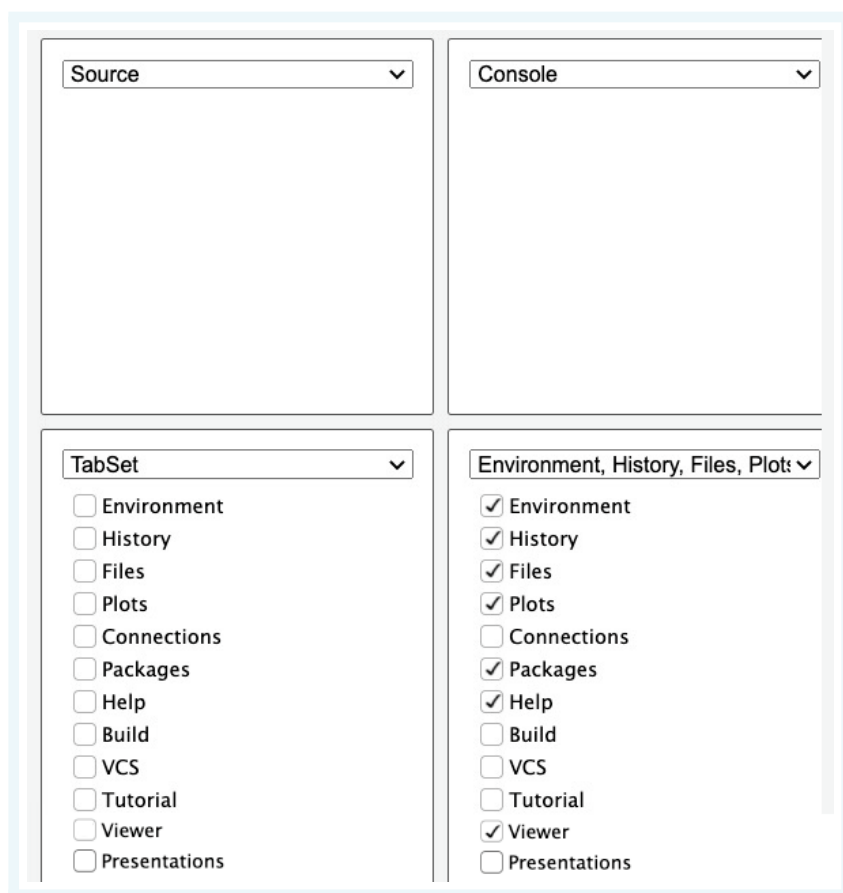
By default, RStudio is arranged into four window panes.

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.

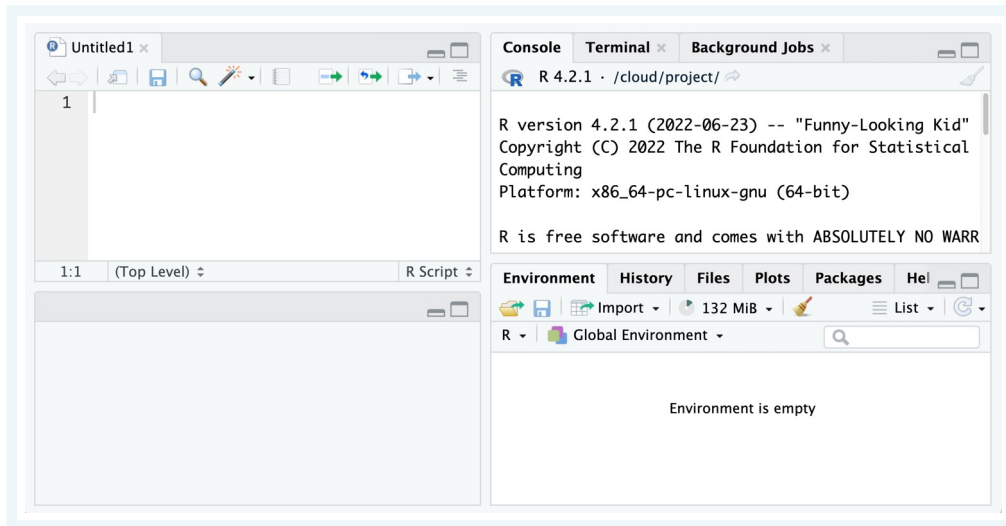


At the top left pane is the Source tab, and at the top right pane, you should have the Console tab.

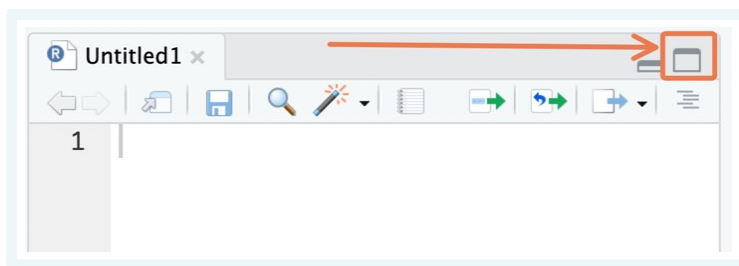
Then at the bottom left pane, no tab options should be checked—this section should be left empty, with the drop-down saying just “TabSet”.

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

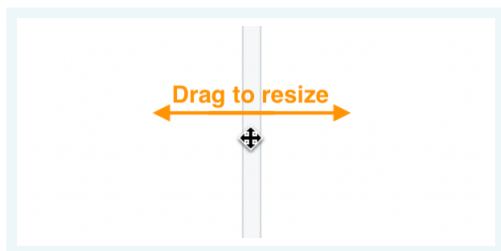
Great, now you should have an RStudio window that looks something like this:



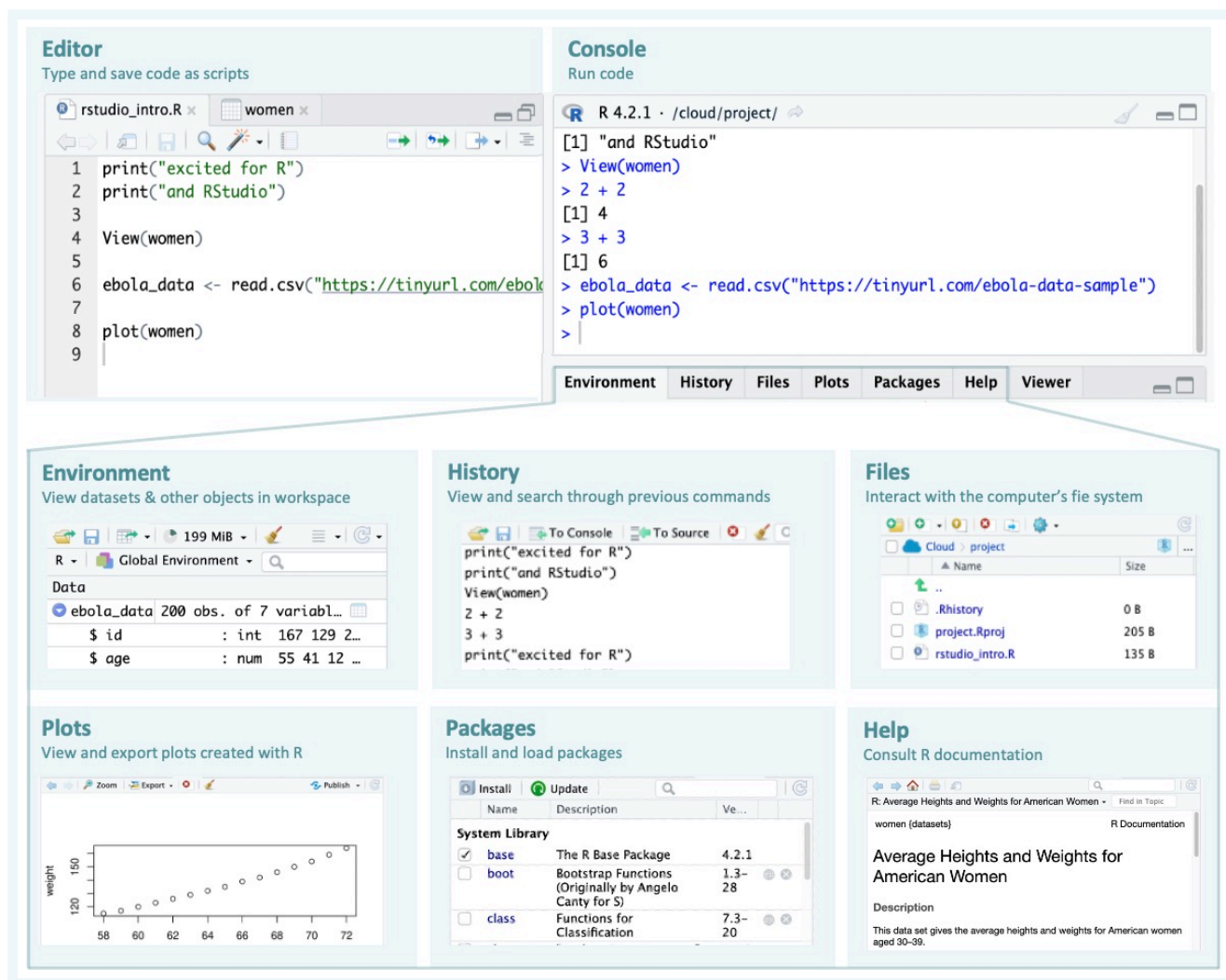
The top-left pane is where you will do most of the coding. Make this larger by clicking on its maximize icon:



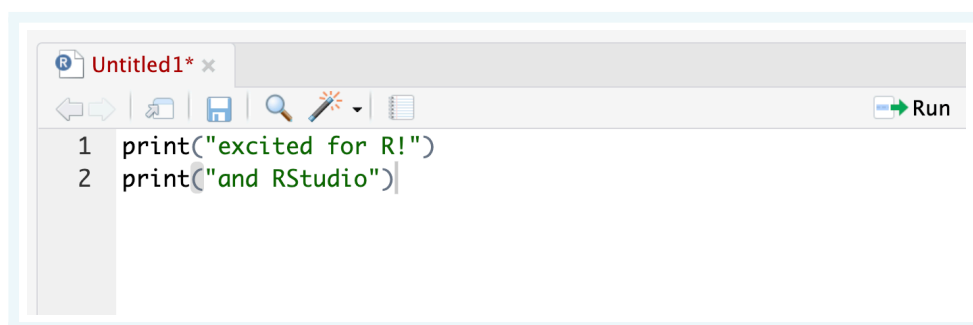
Note that you can drag the bar that separates the window panes to resize them.



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



3.3.1 Source/Editor



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.

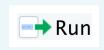
You can also **run multiple lines at once**. To try this, add a second line to your script, so that it now reads:

```
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 (), 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.



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

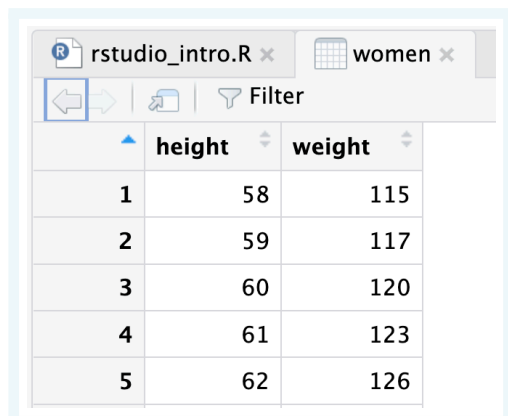
Next, **save the script**. Hit Command/Control + S to bring up the Save dialog box. Give it a file name like "rstudio_intro".

- 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

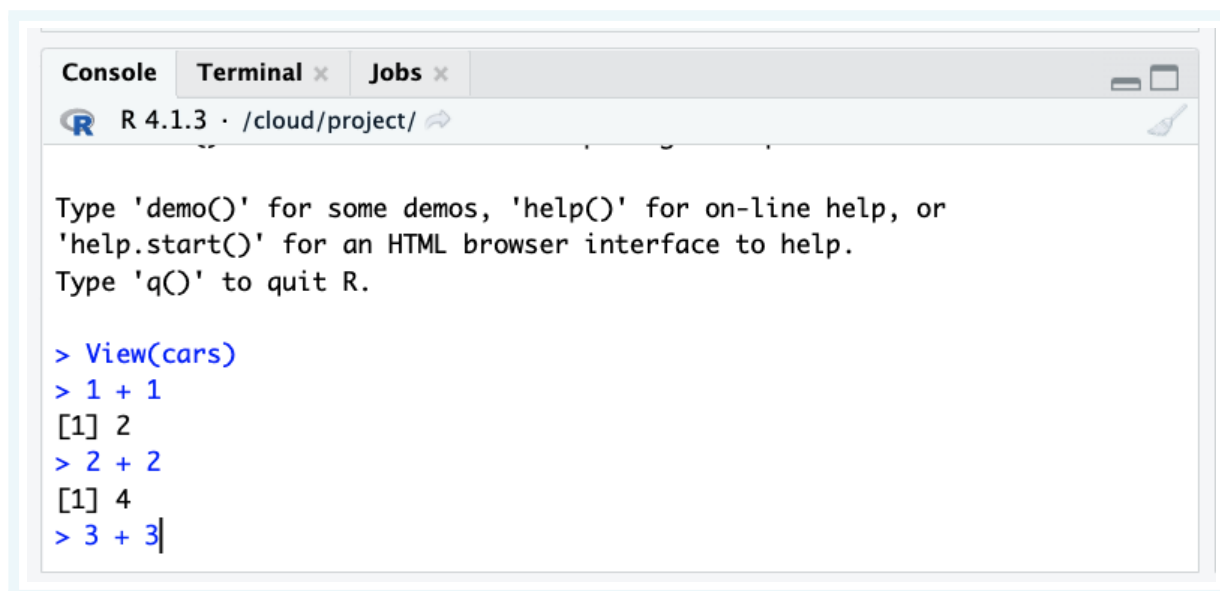
women is the name of a dataset that comes loaded with R. It gives the average heights and weights for American women aged 30–39.

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

3.3.2 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’.



```

R 4.1.3 · /cloud/project/

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.

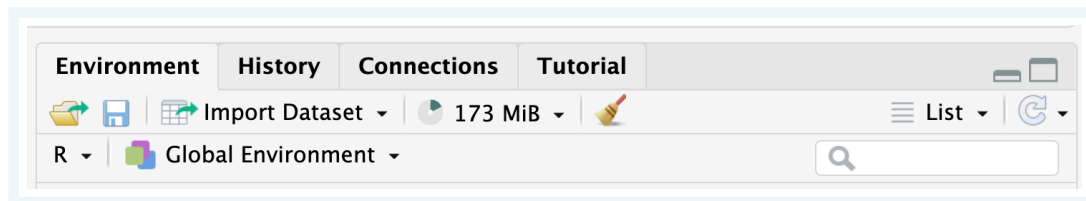
> View(cars)
> 1 + 1
[1] 2
> 2 + 2
[1] 4
> 3 + 3

```

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*.

3.3.3 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:

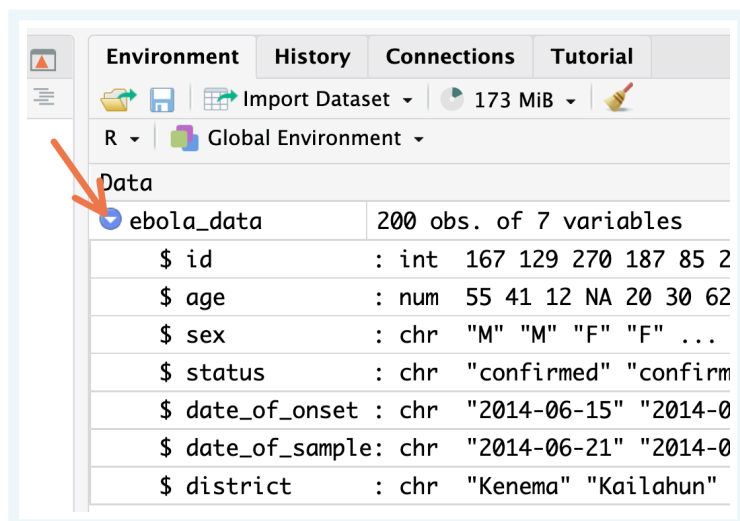
```
ebola_data <- read.csv("https://tinyurl.com/ebola-data-sample")
```

i 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.



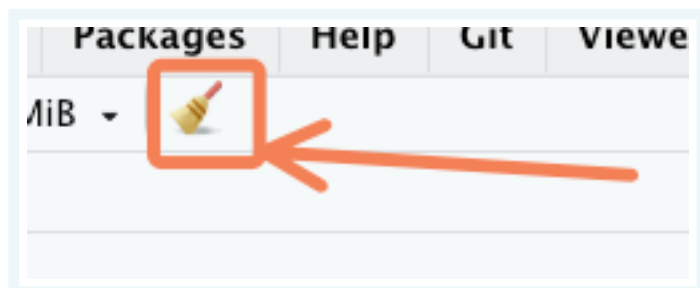
Try clicking directly on the `ebola_data` dataset from the Environment tab. This opens it in a 'View' tab.

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.

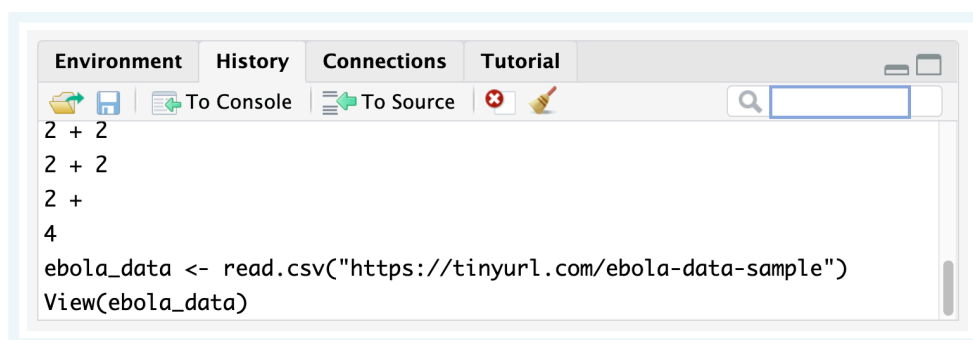
The broom icon, at the top of the Environment pane can also be used to clear your workspace.



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

3.3.4 History

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



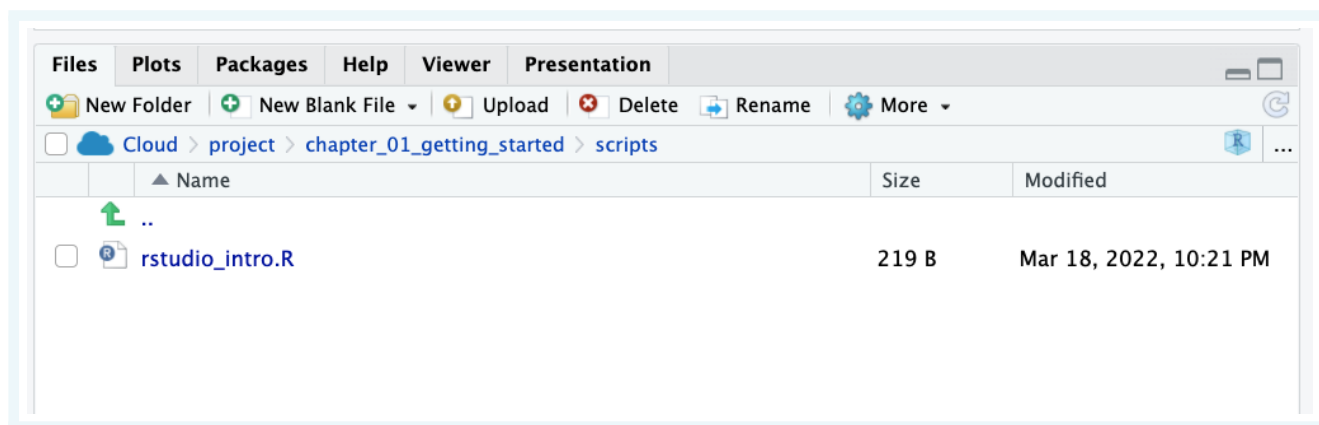
You can click a line to highlight it, then send it to the console or to your script with the “To Console” and “To Source” icons at the top of this tab.

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.

3.3.5 Files

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



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

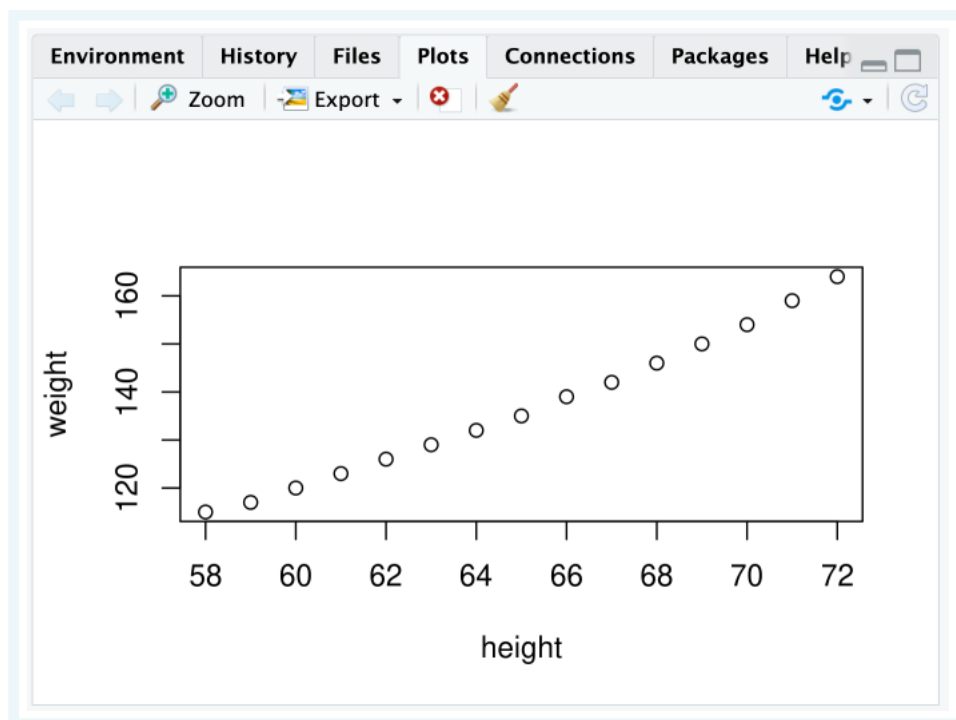
Try playing with some of the buttons here, to see what they do. You should try at least the following:

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

3.3.6 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)
```

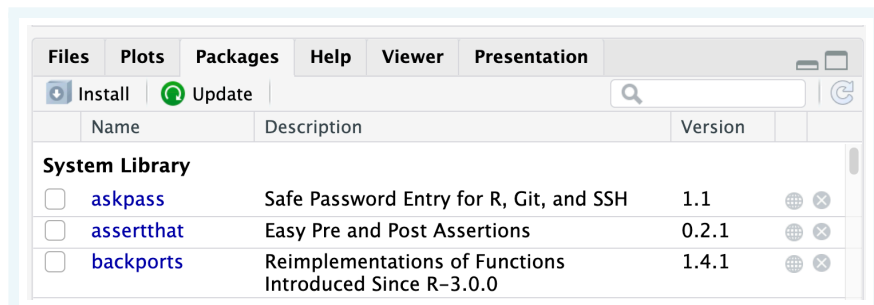


That code creates a plot of the two variables in the `women` dataset. You should see this figure in the Plots tab.

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.

3.3.7 Packages

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

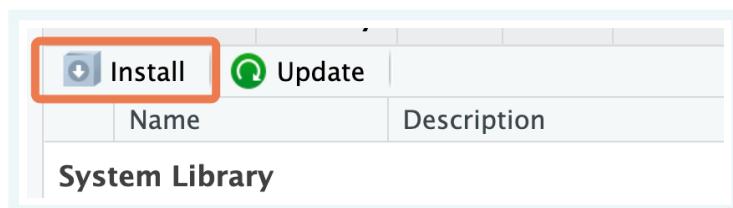


Packages are collections of R code that extend the functionality of R. We will discuss packages in detail in a future lesson.

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.

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

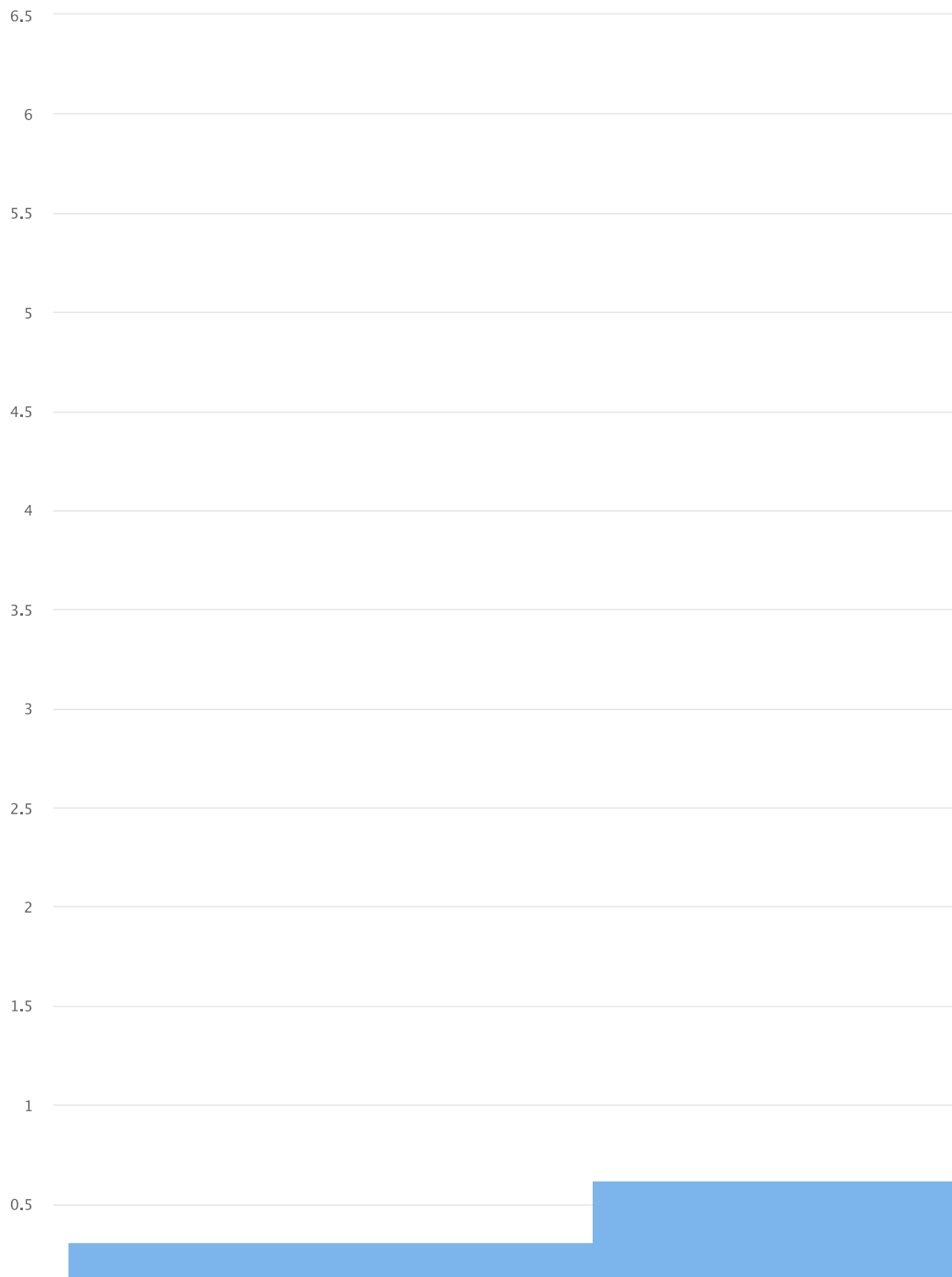
The first line installs the package. The second line *loads* the package from your package library.

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:

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

```
Registered S3 method overwritten by 'quantmod':
  method      from
as.zoo.data.frame zoo
```



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.)

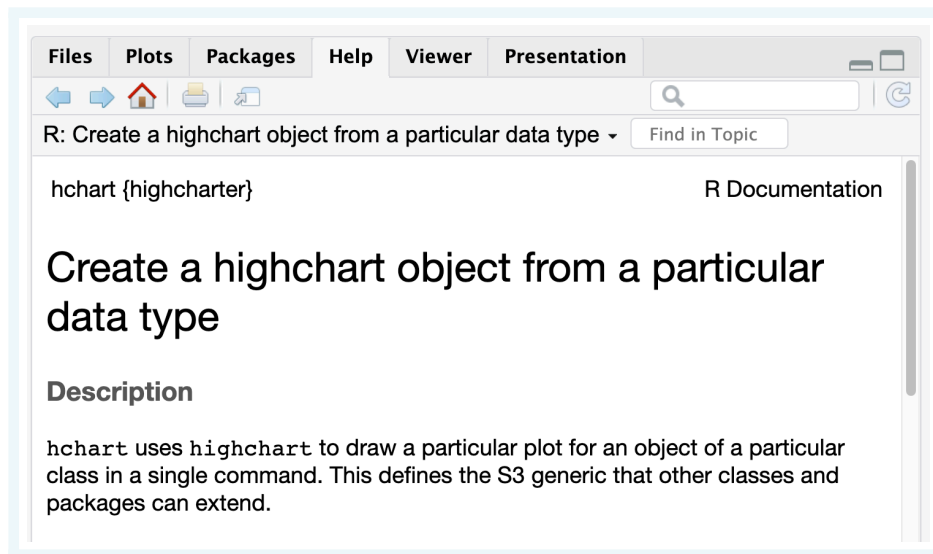
3.3.8 Viewer

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

3.3.9 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
```



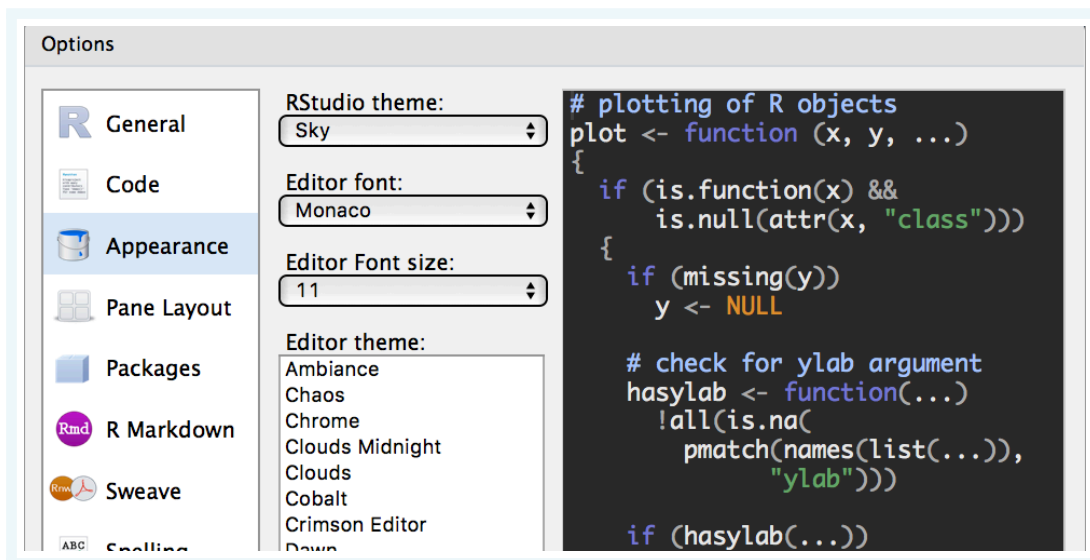
Help files are not always very easy to understand for beginners, but with time they will become more useful.

3.4 RStudio options

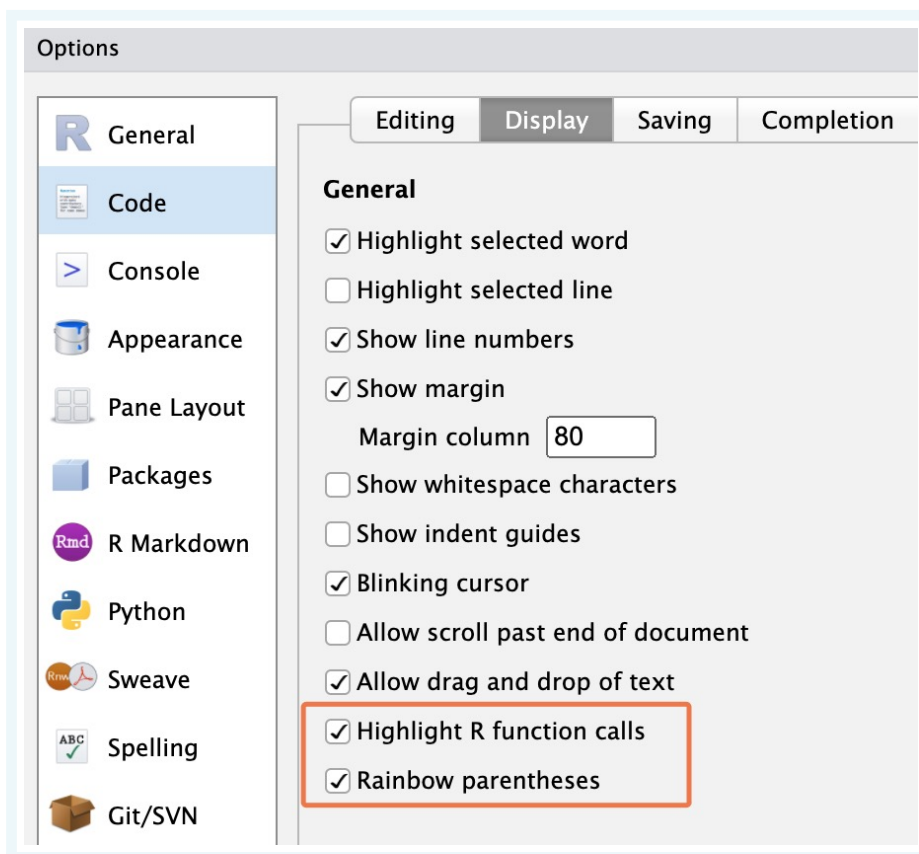
RStudio has a number of useful options for changing its look and functionality. Let's try these. You may not understand all the changes made for now. That's fine.

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.)



- 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.



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

Highlight R function calls

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



- 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.

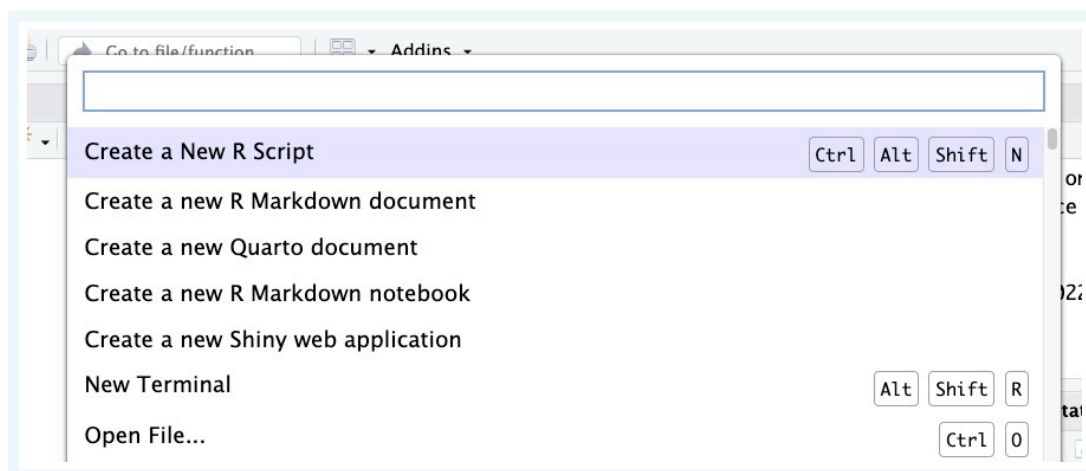
This also means that you never want to “**save your workspace to .RData on exit**”, so set this to **Never**.

3.5 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*).



Try using it to:

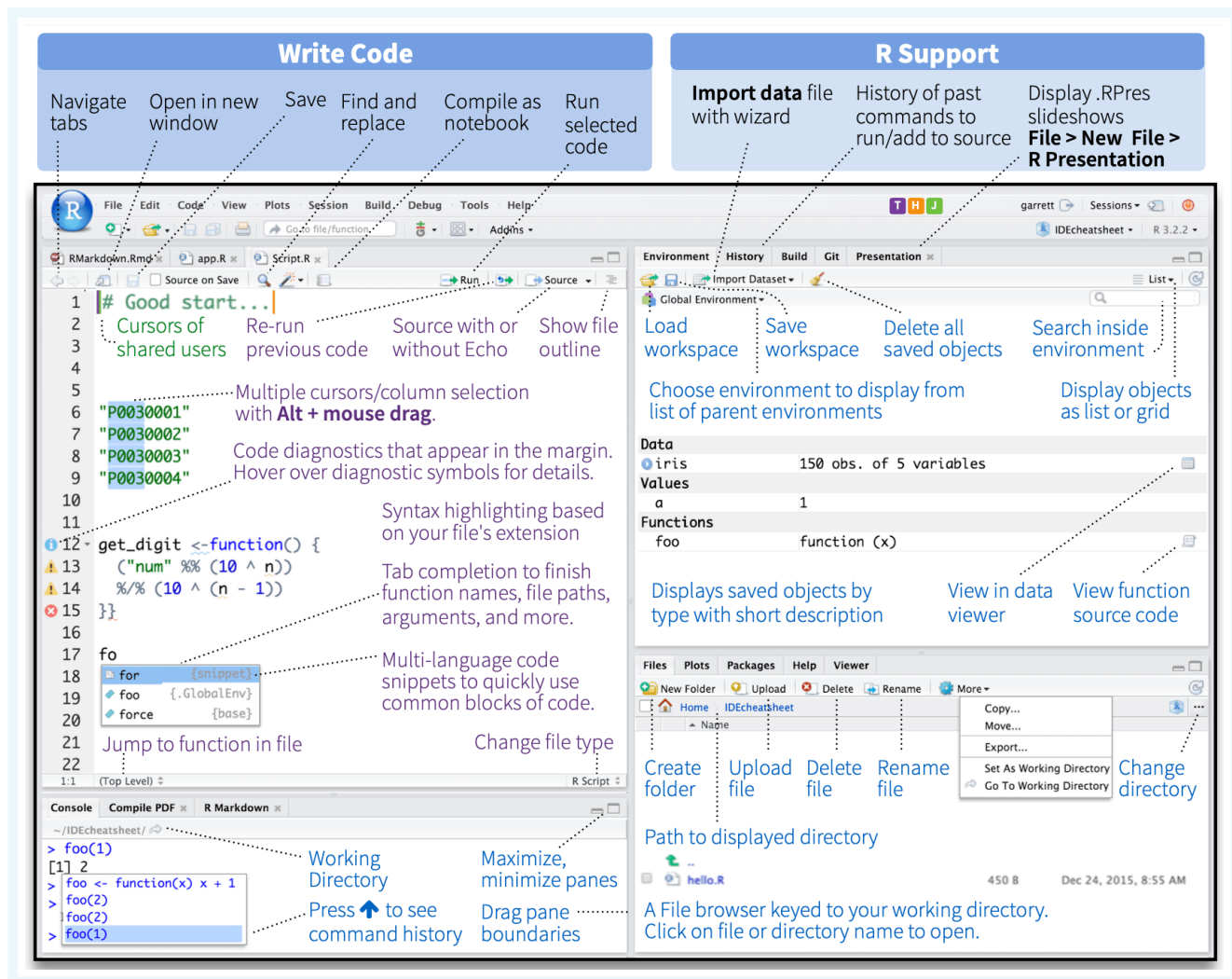
- Create a new script (Search “new script” and click on the relevant option)
- Rename a script (Search “rename” and click on the relevant option)

3.6 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:



See you in the next lesson!

3.7 Further resources

1. 23 RStudio Tips, Tricks, and Shortcuts

3.8 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>.

Chapter 4

Coding basics

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.

4.1 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.

4.2 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”.

Practice

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.

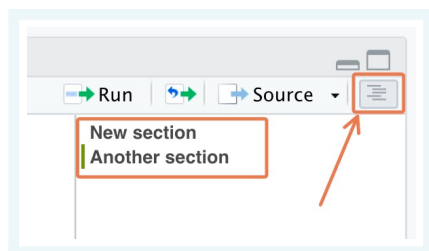
A fantastic use of comments is to separate your scripts into sections. If you put four dashes after a comment, RStudio will create a new section in your code:

```
## 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:

```
1 # New section ----  
2
```

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:



4.3 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.

 Practice**Question 2**

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

```
2 - 2 / 2
```

```
[1] 1
```

4.4 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 print `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.

4.5 Objects in R

4.5.1 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.

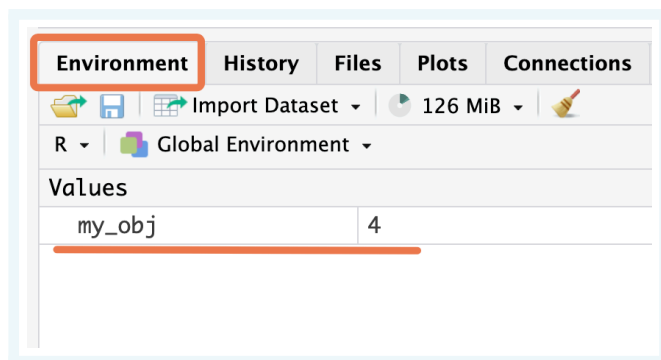
Side Note

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.

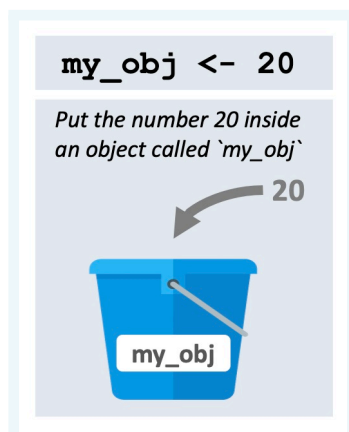


4.5.2 What is an object?

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’”.

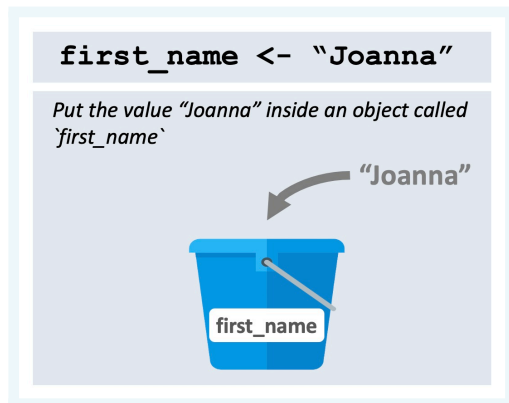


Once the code is run, we would say, in R terms, that “the value of object called `my_obj` is 20”.

And if you run this code:

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

you are instructing R to “put the value ‘Joanna’ inside the bucket called ‘first_name’”.



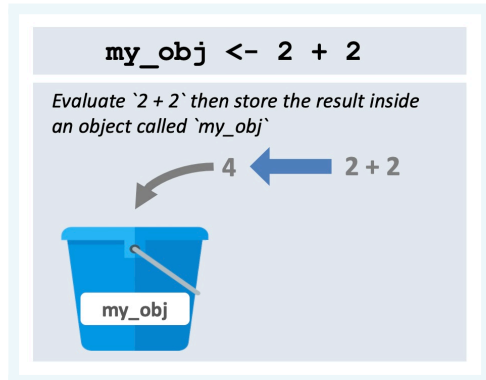
Once the code is run, we would say, in R terms, that “the value of the `first_name` object is Joanna”.

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.



Practice

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

4.5.3 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.

4.5.4 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)
```

4.5.5 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.

4.5.6 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.

Practice

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.

4.5.7 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.

Practice

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_1st_name, my_last_name)
```

4.5.8 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.

💡 Practice

Question 6

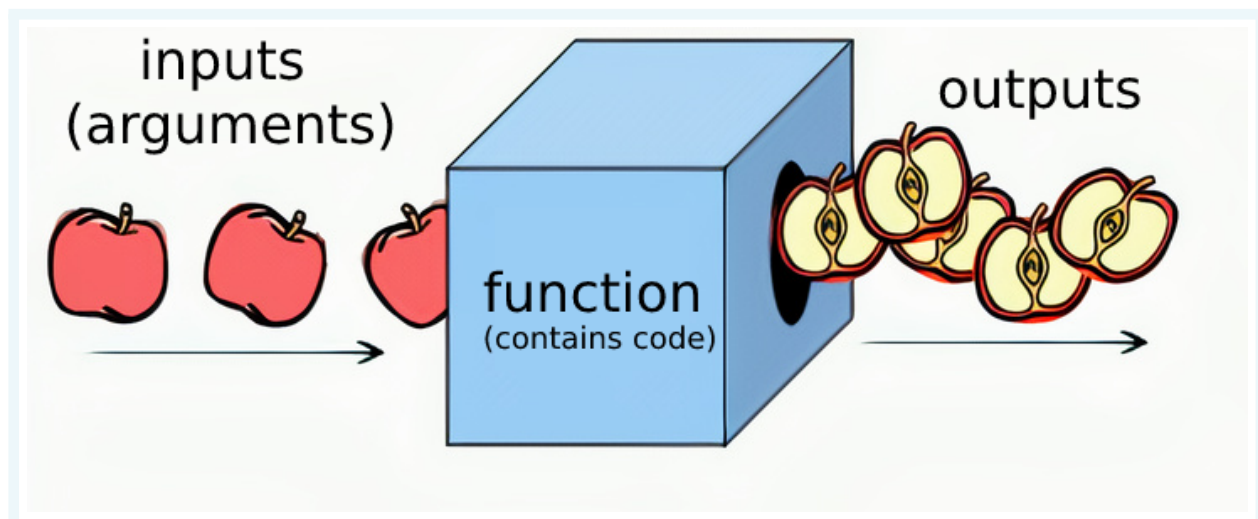
In the code chunk below, we are attempting to take the top 20 rows of the `ebola_data` table. All but one of these lines has an error. Which line will run properly?

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

4.6 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)
```

4.6.1 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)

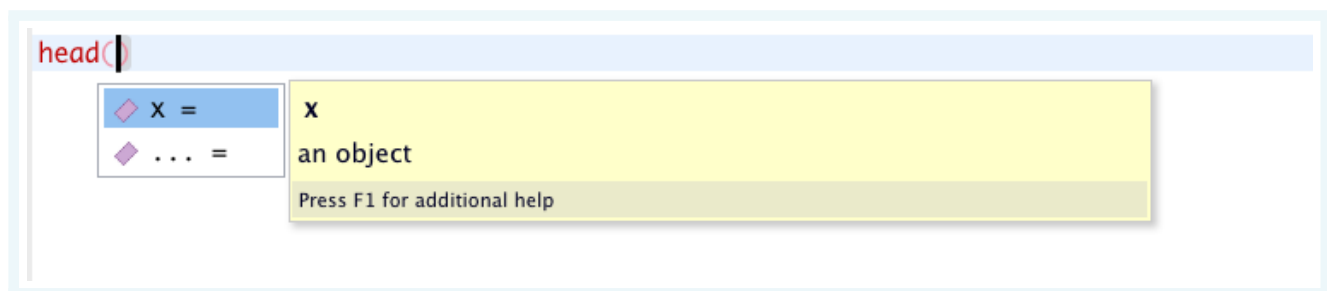
Some function arguments can be skipped altogether, because they have *default* values.

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:



Practice

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"
```

4.6.2 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.

💡 Practice

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"))
```

4.7 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.

4.7.1 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.

4.7.2 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()`:

```
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:

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

is equivalent to:

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

 Practice**Question 9**

Consider the code below:

```
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.

4.7.3 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.

Practice

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.

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

B.

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

C.

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

4.8 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!

4.9 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}`

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/>.

Chapter 5

Data dive: Ebola in Sierra Leone

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()`.

5.1 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:

```
# A tibble: 10 x 7
      id   age sex   status   date_of_onset date_of_sample district
  <dbl> <dbl> <chr> <chr>      <date>        <date>        <chr>
1     92     6 M confirmed 2014-06-10    2014-06-15    Kailahun
2     51    46 F confirmed 2014-05-30    2014-06-04    Kailahun
3    230   NA M confirmed 2014-06-26    2014-06-30    Kenema
4    139    25 F confirmed 2014-06-13    2014-06-18    Kailahun
5      8     8 F confirmed 2014-05-22    2014-05-27    Kailahun
6    215    49 M confirmed 2014-06-24    2014-06-29    Kailahun
7    189    13 F confirmed 2014-06-19    2014-06-24    Kailahun
8    115    50 M confirmed 2014-06-10    2014-06-25    Kailahun
9    218    35 F confirmed 2014-06-25    2014-06-28    Kenema
10   159    38 F confirmed 2014-06-14    2014-06-22    Kailahun
```

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?**

5.2 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

i 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**

5.2.1 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
```

5.2.2 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:

```
## 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:

```
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()`.)

5.3 Importing data into R

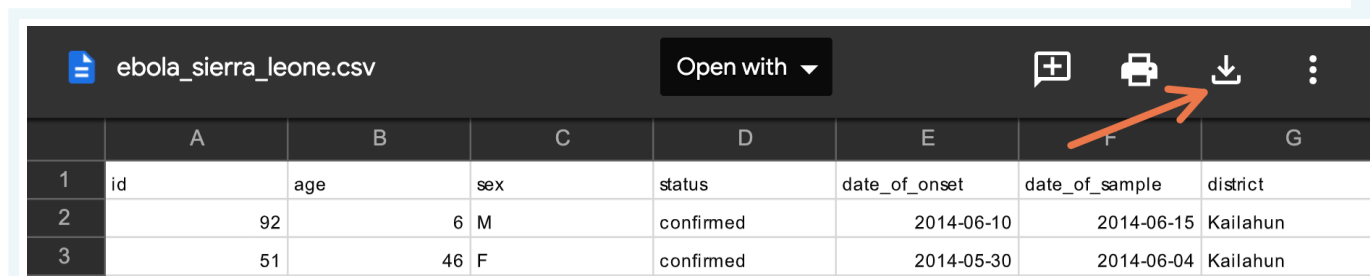
Now that the needed packages are loaded, you should import the dataset.

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 was “confirmed” or “suspected”; the `date_of_onset`, when Ebola-like symptoms arose in a patient; and the `date_of_sample`, when the test sample was taken. To learn more about these data, visit the source publication here: [bit.ly/ebola-data-source](https://doi.org/10.1073/pnas.1518587113). 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.

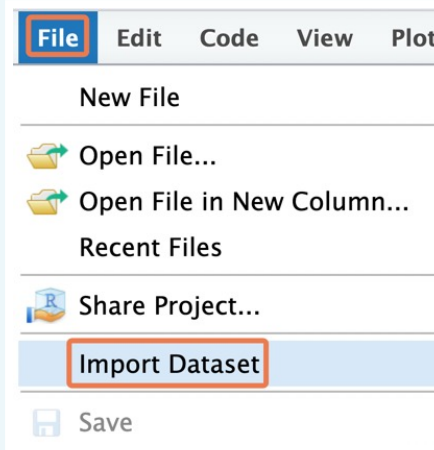
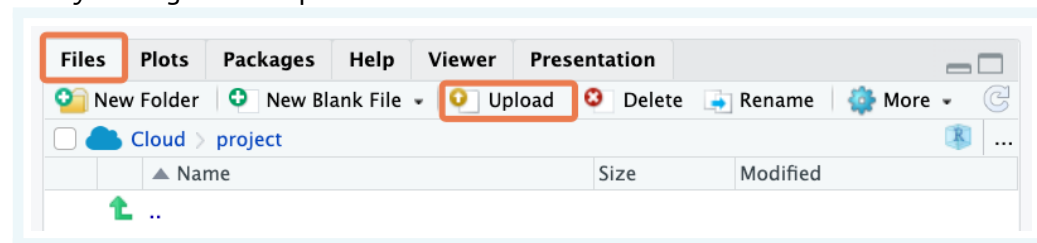


	A	B	C	D	E	F	G
1	id	age	sex	status	date_of_onset	date_of_sample	district
2	92	6	M	confirmed	2014-06-10	2014-06-15	Kailahun
3	51	46	F	confirmed	2014-05-30	2014-06-04	Kailahun

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

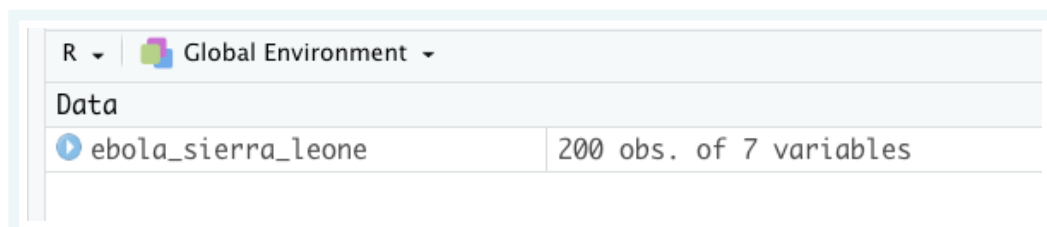
NOTE: If you are using RStudio Cloud, you need to upload your dataset to the cloud. Do this in the “Files” tab by clicking on the “Upload” button.



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:

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:



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 —

**Copy this
(or something similar)**

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")
```

i Recap

Nice work so far!

Your R script should look similar to this:

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

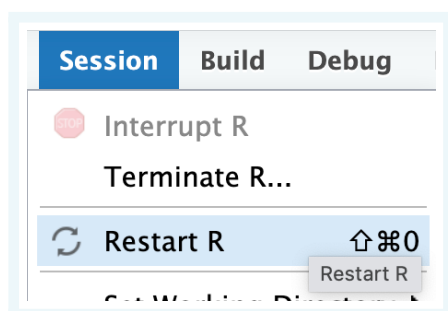
## Load packages ----
if(!require(pacman)) install.packages("pacman")
pacman::p_load(
  tidyverse,
  inspectdf,
  plotly,
  janitor,
  visdat
)

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

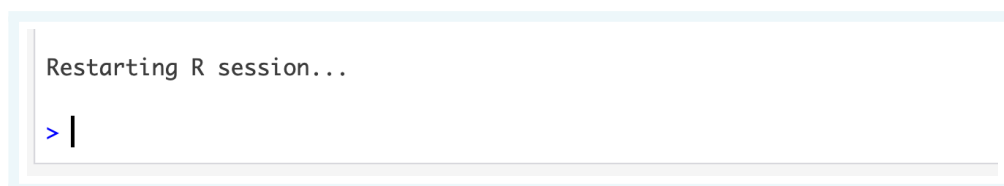
5.4 Intro to reproducibility

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.

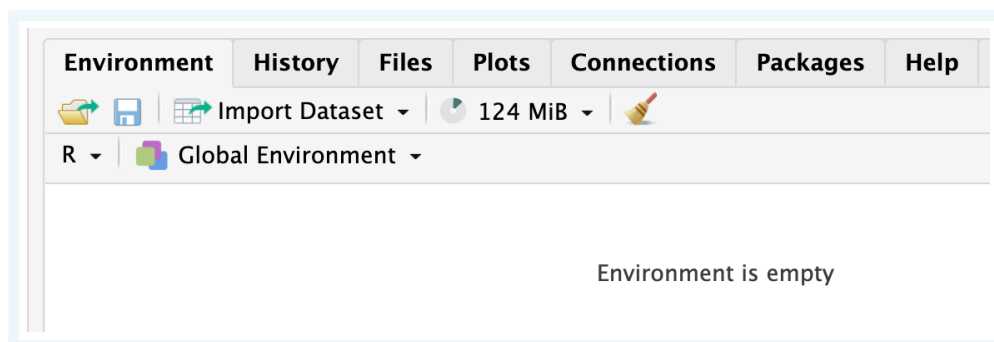
Try restarting R and rerunning the script now. Save your script with **Control/Command + s**, then *restart* R with the RStudio Menu, at **Session > Restart R**. On RStudio Cloud, the menu option looks like this:



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



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!

Vocab

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.

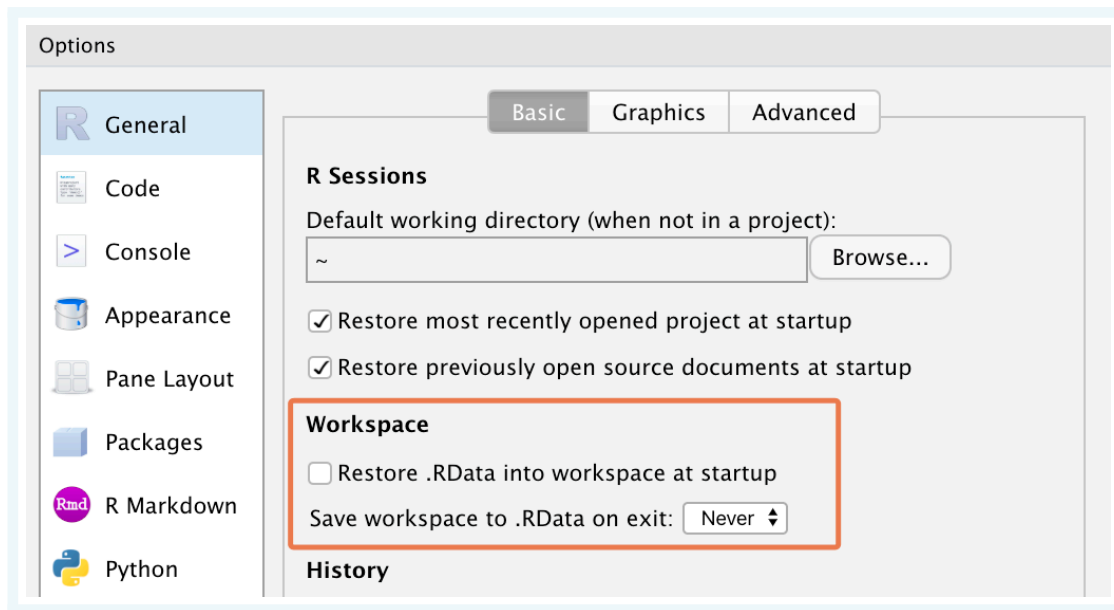
While you can use RStudio’s graphical user interface to point-and-click your way through the data import process, you should always copy the relevant code to your script so that your script remains a reproducible record of all your analysis steps.

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”.



5.5 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 x 7
   id   age sex  status  date_of_onset date_of_sample district
<dbl> <dbl> <chr> <chr>    <date>        <date>        <chr>
1    92     6 M confirmed 2014-06-10    2014-06-15    Kailahun
2    51    46 F confirmed 2014-05-30    2014-06-04    Kailahun
3   230    NA M confirmed 2014-06-26    2014-06-30    Kenema
4   139    25 F confirmed 2014-06-13    2014-06-18    Kailahun
5     8     8 F confirmed 2014-05-22    2014-05-27    Kailahun
6   215    49 M confirmed 2014-06-24    2014-06-29    Kailahun
```

```
tail(ebola_sierra_leone)
```

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

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
```

i 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
Min.   : 1.00   Min.   : 1.80   Length:200   Length:200
1st Qu.: 62.75   1st Qu.:20.00   Class :character   Class :character
Median :131.50   Median :35.00   Mode  :character   Mode  :character
Mean   :136.72   Mean   :33.85
3rd Qu.:208.25   3rd Qu.:45.00
Max.   :285.00   Max.   :80.00
      NA's      :4

date_of_onset    date_of_sample    district
Min.   :2014-05-18   Min.   :2014-05-23   Length:200
1st Qu.:2014-06-01   1st Qu.:2014-06-07   Class :character
Median :2014-06-13   Median :2014-06-18   Mode  :character
Mean   :2014-06-12   Mean   :2014-06-17
3rd Qu.:2014-06-23   3rd Qu.:2014-06-29
Max.   :2014-06-29   Max.   :2014-07-17

```

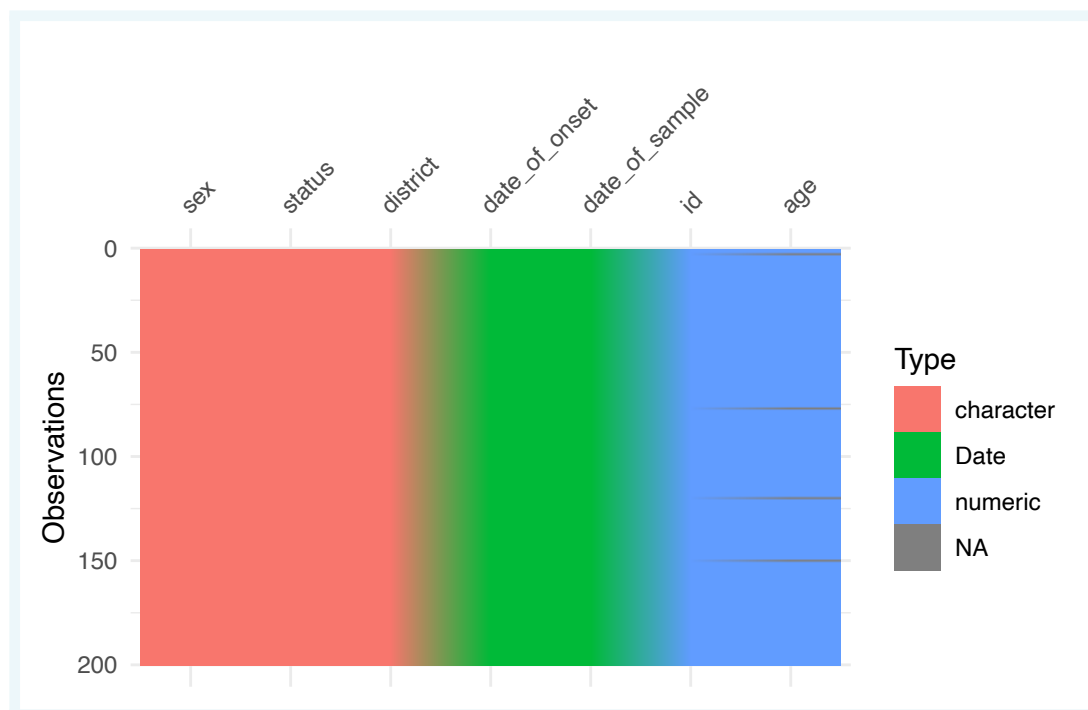
As you can see, for numeric columns in your dataset, `summary()` gives you the minimum value, the maximum value, the mean, median and the 1st and 3rd [quartiles](#).

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.

5.5.1 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.

5.5.2 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).

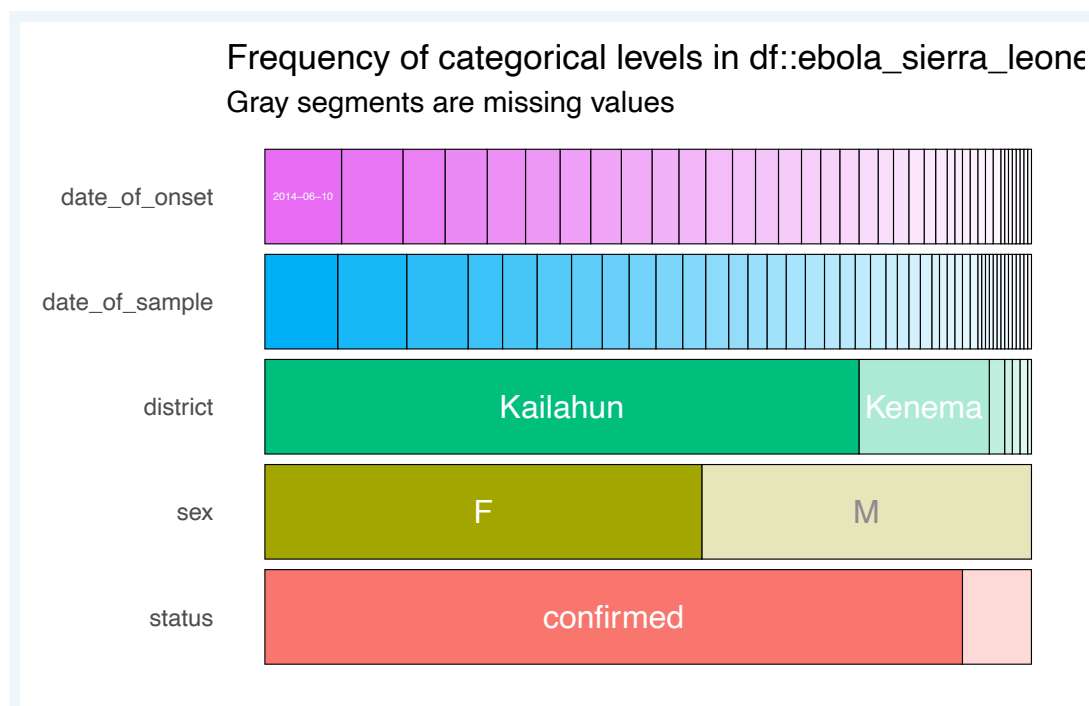
```
inspect_cat(ebola_sierra_leone)
```

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

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

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

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

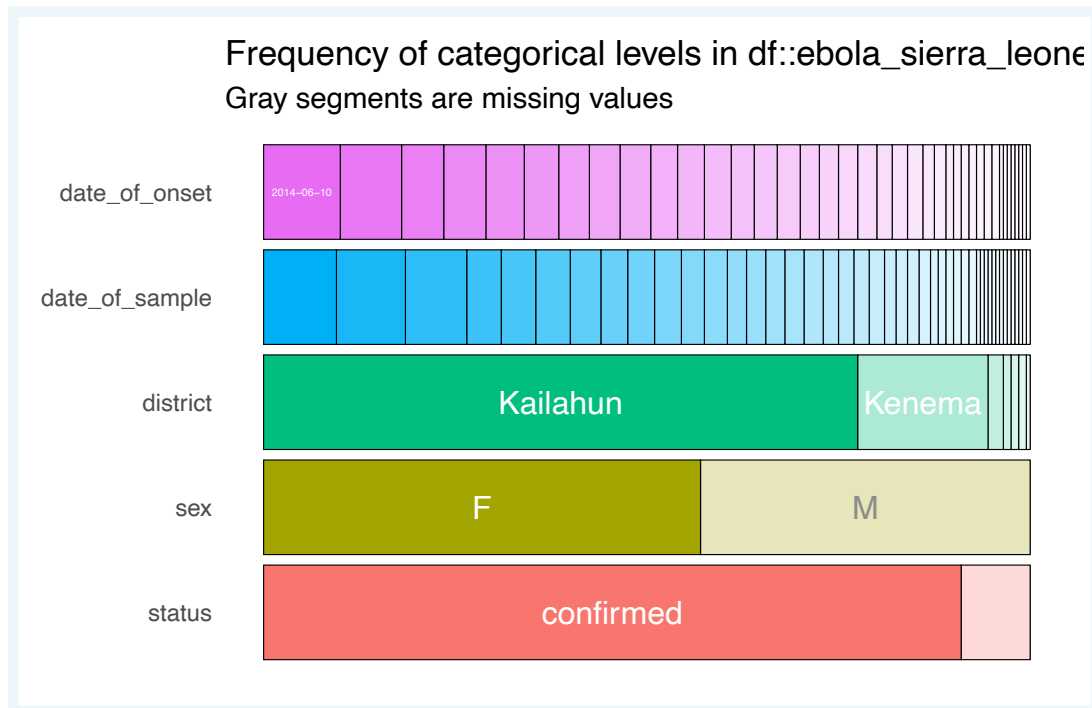


You get a wonderful figure showing the distribution of all categorical and date variables!

i Side Note

You could also run:

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



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:



i 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.

5.6 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.

5.6.1 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
[16] 35.0 43.0 17.0  3.0 60.0 38.0 41.0 49.0 12.0 74.0 21.0 27.0 41.0 42.0 60.0
[31] 30.0 50.0 50.0 22.0 40.0 35.0 19.0  3.0 34.0 21.0 73.0 65.0 30.0 70.0 12.0
[46] 15.0 42.0 60.0 14.0 40.0 33.0 43.0 45.0 14.0 14.0 40.0 35.0 30.0 17.0 39.0
[61] 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
[76]  7.0  NA 35.0 36.0 42.0 35.0 25.0 30.0 30.0 28.0 14.0 20.0 60.0 67.0 35.0
[91] 50.0  4.0 28.0 38.0 30.0 26.0 37.0 30.0  3.0 56.0 32.0 35.0 54.0 42.0 48.0
[106] 11.0  1.8 63.0 55.0 20.0 62.0 62.0 42.0 65.0 29.0 20.0 33.0 30.0 35.0  NA
[121] 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 25.0 10.0
[136] 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
[151] 54.0 18.0 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
[166] 30.0 60.0  5.0 18.0  2.0 70.0 35.0  3.0 30.0 80.0 62.0 20.0 45.0 18.0 28.0
[181] 48.0 38.0 39.0 26.0 60.0 35.0 20.0 50.0 11.0 36.0 29.0 57.0 35.0 26.0  6.0
[196] 45.0 27.0  6.0 40.0 29.0
```

i 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.

5.6.2 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 NAs 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 NAs:

```
median(ebola_sierra_leone$age) # does not work
```

```
[1] NA
```

```
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
```

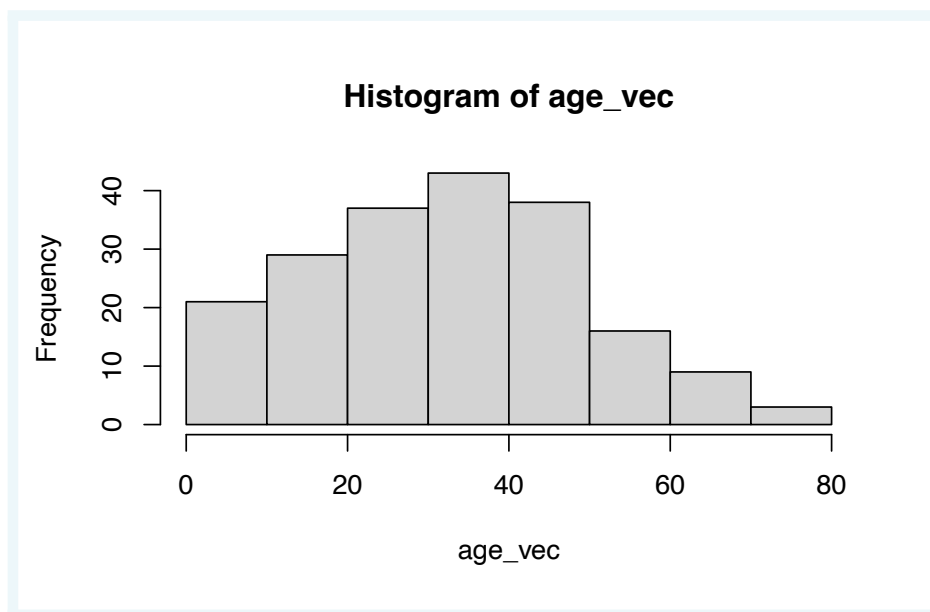
Do not feel intimidated by the long list of functions! You should not have to memorize them; rather you should feel free to Google the function for whatever operation you want to carry out. You might search something like “what is the function for standard deviation in R”. One of the first results should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

5.6.3 Visualizing a numeric variable

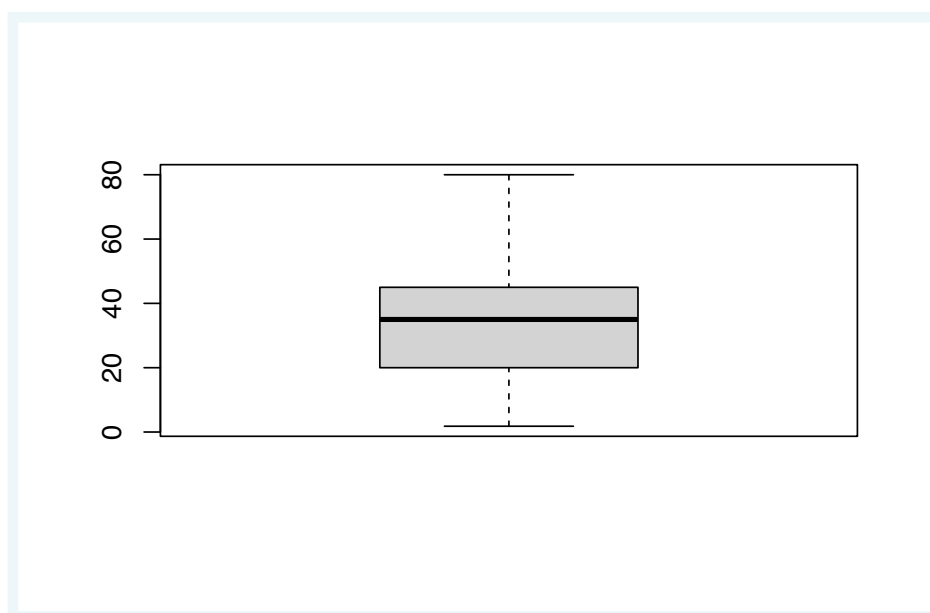
Now let’s create a graph to visualize the age variable. The two most common graphics for inspecting the distribution of numerical variables are [histograms](#) (like the output of the `inspect_num()` function you saw earlier) and [boxplots](#).

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.

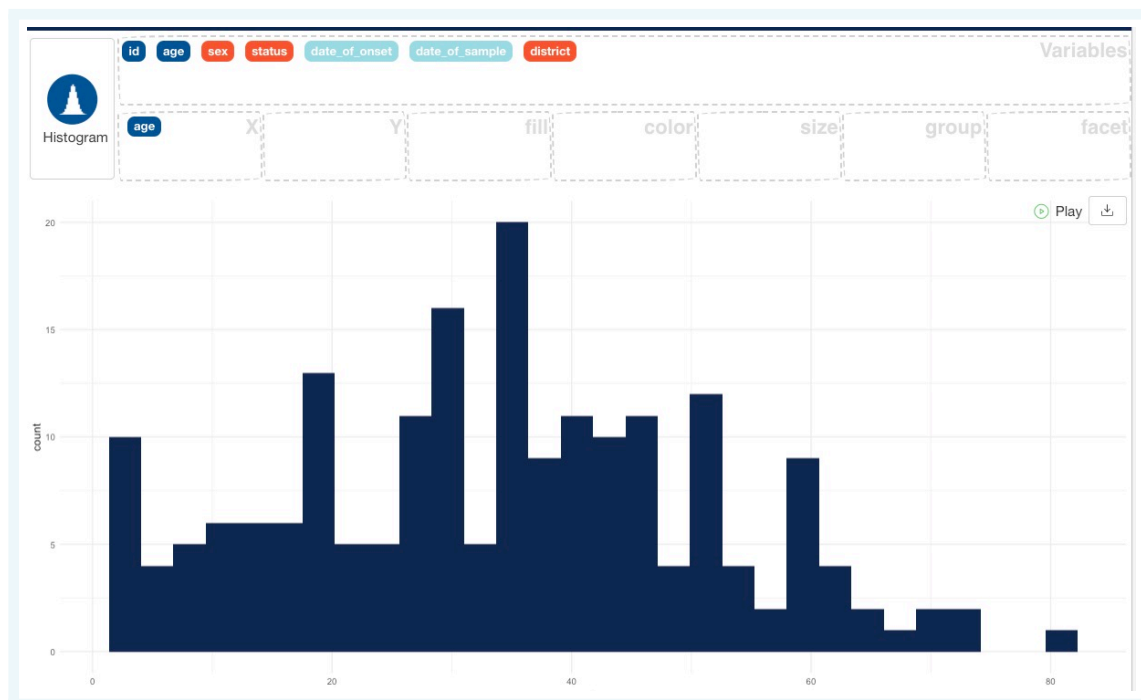
The workhorse function of the `{esquisse}` package is `esquisser()`, and this function takes a single argument—the dataset you want to visualize. So we can run:

```
esquisser(ebola_sierra_leone)
```

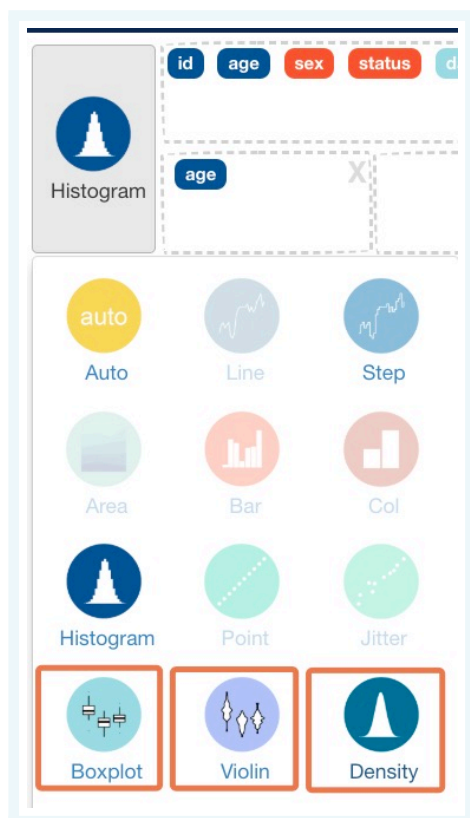
This should bring a graphic user interface that you can use to plot different variables. To visualize the age variable, simply drag age from the list of variables into the x axis box:



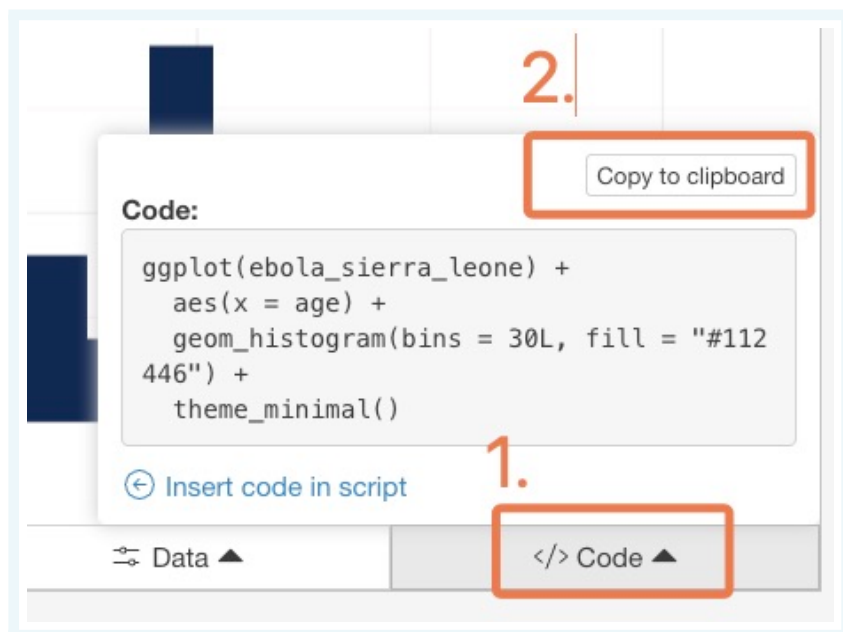
When age is in the x axis box, you should automatically get a histogram of ages:



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”:



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:

- 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.

5.7 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"
[5]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[9]	"Kenema"	"Kailahun"	"Kailahun"	"Kailahun"
[13]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[17]	"Kailahun"	"Kenema"	"Kono"	"Kailahun"
[21]	"Kailahun"	"Kailahun"	"Kenema"	"Kailahun"
[25]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[29]	"Kenema"	"Kenema"	"Kenema"	"Kailahun"
[33]	"Kailahun"	"Bo"	"Kailahun"	"Kailahun"
[37]	"Kailahun"	"Kenema"	"Kenema"	"Kenema"
[41]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[45]	"Kailahun"	"Kailahun"	"Western Urban"	"Kailahun"
[49]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[53]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[57]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[61]	"Kailahun"	"Kenema"	"Western Urban"	"Kambia"
[65]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[69]	"Kailahun"	"Kailahun"	"Kailahun"	"Kailahun"
[73]	"Kenema"	"Kailahun"	"Kailahun"	"Kenema"

```

[77] "Kailahun"      "Kailahun"      "Kenema"        "Kailahun"
[81] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[85] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[89] "Kailahun"      "Kenema"        "Kailahun"      "Kailahun"
[93] "Kailahun"      "Kono"          "Port Loko"     "Kenema"
[97] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[101] "Kenema"        "Kailahun"      "Kailahun"      "Kenema"
[105] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[109] "Kailahun"      "Kailahun"      "Kenema"        "Western Urban"
[113] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[117] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[121] "Kailahun"      "Kailahun"      "Kenema"        "Kailahun"
[125] "Kailahun"      "Kenema"        "Kailahun"      "Port Loko"
[129] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[133] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[137] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[141] "Kailahun"      "Kailahun"      "Kailahun"      "Kenema"
[145] "Kenema"        "Kailahun"      "Kenema"        "Kailahun"
[149] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[153] "Kenema"        "Kailahun"      "Kailahun"      "Kenema"
[157] "Kailahun"      "Kenema"        "Kailahun"      "Kailahun"
[161] "Kenema"        "Kailahun"      "Kailahun"      "Kailahun"
[165] "Kailahun"      "Bo"           "Kailahun"      "Kailahun"
[169] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[173] "Kenema"        "Kailahun"      "Kailahun"      "Kenema"
[177] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[181] "Kailahun"      "Kailahun"      "Kailahun"      "Western Urban"
[185] "Kailahun"      "Kailahun"      "Kenema"        "Kailahun"
[189] "Kailahun"      "Kailahun"      "Kailahun"      "Kailahun"
[193] "Kailahun"      "Kenema"        "Kenema"        "Kailahun"
[197] "Kailahun"      "Kailahun"      "Kailahun"      "Kenema"

```

Sorry for printing that very long vector!

5.7.1 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
2	155	1	34	2
Port Loko	Western Urban			
2	4			

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

`table()` is a useful “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

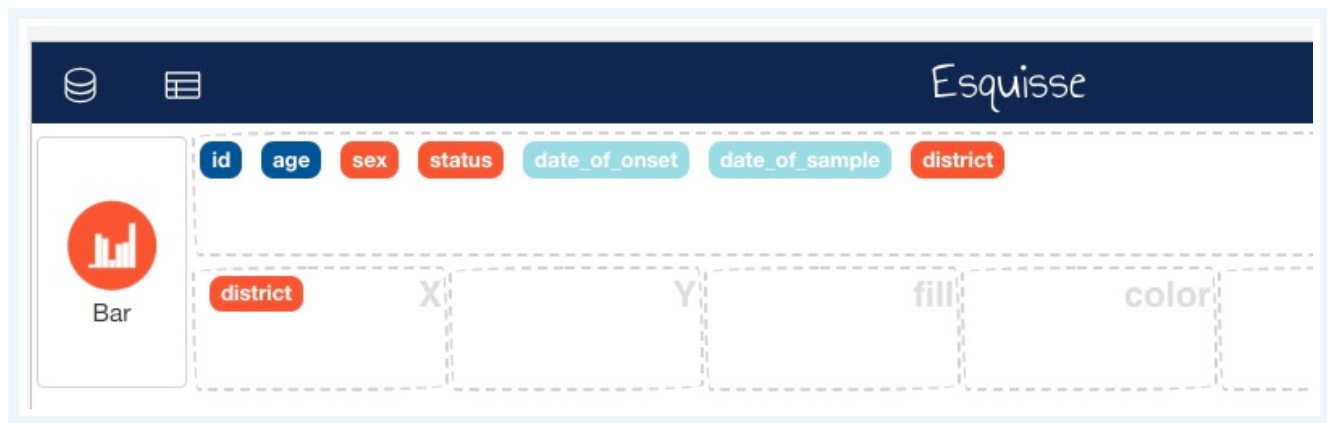
The output shows us that there were 0 women in the Bo district, 2 men in the Bo district, 91 women in the Kailahun district, and so on.

5.7.2 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.

5.8 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
```

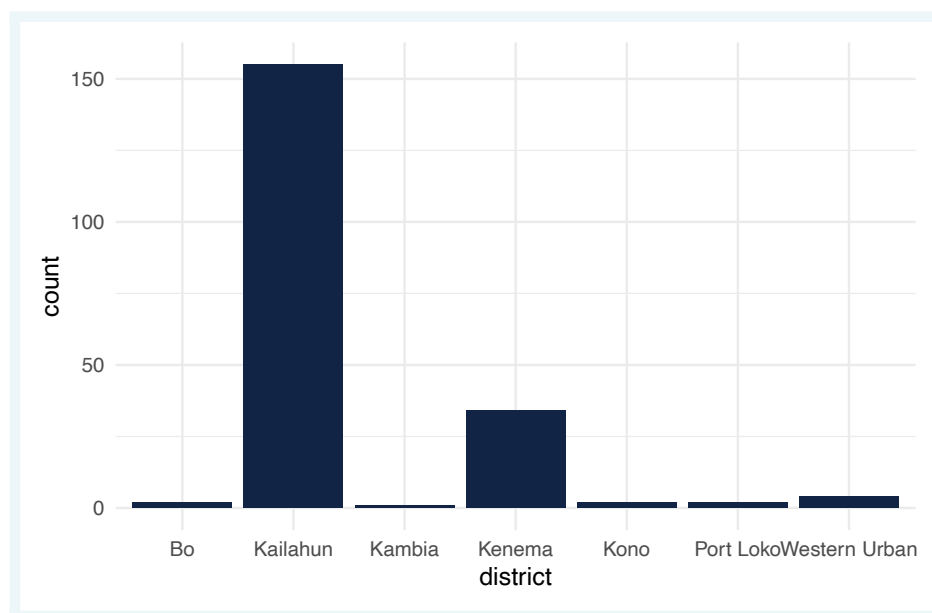
As seen in the table, there were more cases in women. Specifically, 57% of cases are of women.

- **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()
```

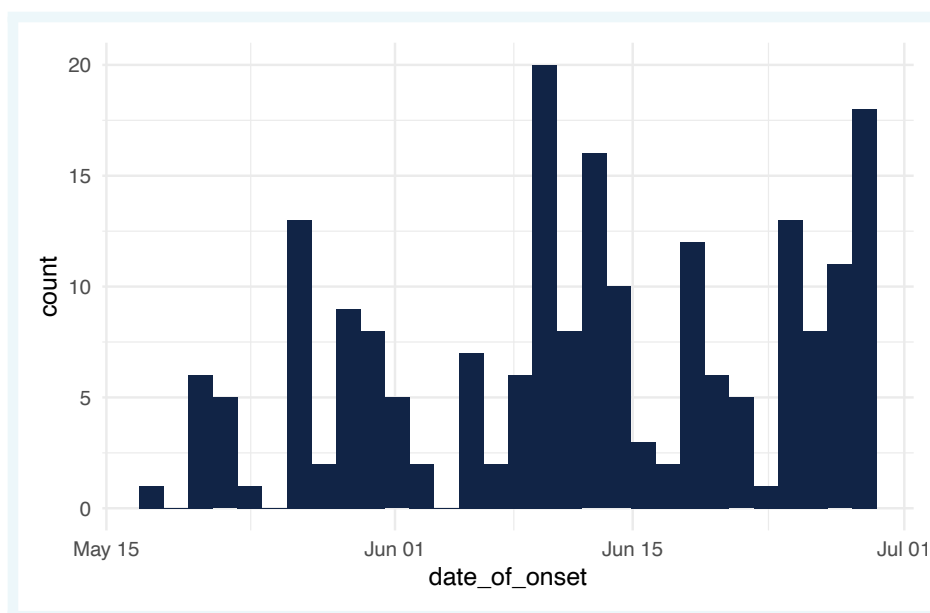


As seen, the Kailahun district had the majority of cases.

- **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!

5.9 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

5.10 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.

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 Golemund. “R Markdown: The Definitive Guide.” Home, April 11, 2022. <https://bookdown.org/yihui/rmarkdown/>.

Chapter 6

RStudio projects

6.1 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.

6.2 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.

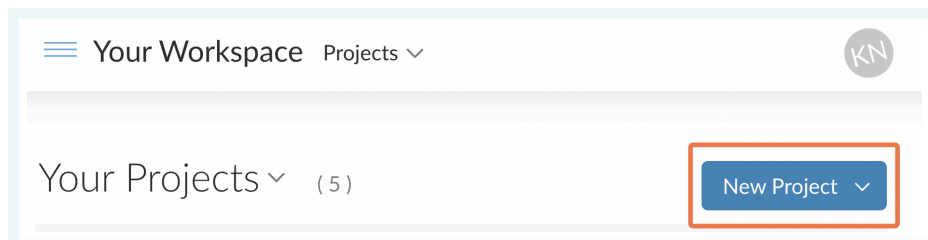
6.3 Creating a new RStudio Project

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.

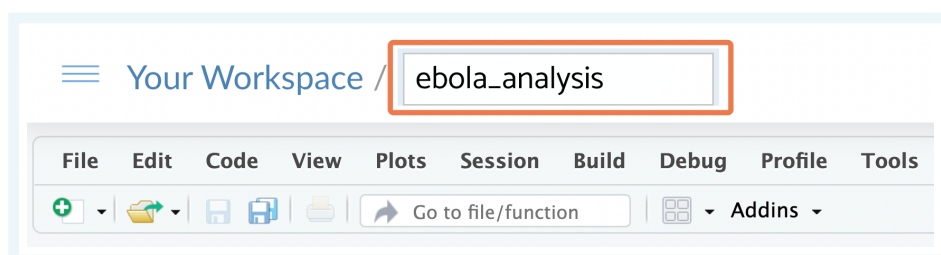
6.3.1 On RStudio Cloud

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

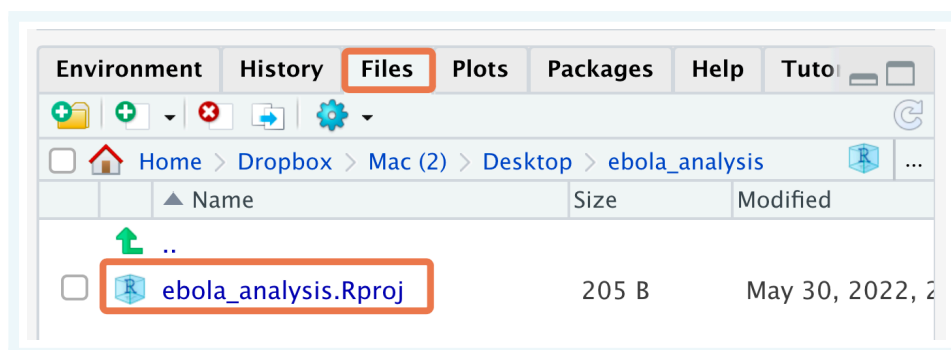
The steps are pretty simple: go to your Cloud homepage, rstudio.cloud, and click on the “New Project” button.



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



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:

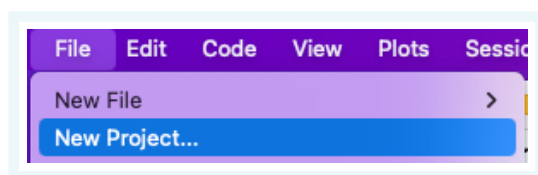


i Key Point

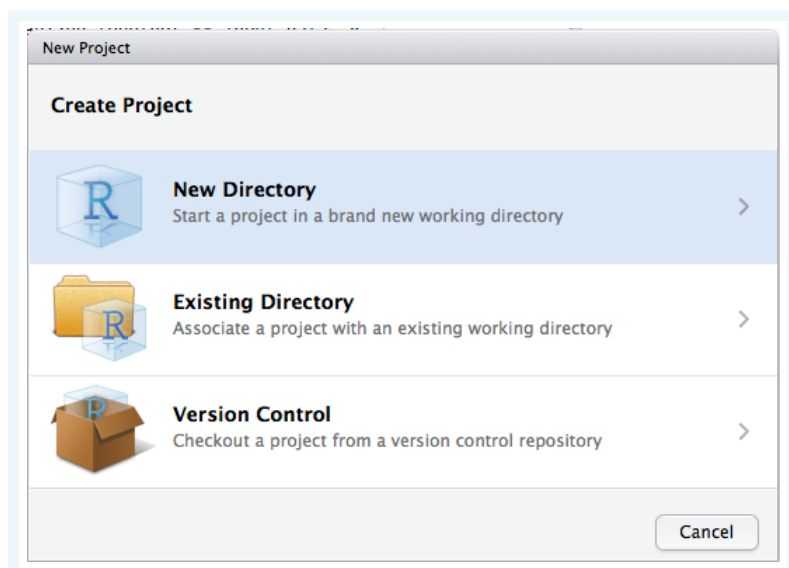
The `.Rproj` file is what turns a regular computer folder into an “RStudio Project”.

6.3.2 On a local computer

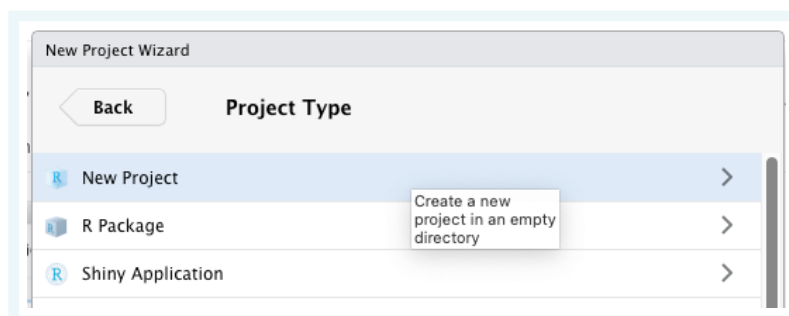
If you are on a local computer, open RStudio, then on the RStudio menu, go to `File > New Project`. Your options may look a little different from the screenshots below depending on your operating system.



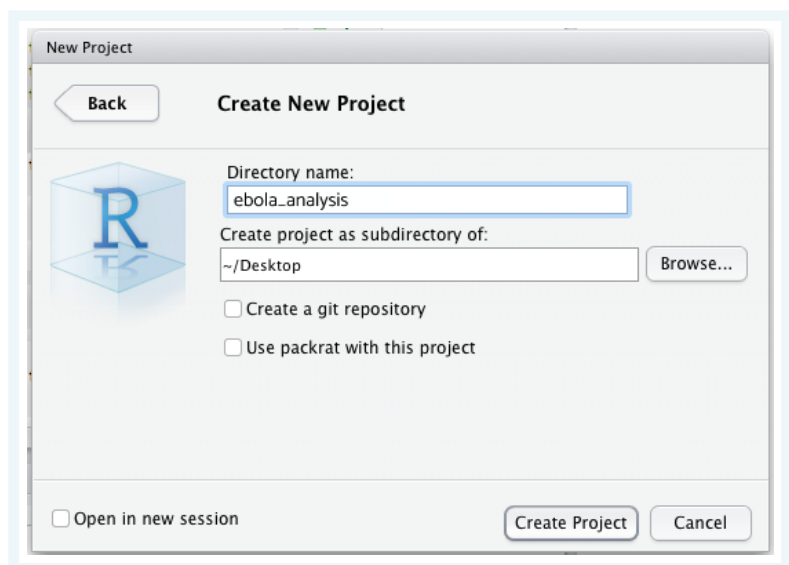
Choose “New directory”



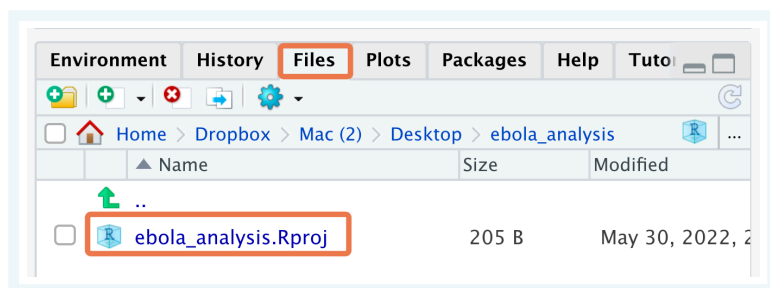
Then choose “New Project”:



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?”)



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:



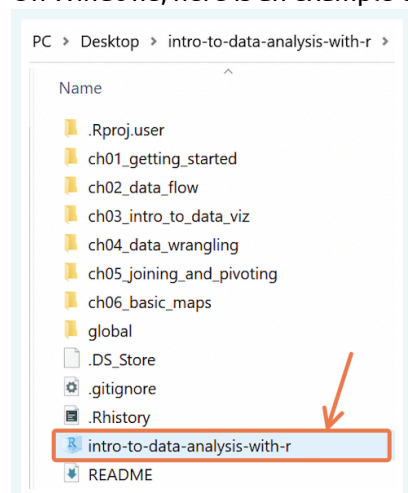
Key Point

Click on the .Rproj file to open your project

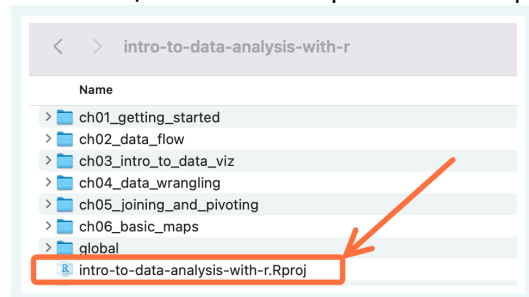
The .Rproj file is what turns a regular computer folder into an “RStudio Project”.

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

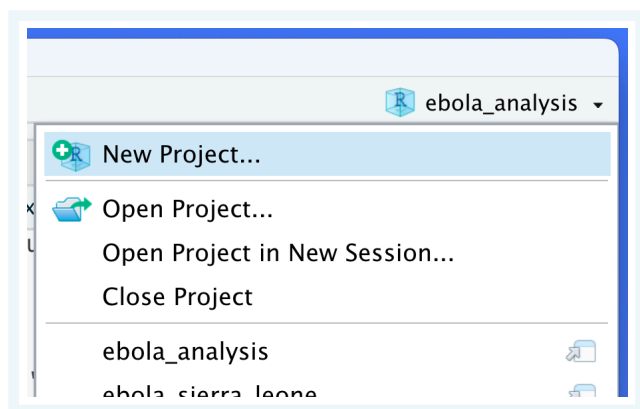
On Windows, here is an example of what a .Rproj file will look like from the File Explorer:



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

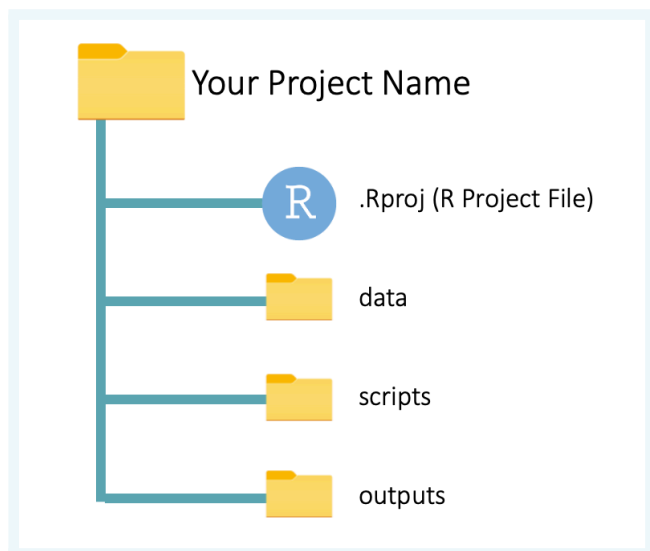


Note also that there is a header at the top right of RStudio window that tells you which Project you currently have open. Clicking on this gives you some additional Project options. You can create a new project, close a project and open recent projects, among other options.



6.4 Creating Project subfolders

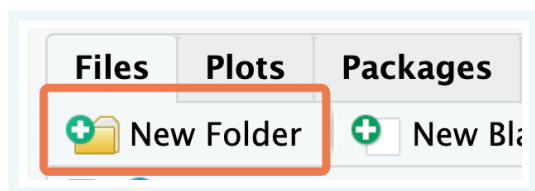
Data analysis projects usually have at least three sub-folders: one for data, another for scripts, and a third for outputs, as seen below:



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:



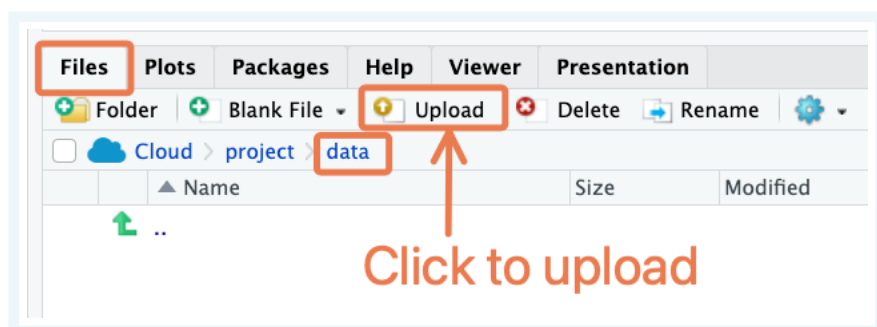
6.5 Adding a dataset to the “data” folder

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.

6.5.1 On RStudio Cloud

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

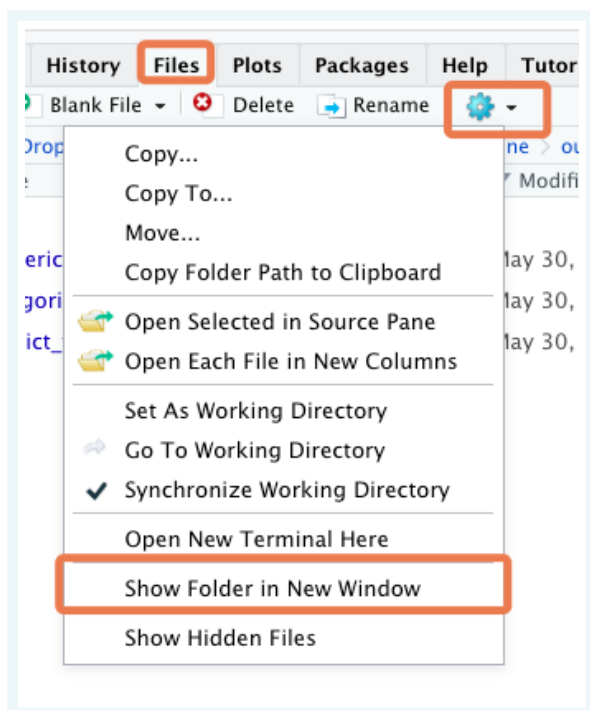


This will bring up a dialog box where you can select the file for upload.

6.5.2 On a local computer

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

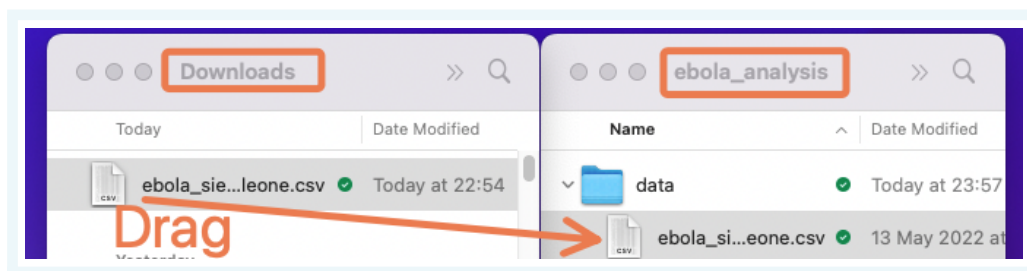
- First, locate the Project folder with your computer’s File Explorer/Finder. If you’re having trouble locating this, RStudio can help: go to the “Files” tab, click on “More” (the gear icon), then click “Show Folder in New Window”.



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

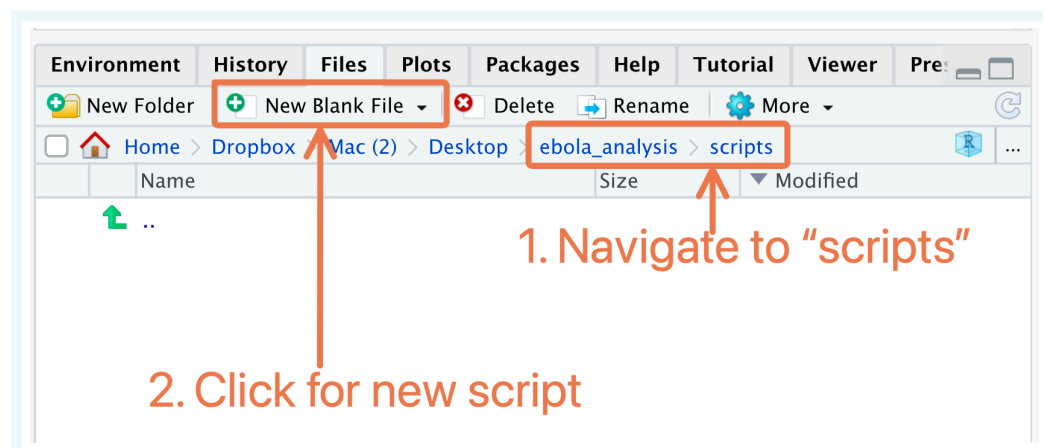
- Now, move the Ebola dataset you downloaded in the previous lesson to the newly-created "data" sub-folder.

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



6.6 Creating a script in the "scripts" folder

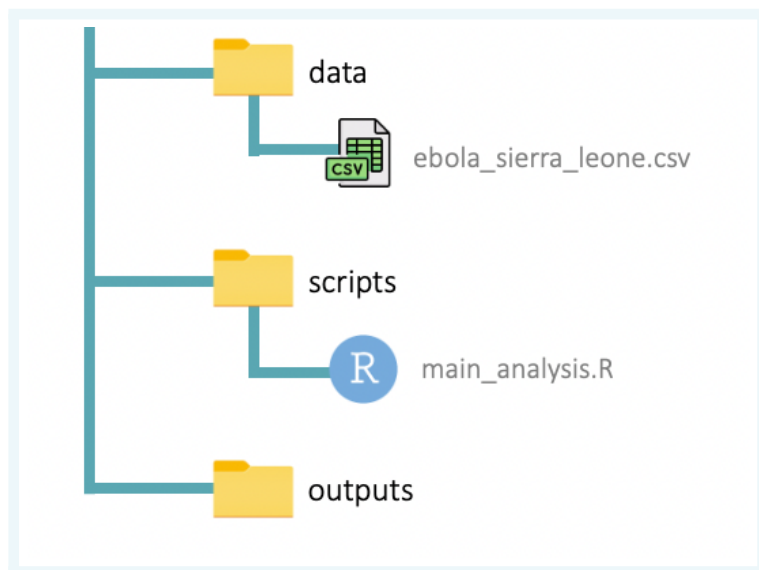
Next, create and save a new R script within the "scripts" folder. You can call this "main_analysis" or something similar. To create a new R script within a folder, first navigate to that folder in the Files pane, then click the "New Blank File" button and select "R script" in the dropdown:



i 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:

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

6.7 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.

```

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

## Load packages ----

```

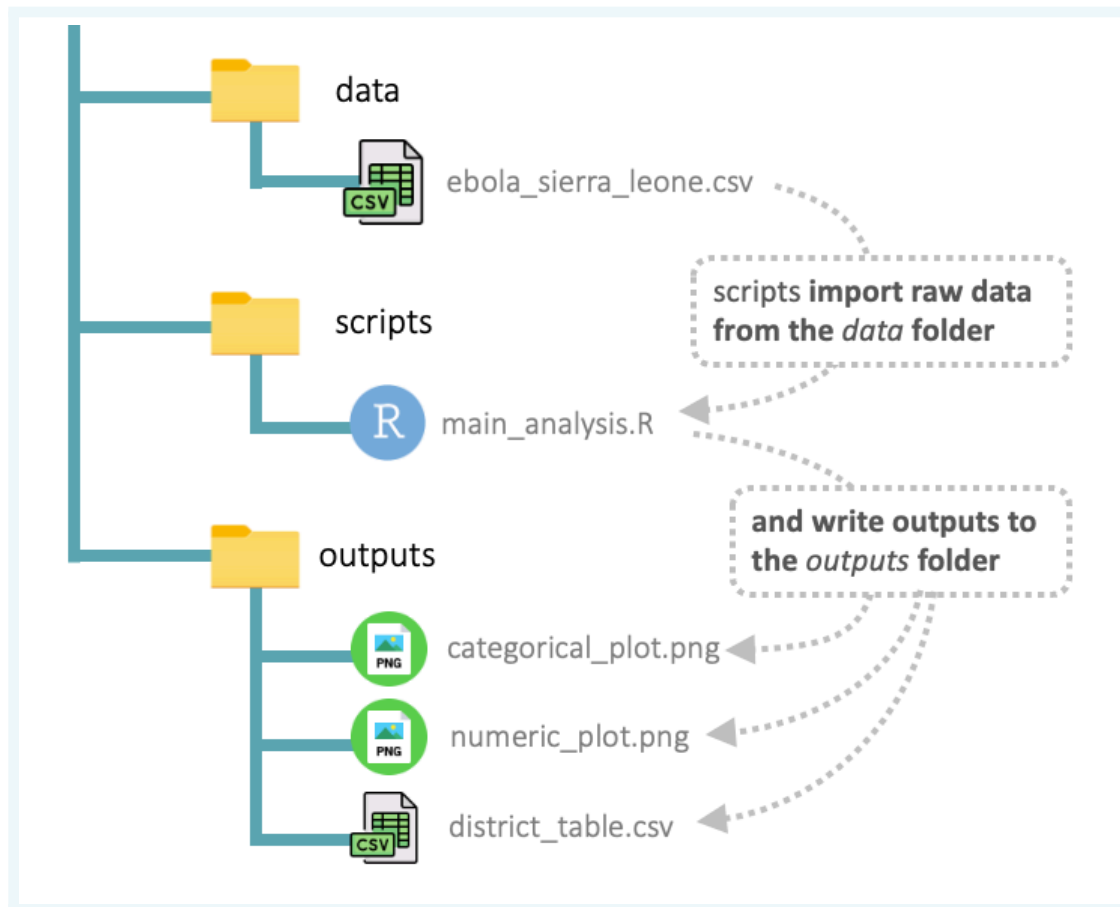


Figure 6.1: Figure: Data flow in an R project. Scripts in the “scripts” folder import data from “data” folder and export data and plots to the “outputs” folder

```

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:

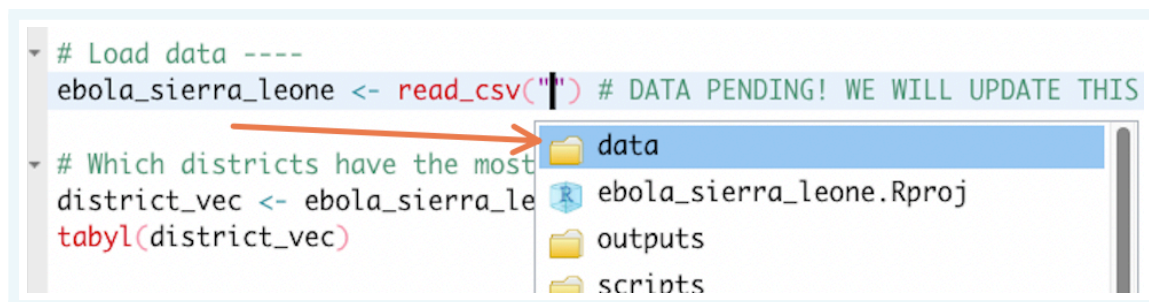
```

## 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:



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:

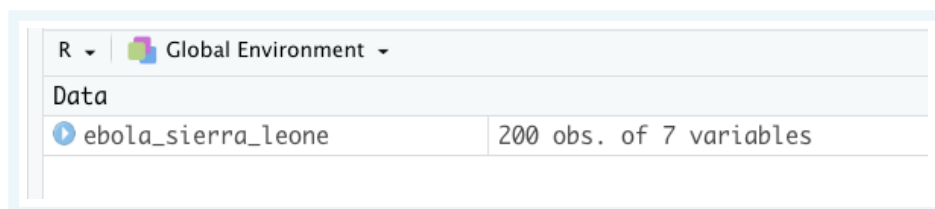
```

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:



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 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.

6.7.1 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

6.8 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"))
```

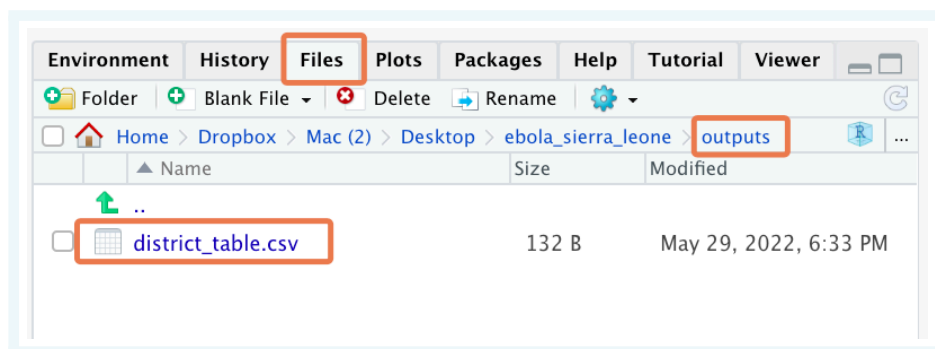
The path “outputs/district_table.csv” tells `write_csv()` to save the plot as a CSV file named “districts_table” in the “outputs” folder of the Project.

Side Note

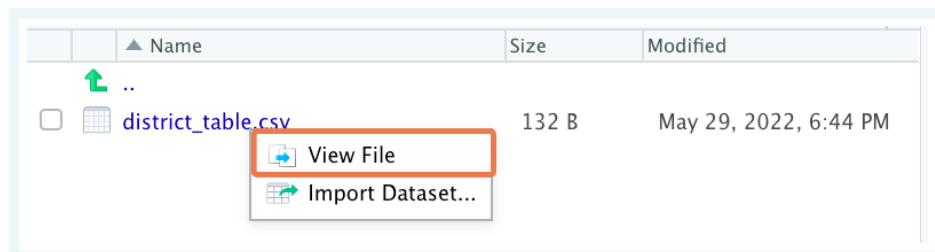
You can replace “district_table.csv” with any other appropriate name, for example “freq table across districts.csv”:

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

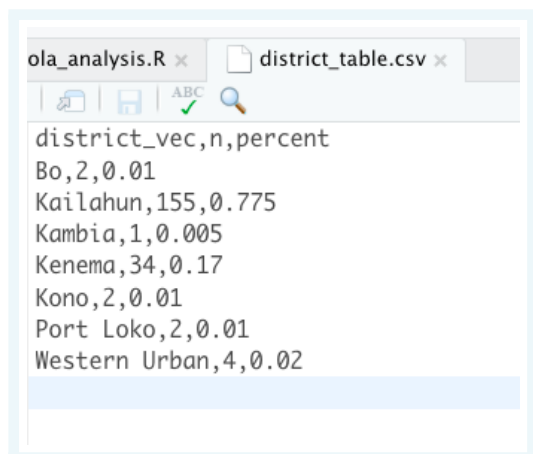
Great work! Now, if you go to the Files tab and navigate to the outputs folder of your Project, you should see this newly created file:



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



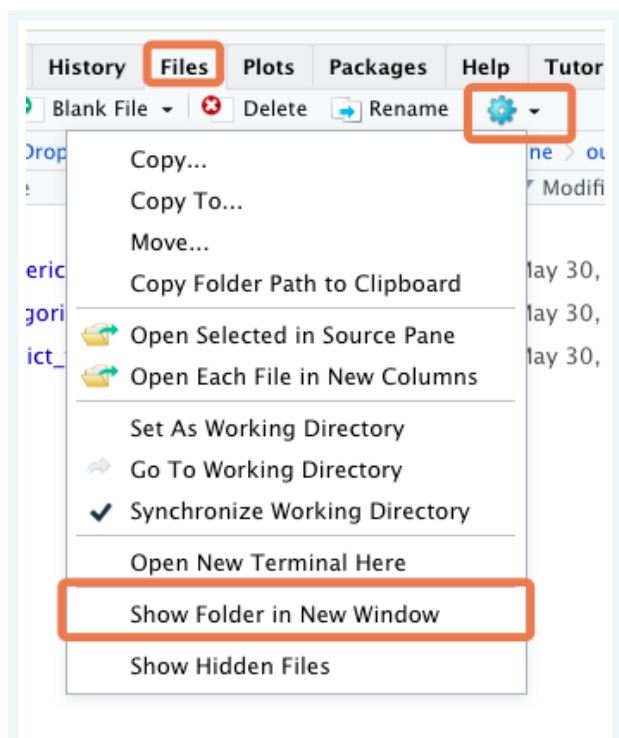
This should bring up an RStudio viewer window:



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.

i 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".



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.

6.8.1 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:

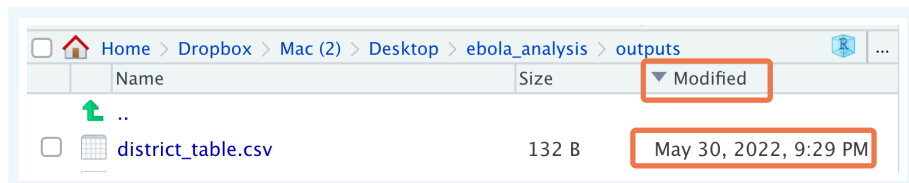
district	n	percent
Kailahun	155	0.775
Kenema	34	0.170
Western Urban	4	0.020
Bo	2	0.010
Kono	2	0.010

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:



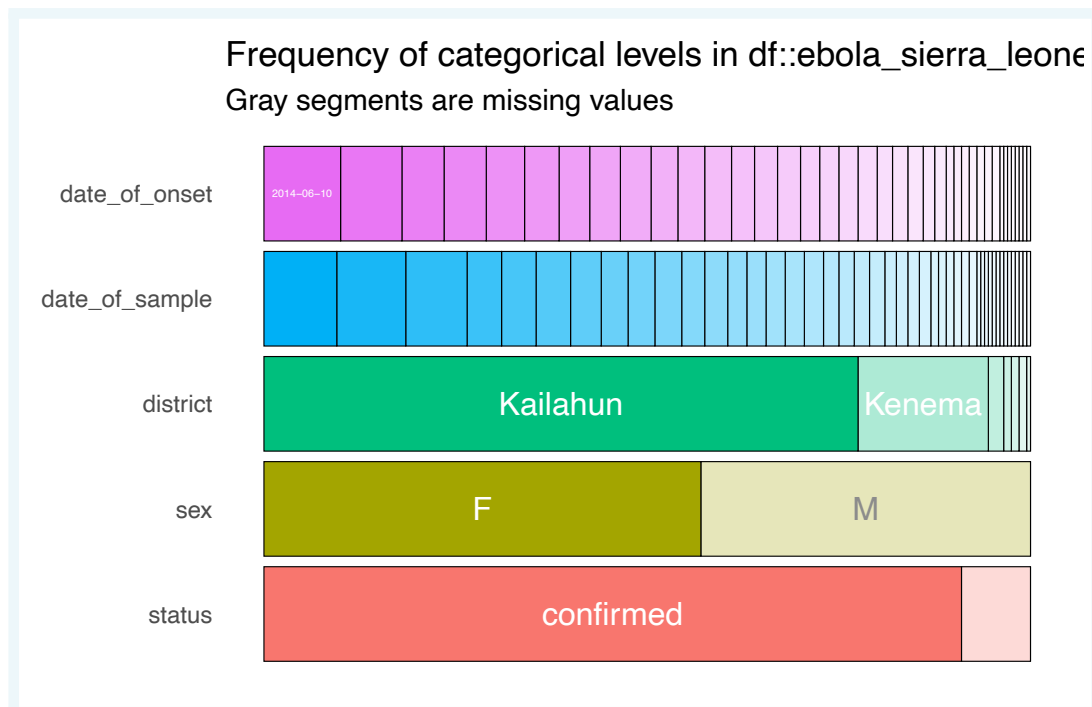
6.9 Exporting plots to the “outputs” folder

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)
```

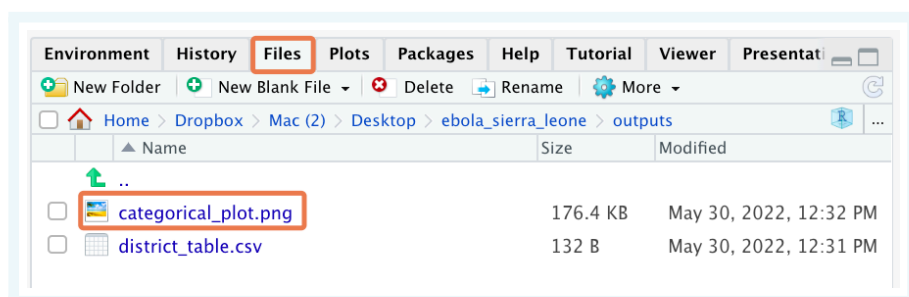
This command uses the `ggsave()` function to export the `categ_vars_plot` figure. The `plot` argument of `ggsave()` takes in the object to be saved (in this case `categ_vars_plot`), and the `filename` argument takes in the target file path for the plot.

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:



Also note that the `ggsave()` function lets you save plots to multiple image formats. For example, you could instead write:

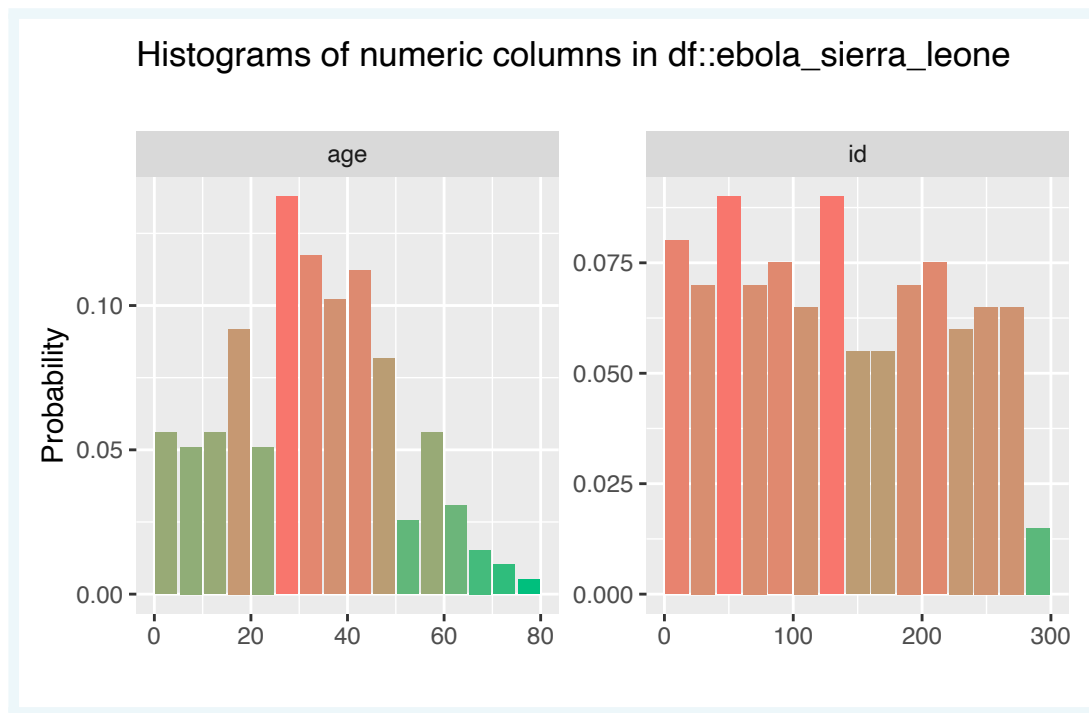
```
ggsave(filename = "outputs/categorical_plot.pdf", plot = categ_vars_plot)
```

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!

6.10 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

6.11 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.

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>

Chapter 7

R Markdown

7.1 Introduction

The `{rmarkdown}` package enables you to generate dynamic documents by combining formatted text and results produced by R code. With R Markdown, you can create documents in various formats such as HTML, PDF, Word, and many others, making it a versatile tool for exporting, communicating, and sharing your analysis results.

This document itself was created using R Markdown. While there is an entire book dedicated to R Markdown, we will cover some of the essential concepts here.

Note that working with R Markdown requires using a lot of the graphical user interface (GUI) tools in RStudio. Because of this, the written notes in this lesson will not be as detailed as in other lessons. For deeper understanding, we recommend that you follow along with the accompanying video tutorial.

Learning objectives

- Create and knit an R Markdown document that includes code and free text
- Output documents in multiple formats, including HTML, PDF, Word, PowerPoint, and flexdashboards
- Understand basic Markdown syntax
- Use R chunk options, such as *eval*, *echo*, and *message*
- Know the syntax for inline R code
- Recognize useful packages for table formatting in R Markdown
- Understand how to use the `{here}` package to set the project folder as the working directory in R Markdown files

7.2 Project setup

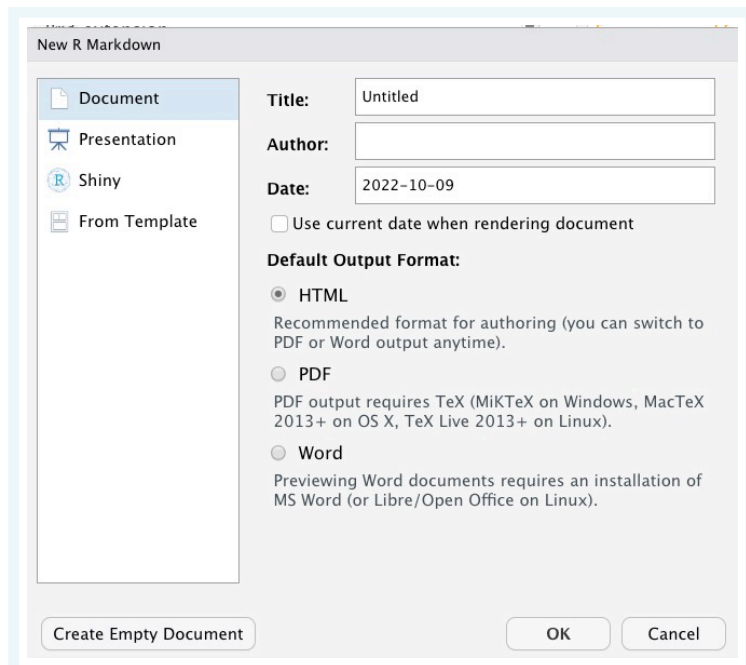
To begin, open RStudio and click on the *File* menu. Select *New Project...* and then click on *New Directory*. Choose a name for your project and specify the directory where you want to store it. Remember the location for future reference. Once you have filled out these fields, click *Create Project*.

Next, let's set up some folders within the project. In the *Files* pane, click on *New Folder* and name it "data". Click *OK*. This folder will store the project's data. Create another folder called "rmd" to store your R Markdown documents.

7.3 Create a new document

An R Markdown document is a simple text file with the `.Rmd` extension.

To create a new R Markdown document in RStudio, go to the *File* menu, choose *New file*, and then select *R Markdown....* If prompted, install the necessary packages. Once RStudio has the required packages, the following dialog box will appear:

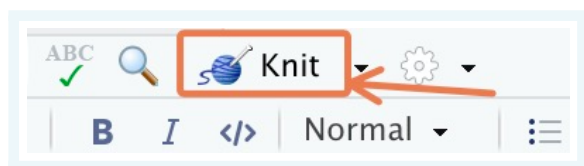


For now, keep the default values and click OK. A file with sample content will be displayed.

Experiment with editing some of the text in the file. Notice that it consists of free text and code sections.

Save your file using `Cmd/Ctrl + S`, and make sure to give it the `".Rmd"` extension. For example, `"ebola_analysis.Rmd"`. Save it in the `"rmd"` folder you created earlier.

To render the document, click on the `"knit"` button at the top right:



This will generate 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:



The rendered file will be stored in the same directory as your Rmd file, with the same name but ending in ".html" instead of ".rmd".

Vocab

HTML stands for Hypertext Markup Language and is the standard format used for most documents on the web.

7.4 R Markdown Header (YAML)

Let's return to the rest of the Rmd file and examine it part by part.

The first part of the document is its *header*, also known as "YAML" (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, and various options that allow you to configure and customize the entire document and its rendering. For example, the line `output: html_document` specifies that the generated document should be in HTML format.

You can change the `html_document` text to experiment with other formats.

7.4.1 Word Document

If you set the output to “`word_document`”, and click to `knit` the file, the rendered document will look like this:

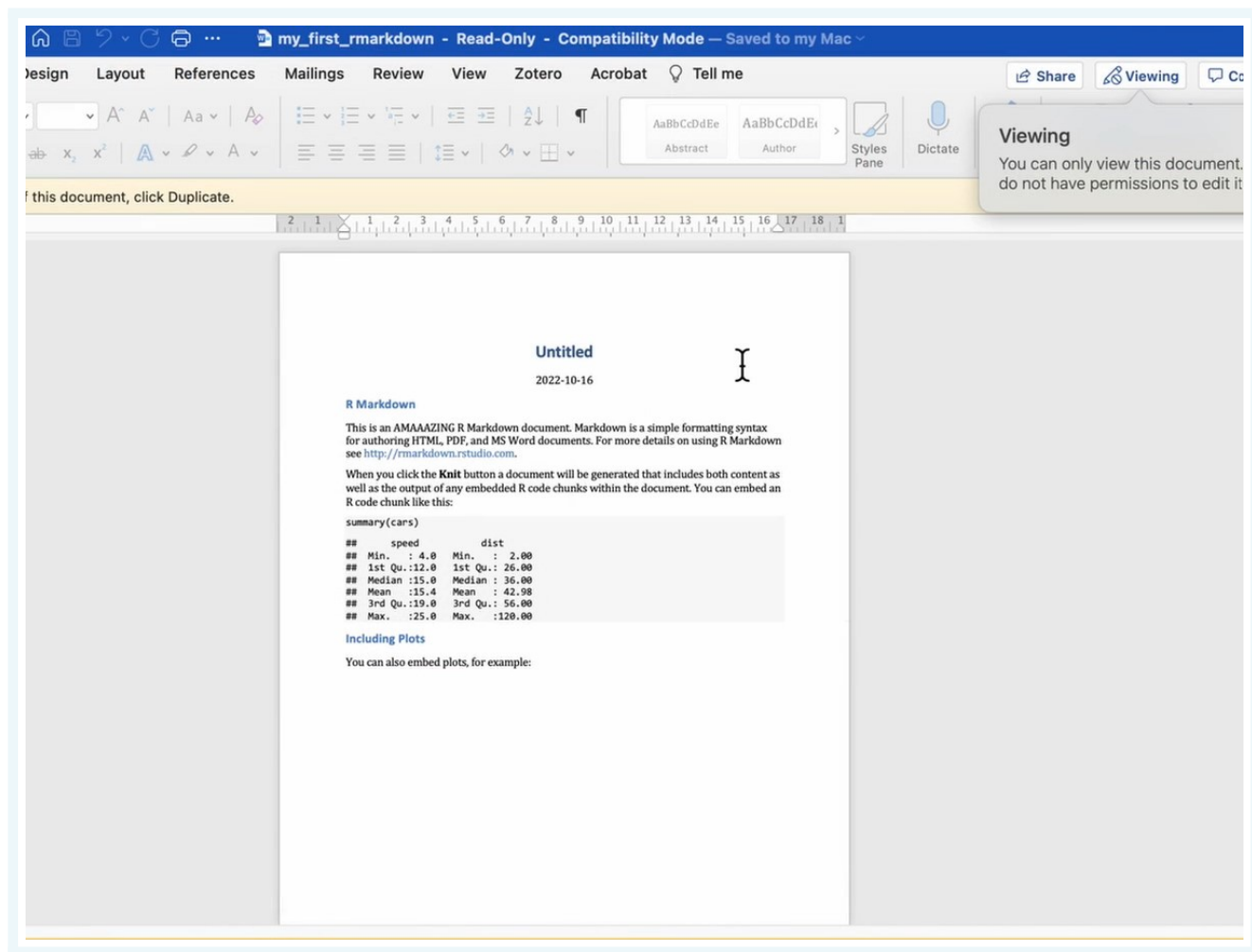


Figure 7.1: Image of the R Markdown document open in the Microsoft Word program

A “.docx” version of your document will be created in the “rmd” folder.

7.4.2 PowerPoint Document

When the output is set to “`powerpoint_document`”, the result will be:

7.4.3 PDF Document

If you change the output setting to “`pdf_document`”, you can obtain the same document in PDF format (you may be prompted to install `tinytex` on your computer, see below):

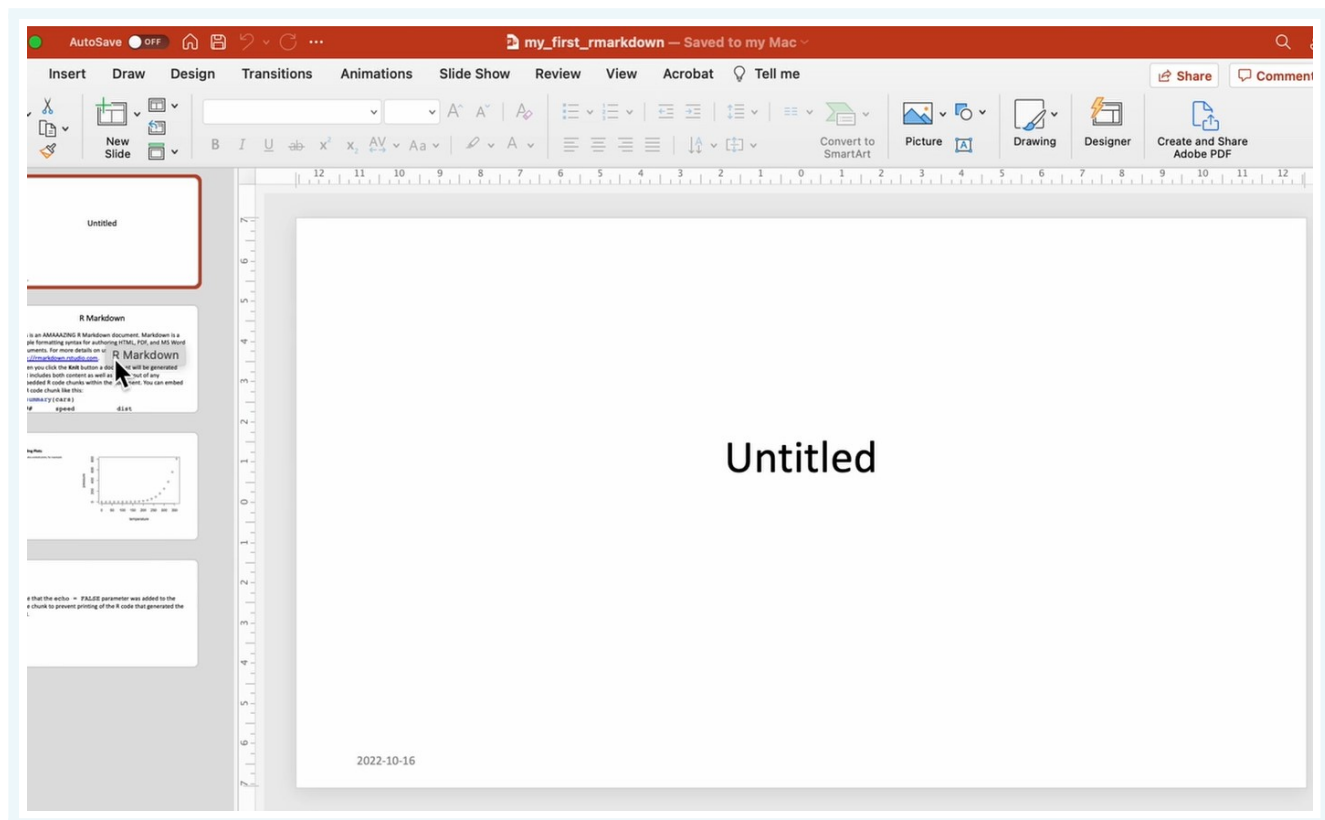
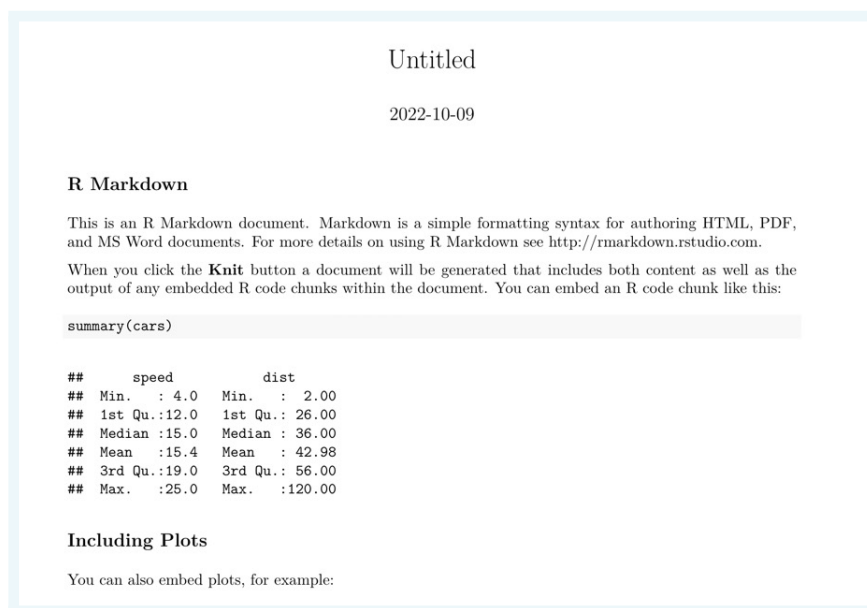


Figure 7.2: Image of the R Markdown document open in the Microsoft PowerPoint program



i Key Point

For PDF generation, you must have a working LaTeX installation on your system. If not, Yihui Xie's `tinytex` extension aims to simplify the installation of a minimal LaTeX distribution regardless of your machine's operating system.

To use it, 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 is available on [the tinytex website](https://yihui.org/en/tinytex/).

7.4.4 Prettydoc

To try the “prettydoc” format, type `install.packages('prettydoc')` into the console and press *Enter*. The output format for prettydoc is slightly different from the previous three. You need to use `prettydoc::html_pretty` in the output section. When you knit a prettydoc, you should see something like this:

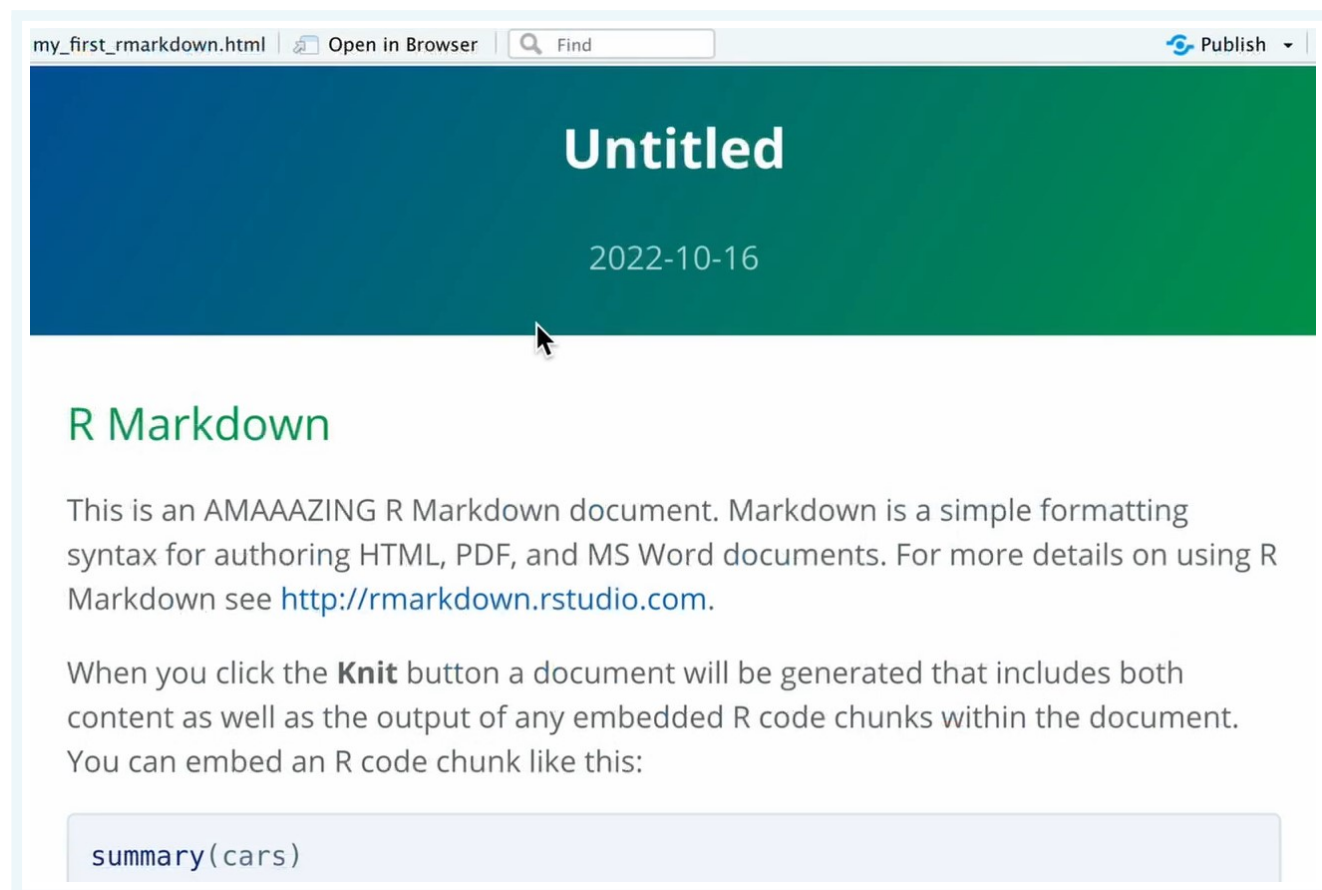


Figure 7.3: Image of the R Markdown document as a prettydoc

7.4.5 Flexdashboard

You can even create a simple dashboard format. First, run `install.packages('flexdashboard')`. Then, set the output to `flexdashboard::flex_dashboard` and knit. The result will be similar to the following:

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 modify the header style for those sections, and flexdashboard will render those headers as tabs.

Many other formats are available, and we encourage you to explore them on your own!

7.5 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. There is a pair of buttons to toggle between the modes:

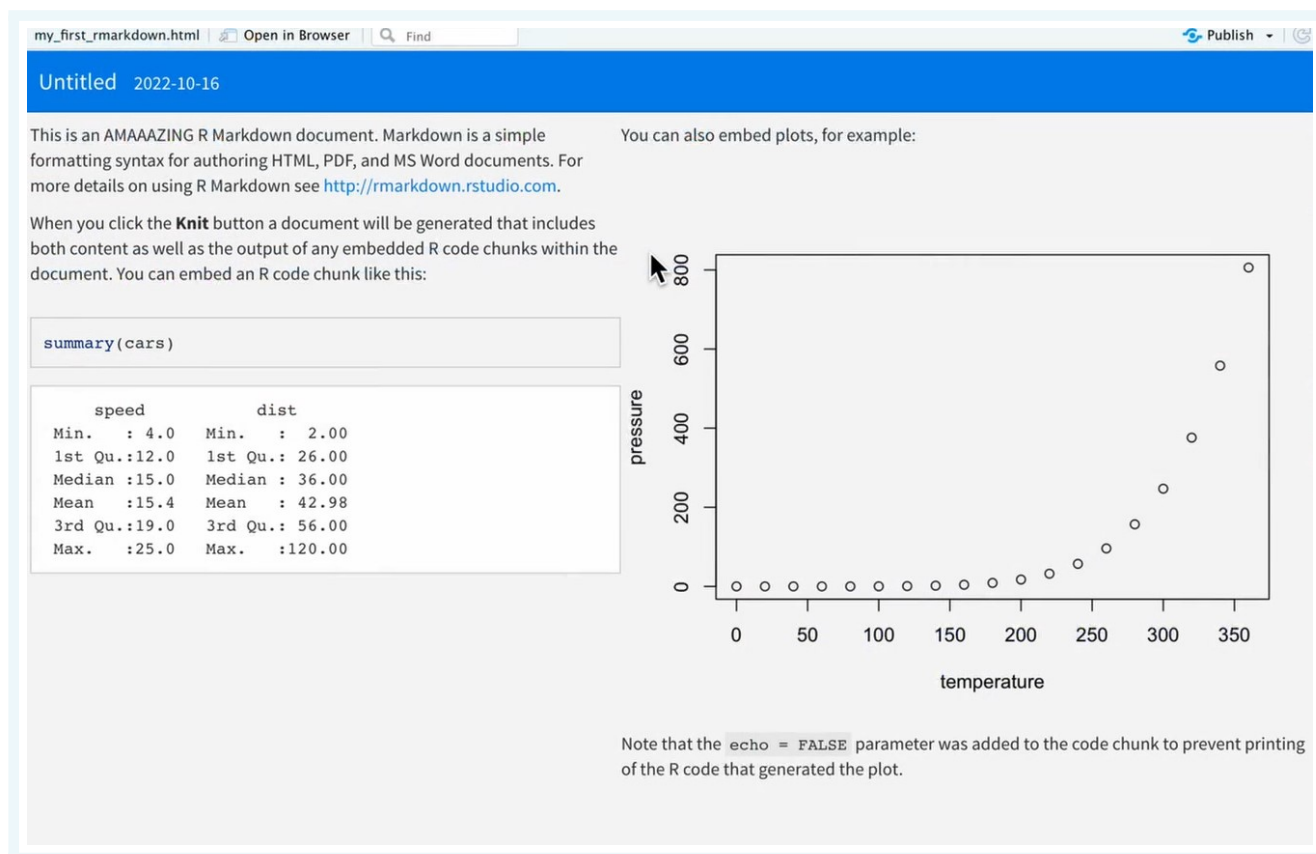
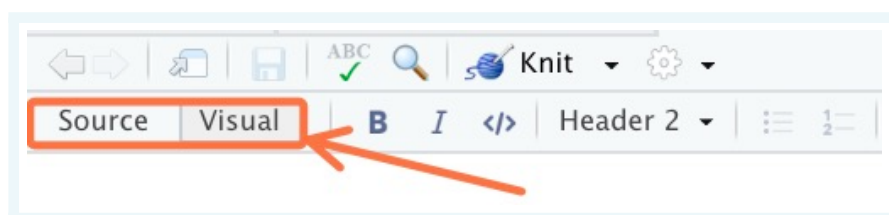


Figure 7.4: Image of the R Markdown document as a flexdashboard



What's the difference between these two modes?

In source mode, you see the raw markdown syntax.

i Vocab

Markdown is a simple set of conventions for adding formatting to plain text. For example, to italicize text, you wrap it in asterisks `*text here*`, and to start a new header, you use the pound sign `#`. We will learn these in detail below.

In visual mode, you see a Microsoft Word-like view with a toolbar for easy formatting.

This means you don't have to 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.

While visual mode is much easier to use, we will teach you markdown syntax here for three reasons:

1. Visual mode can sometimes be buggy, and to debug this, you'll need to switch to source mode.
2. Understanding markdown syntax is useful outside of Rmarkdown.
3. Visual mode is not available in RStudio's collaborative mode, which you may want to use.

7.6 Markdown syntax

In the “Help” tab of the top RStudio menu, if you look up “Markdown Quick Reference”, you will find a wide variety of RMD options available.

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 to help 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. Otherwise, they will all crunch together rather than making a list.

We see that words placed between asterisks are italicized, and lines that begin with a dash are transformed into a bulleted list.

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](https://example.com)

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. In *Visual* mode, you can 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. Alternatively, 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. 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, clicking on the *Show document outline* icon on the far right of the toolbar associated with the R Markdown file will display a table of contents automatically generated from the titles, allowing for easy navigation within the document:

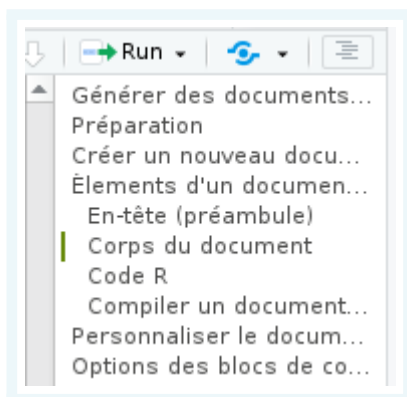


Figure 7.5: Dynamic TOC

7.6.1 Customizing the generated document

The generated document can be customized by modifying options in the document's preamble. RStudio offers a graphical interface to change these options more easily. To access it, click on the gear icon to the right of the *Knit* button and choose *Output Options...*

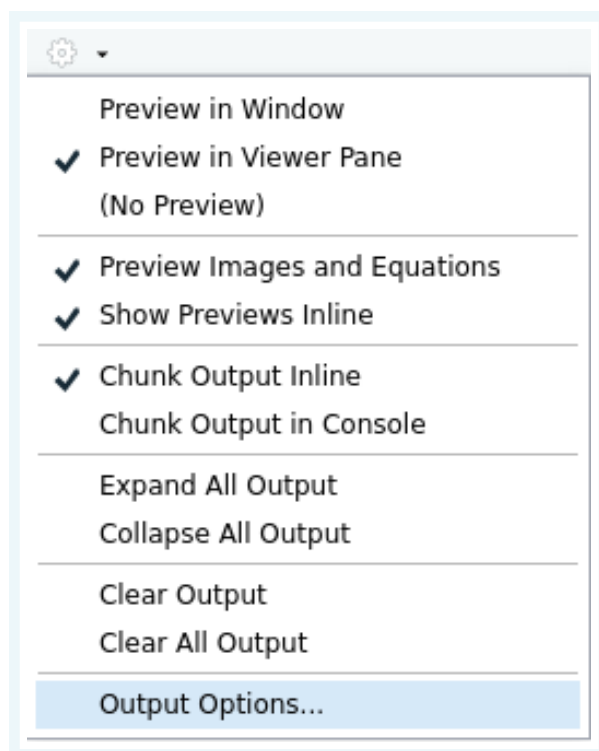


Figure 7.6: R Markdown Output Options

A dialog box will appear, allowing you to select the desired output format and various options depending on the format:

For example, with the HTML format, 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 generated graphics.

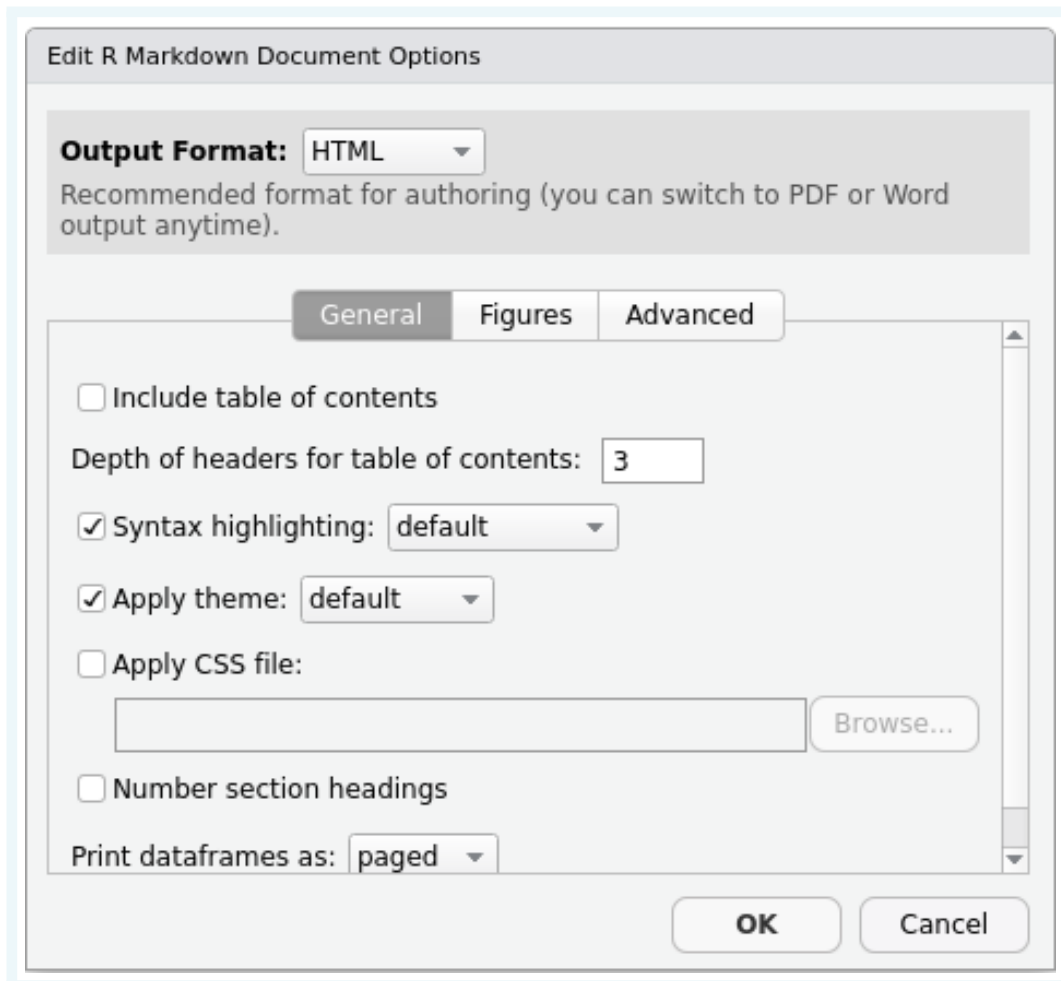


Figure 7.7: R Markdown Output Options Dialog

When you change options, RStudio will modify the preamble of your document. For instance, 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
    toc: yes
---
```

You can also 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
    toc: yes
  pdf_document:
    fig_caption: yes
    highlight: kate
---
```

The complete list of possible options is available on [the official documentation site](#) (which is very comprehensive and well-made) and on the cheat sheet and reference guide, accessible from RStudio via the *Help* menu, then *Cheatsheets*.

7.7 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.

In RStudio blocks of R code are usually displayed with a slightly different background color to distinguish them from the rest of the document.

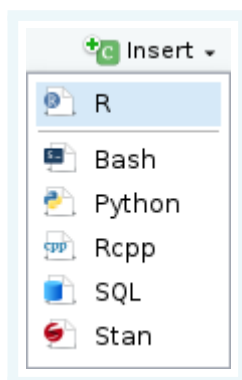


Figure 7.8: Code block insertion menu

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.

7.7.1 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*.

7.7.2 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.

### 7.7.3 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.

### 7.7.4 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
<code>echo</code>	<code>TRUE/FALSE</code>	Show (or hide) this R code chunk in the resulting knitted document
<code>eval</code>	<code>TRUE/FALSE</code>	Run (or not) the code in this code chunk in the resulting knitted document
<code>include</code>	<code>TRUE/FALSE</code>	Combines the options “ <code>echo</code> and <code>eval</code> ”; either show and run, or hide and don’t run
<code>message</code>	<code>TRUE/FALSE</code>	Show (or hide) any system messages generated by running this code chunk in the resulting knitted document
<code>warning</code>	<code>TRUE/FALSE</code>	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).

### 7.7.5 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:

You can then modify the most common options, and click on *Apply* to apply them.

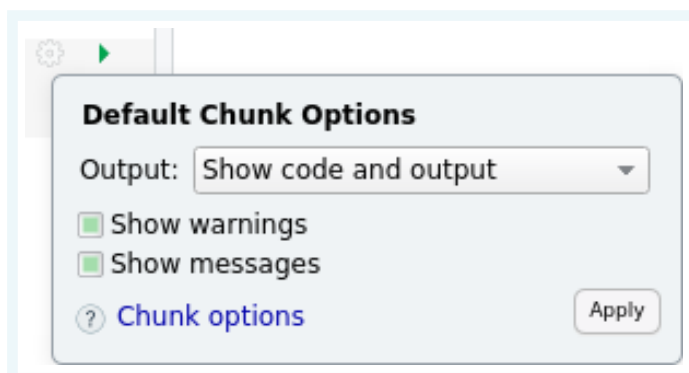


Figure 7.9: Code Block Options Menu

### 7.7.6 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)
```

7.8 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.

7.9 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

height	weight
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)
```

	height	weight
	58	115
	59	117
	60	120
	61	123
	62	126
	63	129
	64	132
	65	135
	66	139
	67	142
1–10 of 15 rows		
	Previous	1 2 Next

You can see many other types of table formats people have created at <https://www.rstudio.com/blog/rstudio-table-contest-2022/>

7.10 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.

7.10.1 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:

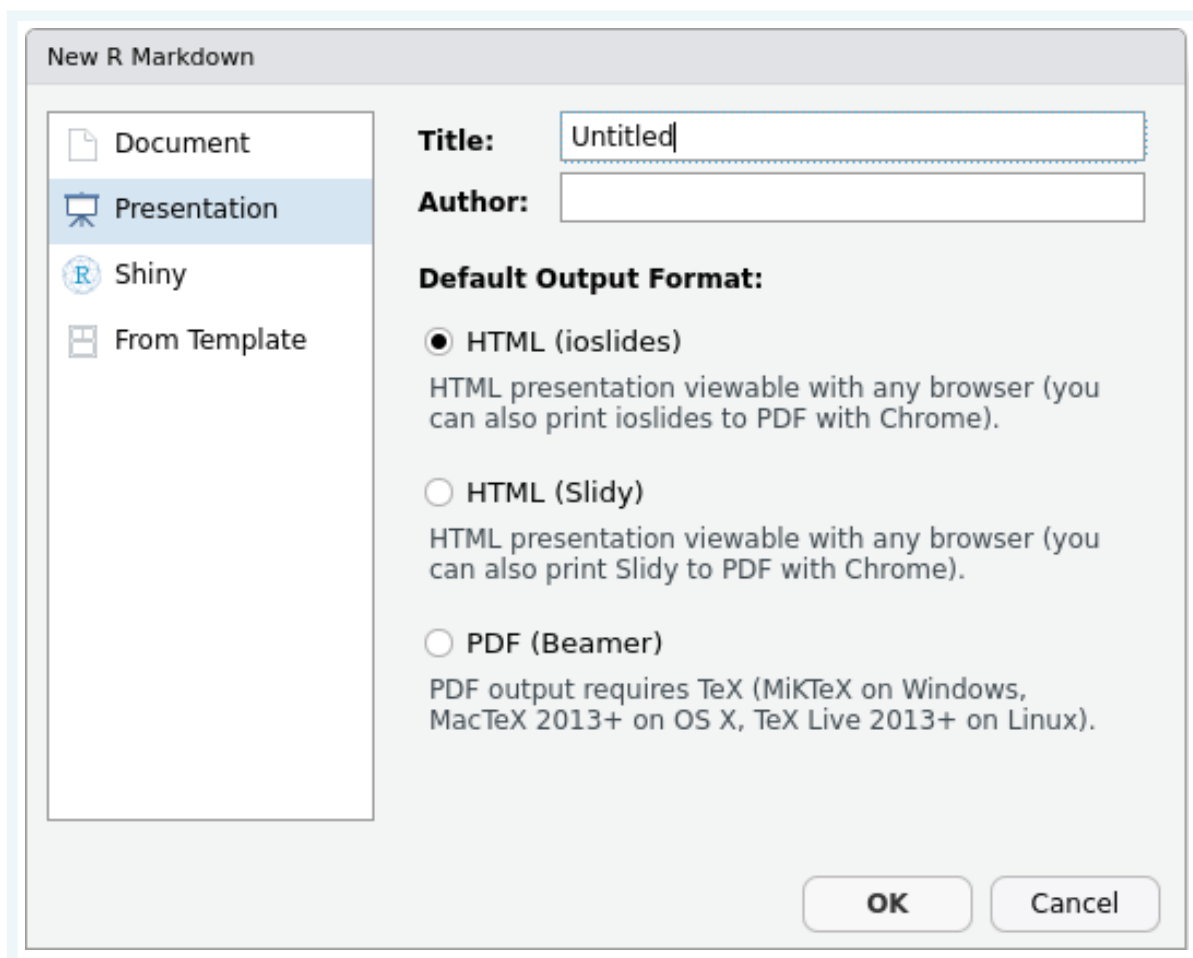


Figure 7.10: Create an R Markdown presentation

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.

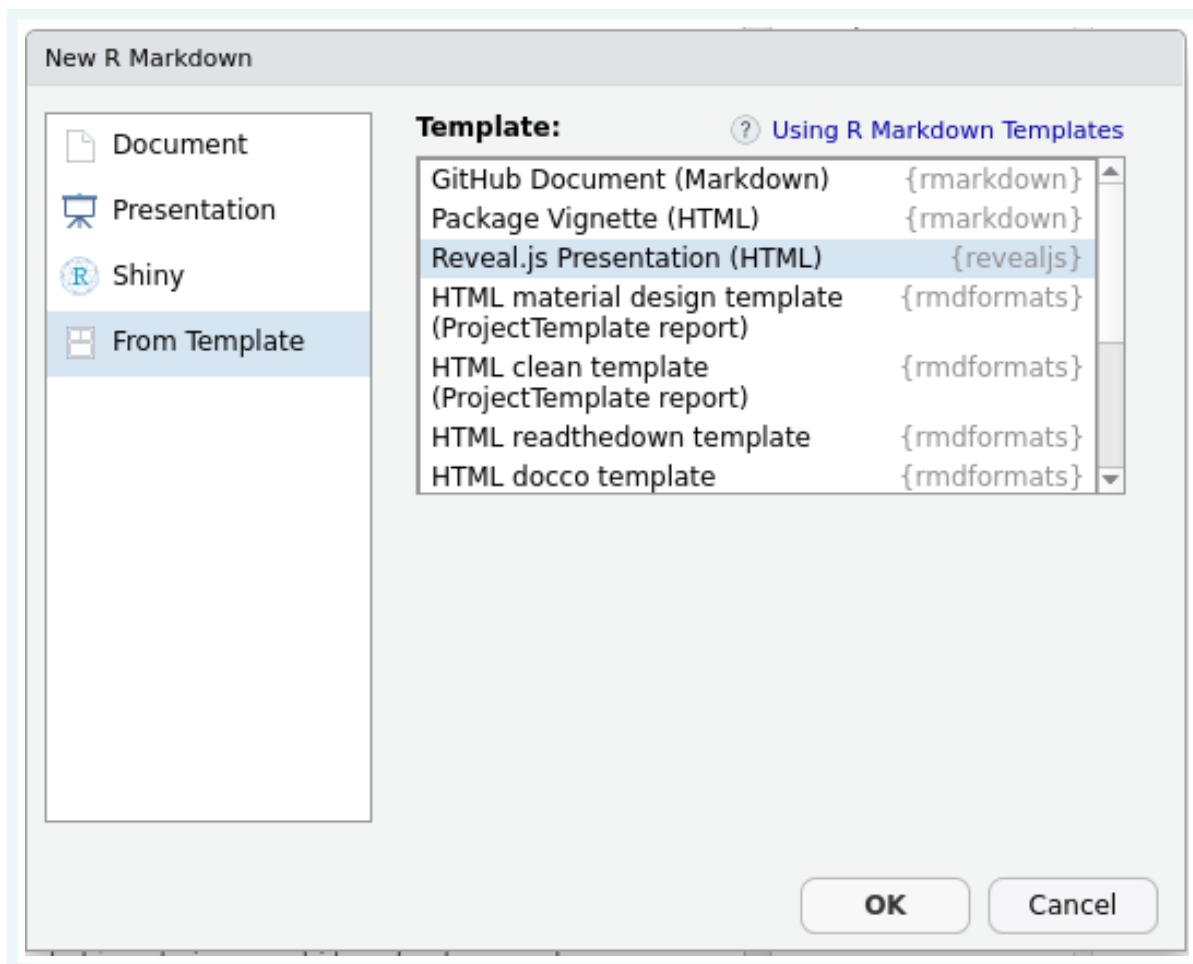


Figure 7.11: Create a presentation from a template

7.10.2 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*):

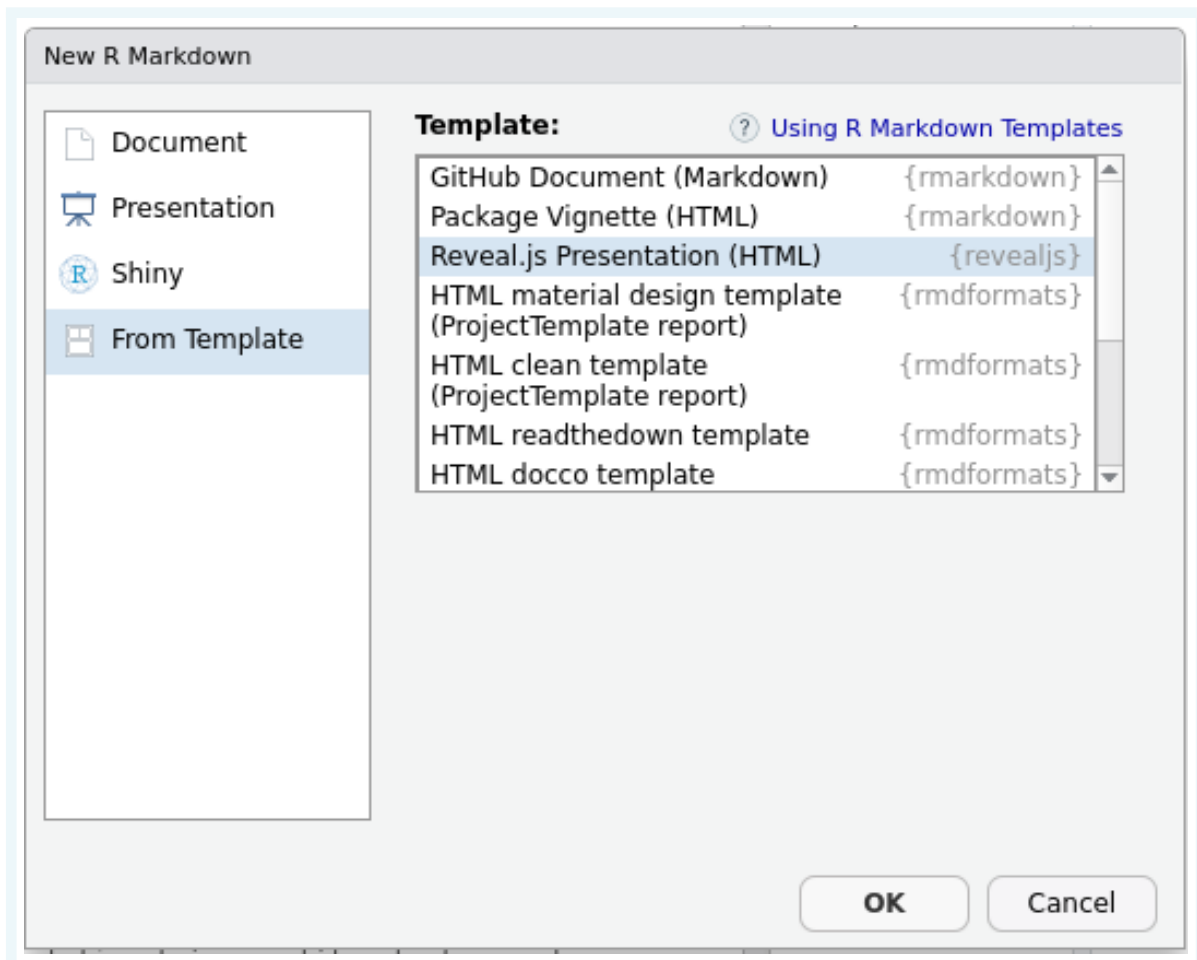


Figure 7.12: Create a document from a template

7.11 Resources

Below are some resources to help you learn more about R Markdown:

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*).

Chapter 8

Data structures

8.1 Intros

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.

8.2 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.

8.3 Packages

Please load the packages needed for this lesson with the code below:

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

8.4 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.

8.5 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"
```

Practice

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.

8.6 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 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.

8.7 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
```

Practice

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)
```

Practice

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

8.8 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 x 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.

8.8.1 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()`.

8.9 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.

8.10 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))
```

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>.

Chapter 9

Using ChatGPT for Data Analysis

9.1 Introduction

ChatGPT, developed by OpenAI, is a language model that can be used to assist data analysts in various tasks. It can:

1. Explain unfamiliar code
2. Debug simple errors
3. Add code comments
4. Reformat code
5. Generate plots
6. Perform simple data wrangling
7. Translate languages
8. Translate programming languages
9. Perform a fluid find and replace operation

Let's get started and see how it works!

9.2 Models

As at our last check, ChatGPT comes in two main versions, both available at chat.openai.com:

Model	Availability	Speed	Task Complexity
GPT-3.5	Free	Fast	Simple tasks
GPT-4	Paid / Bing	Slower	Complex tasks

Both models have their advantages and use cases. GPT-3.5 is suitable for simple tasks and is free to use, while GPT-4, though slower and paid, can handle more complex tasks.

Bing's AI chat service also uses GPT-4 under the hood (last checked in September 2023), so you can go to Bing if you want GPT-4 access without paying. The Bing interface is a bit clunky for data analysis tasks though, so if you can afford to pay for GPT-4, we highly recommend it.

9.3 1. Explain Unfamiliar Code

ChatGPT can explain unfamiliar code. For instance, if you encounter the following code:

```
trees %>%  
  select(Height) %>%  
  filter(Height > 80)
```

You can ask ChatGPT, “What is this code doing?” and paste in the code.

Here is ChatGPT3.5’s response:

The code you’ve provided appears to be written in R and is likely using the dplyr package for data manipulation. Here’s a step-by-step explanation of what this code does:

1. `trees %>%`: This part of the code uses the `%>%` operator, also known as the pipe operator, which is used to chain together multiple data manipulation operations in a more readable and concise manner. It takes the data frame or tibble `trees` and passes it as the first argument to the next operation.
2. `select(Height) %>%`: After the pipe operator, the `select()` function is used. It selects a subset of columns from the data frame. In this case, it selects only the “Height” column from the `trees` data frame.
3. `filter(Height > 80)`: Following the `select()` operation, the `filter()` function is used. This function filters rows based on a specified condition. In this case, it filters the data to keep only those rows where the “Height” column has a value greater than 80.

So, in summary, the code takes the `trees` data frame, selects only the “Height” column, and then filters the rows to retain only those where the height is greater than 80. The result will be a subset of the original data containing only rows where the tree height is above 80.

9.4 2. Debug Simple Errors

ChatGPT can assist in debugging straightforward errors. You can ask ChatGPT, “Please help me understand why the code is not working,” then paste in both your code and the error message.

For example:

```
data_subset_I <-  
  trees %>%  
  select(Height) %>%  
  filter(Height > 80)  
  
summary(data_subset_1)
```

Error message:

```
Error in summary(data_subset_I) : object 'data_subset_I' not found
```

ChatGPT3.5 response:

It seems like you have a typo in your code. In the code snippet you provided, you are creating a subset of data and storing it in the variable `data_subset_I`, but then you are trying to summarize a different variable

data_subset_1 (note the lowercase “l” at the end). The variable names are not consistent, which is likely causing the issue.

You should use the correct variable name data_subset_I when summarizing the data. Here’s the corrected code:

```
data_subset_I <-  
  trees %>%  
  select(Height) %>%  
  filter(Height > 80)  
  
summary(data_subset_I)
```

With this corrected code, it should work as expected and summarize the data_subset_I data frame, which contains the heights of trees greater than 80.

9.5 3. Add Code Comments

ChatGPT can also assist in adding comments to your code. For instance, you can say, “Add comments to the following code,” then paste in your code.

```
data_subset_2 <-  
  trees %>%  
  select(Height) %>%  
  filter(Height > 80)  
  
summary(data_subset_2)
```

ChatGPT3.5 response:

Certainly! Here’s the code with comments added to explain each step:

```
## Select the 'Height' column from the 'trees' dataset  
## and create a new dataset 'data_subset_2'  
data_subset_2 <-  
  trees %>%  
  select(Height) %>%  
  
## Filter 'data_subset_2' to retain only rows where 'Height' is greater than 80  
  filter(Height > 80)  
  
## Display a summary of 'data_subset_2'  
## including statistics like mean, median, and quartiles  
summary(data_subset_2)
```

These comments provide a clear explanation of what each line of code does, making it easier to understand the data manipulation and summary generation process.

9.6 4. Reformat Code

ChatGPT can help in reformatting code. For example, consider the following code:

```
temp<- trees %>% select(Height) %>%filter(Height > 80)
```

You can ask ChatGPT, “Help me reformat this code nicely,” then paste in the code.

ChatGPT3.5 response:

Certainly, here’s the reformatted code:

```
temp <- trees %>%
  select(Height) %>%
  filter(Height > 80)
```

I’ve added line breaks and indentation to make the code more readable and visually organized.

9.7 5. Make Plots

ChatGPT can assist in creating plots with ggplot2. For instance, you might ask:

“Use ggplot2 to create a scatter plot of height against girth for the dataset shown” then paste in the head of the data.

```
head(trees)
```

	Girth	Height	Volume
1	8.3	70	10.3
2	8.6	65	10.3
3	8.8	63	10.2
4	10.5	72	16.4
5	10.7	81	18.8
6	10.8	83	19.7

ChatGPT3.5 response:

To create a scatter plot of height against girth for the dataset you provided using ggplot2 in R, you can follow these steps:

1. First, make sure you have the ggplot2 package installed. If you don’t have it installed, you can install it using the following command:

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

2. Load the ggplot2 library:

```
library(ggplot2)
```

3. Create the scatter plot using the ggplot() function and the geom_point() function to add the points:

```
## Assuming you have already loaded the dataset
## If not, you can load it using data(trees)

## Create the scatter plot
```

```
ggplot(trees, aes(x = Girth, y = Height)) +
  geom_point() +
  labs(x = "Girth", y = "Height") +
  ggtitle("Scatter Plot of Height vs. Girth")
```

This code will create a scatter plot of height against girth using the `trees` dataset, with girth values on the x-axis and height values on the y-axis. Adjust the labels and title as needed for your specific requirements.

You can then ask ChatGPT a follow-up question like:

- Add a best-fit line and write the equation of the line in the caption of the plot.
- Mark the mean height and mean girth on the plot.
- Center the plot title.

Here we may see the difference between GPT3.5 and 4.

ChatGPT3.5 response:

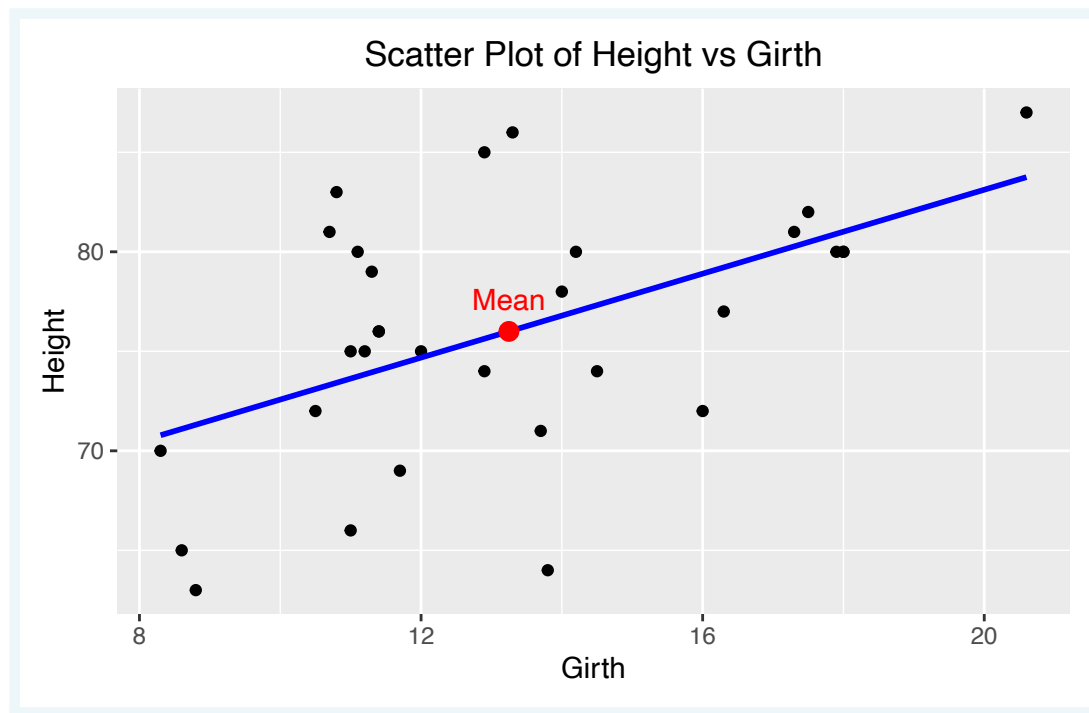
```
scatter_plot <- ggplot(trees, aes(x = Girth, y = Height)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE, color = "blue") +
  labs(x = "Girth", y = "Height") +
  ggtitle("Scatter Plot of Height vs Girth") +
  theme(plot.title = element_text(hjust = 0.5))

## Calculate mean values
mean_girth <- mean(trees$Girth)
mean_height <- mean(trees$Height)

## Add mean points
scatter_plot <- scatter_plot +
  geom_point(data = data.frame(Girth = mean_girth, Height = mean_height),
    color = "red", size = 3) +
  annotate("text", x = mean_girth, y = mean_height, label = "Mean",
    color = "red", vjust = -1)

## Print the scatter plot
print(scatter_plot)
```

``geom_smooth()`` using formula = 'y ~ x'



In the ChatGPT3.5 response, the equation of the line is missing from the caption.

Here is GPT4's response:

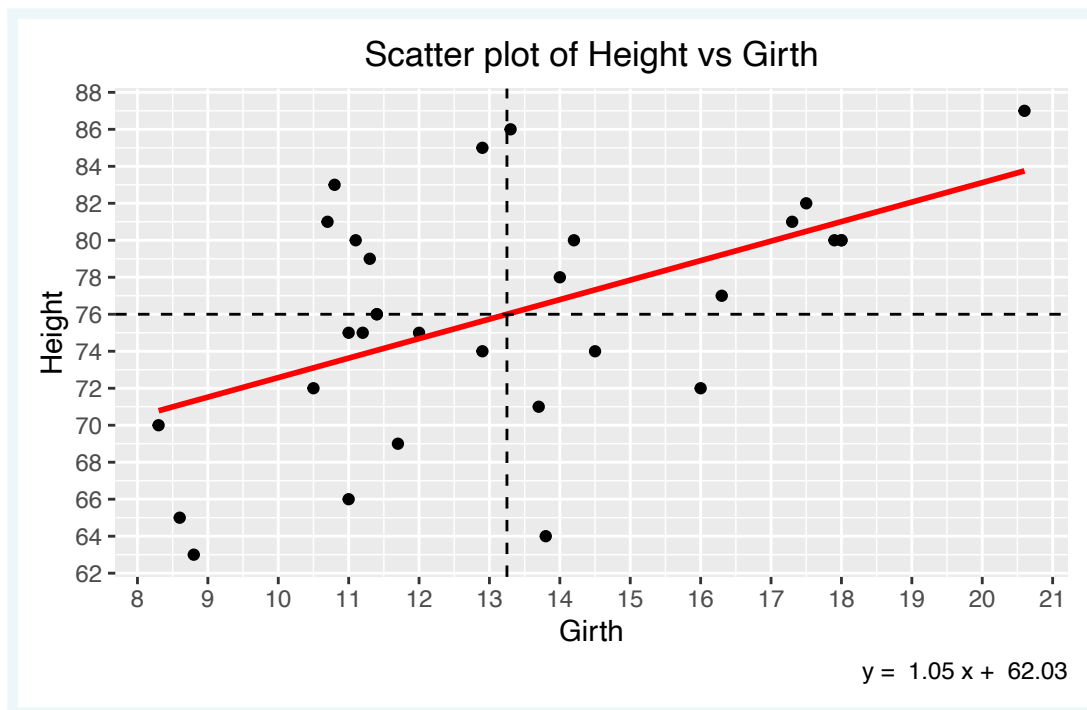
```
library(tidyverse)
library(scales)

## Calculate means
mean_girth <- mean(trees$Girth)
mean_height <- mean(trees$Height)

## Fit a linear model to calculate the equation of the line
fit <- lm(Height ~ Girth, data = trees)
slope <- coef(fit)[2]
intercept <- coef(fit)[1]
eq <- paste("y = ", round(slope, 2), "x + ", round(intercept, 2))

## Generate the plot
p <- ggplot(trees, aes(x = Girth, y = Height)) +
  geom_point() +
  geom_smooth(method = 'lm', se = FALSE, color = "red",
             formula = y ~ x) + # Add best-fit line (linear model)
  geom_vline(xintercept = mean_girth, linetype="dashed") +
  geom_hline(yintercept = mean_height, linetype="dashed") +
  ggtitle("Scatter plot of Height vs Girth") +
  theme(plot.title = element_text(hjust = 0.5)) + # Center title
  scale_x_continuous(breaks = pretty_breaks(n = 10)) +
  scale_y_continuous(breaks = pretty_breaks(n = 10)) +
  labs(x = "Girth",
       y = "Height",
       caption = eq) # Add equation to the caption
```

p



With GPT-4, you get exactly what you want, including the equation in the caption at the bottom right of the plot.

9.8 6. Simple Data Wrangling

ChatGPT can help with simple data wrangling tasks. For instance:

Consider the following dataset:

```
head(trees)
```

	Girth	Height	Volume
1	8.3	70	10.3
2	8.6	65	10.3
3	8.8	63	10.2
4	10.5	72	16.4
5	10.7	81	18.8
6	10.8	83	19.7

You might ask, “Give me dplyr code to find the mean and median girth, height, and volume.”

ChatGPT3.5 gives us this code:

```
## Use summarise_all to compute mean and median for each column
result <- trees %>%
  summarise_all(
    .funs = list(mean = mean, median = median),
```

```
na.rm = TRUE
)

print(result)
```

9.9 7. Translate Language

ChatGPT can help translate text in your tutorial or documentation from one language to another, while preserving the code. If you use other translators, they often translate your code as well, which is undesirable.

For example, you can ask:

“Translate the following section of a tutorial into Spanish”

```
dplyr is part of the core of tidyverse, so it is automatically loaded with:

library(tidyverse)

It can also be loaded individually.

library(dplyr)
```

ChatGPT 3.5 response:

dplyr forma parte del núcleo de tidyverse, por lo que se carga automáticamente con:

```
library(tidyverse)
```

También se puede cargar de forma individual.

```
library(dplyr)
```

You can verify the validity of this translation by translating it back into English in a different session.

9.10 8. Translate Programming Languages

ChatGPT can help translate between different programming languages. For example, you can say:

“Translate the following Stata code to R:”

```
recode age (18/35=1) (36/50=2), into(agegroup)
```

ChatGPT 3.5 response:

In R, you can achieve the same recoding of the “age” variable into an “agegroup” variable using the `cut()` function. Here’s the equivalent R code:

```
## Assuming you have a dataframe called "data" with an "age" variable
data$agegroup <- cut(data$age, breaks = c(18, 35, 50), labels = c(1, 2),
                     include.lowest = TRUE)
```

In this code:

- `data$age` refers to the “age” variable in your dataframe.
- `cut()` is used to create the “agegroup” variable.
- `breaks` specifies the breakpoints for age groups (18, 35, and 50).
- `labels` assigns labels 1 and 2 to the corresponding age groups.
- `include.lowest = TRUE` ensures that values exactly equal to the breakpoints are included in the specified group.

This code will create a new “agegroup” variable in your dataframe, similar to what the Stata code does.

9.11 9. Fluid Find and Replace

Lastly, ChatGPT can help you refactor your code in situations where a simple find and replace with your IDE would be time-consuming. For example, if “Afghanistan” is written in three different ways (with a capital “A”, in lowercase letters, and as a country code in the title, “AF”), you could replace all instances with “Ghana”.

You can ask ChatGPT:

“Replace ‘afghanistan’ with ‘ghana’ for the code below:”

```
afghanistan_2000 <- population %>%
  filter(country == "Afghanistan", year == 2000) %>%
  pull(population)

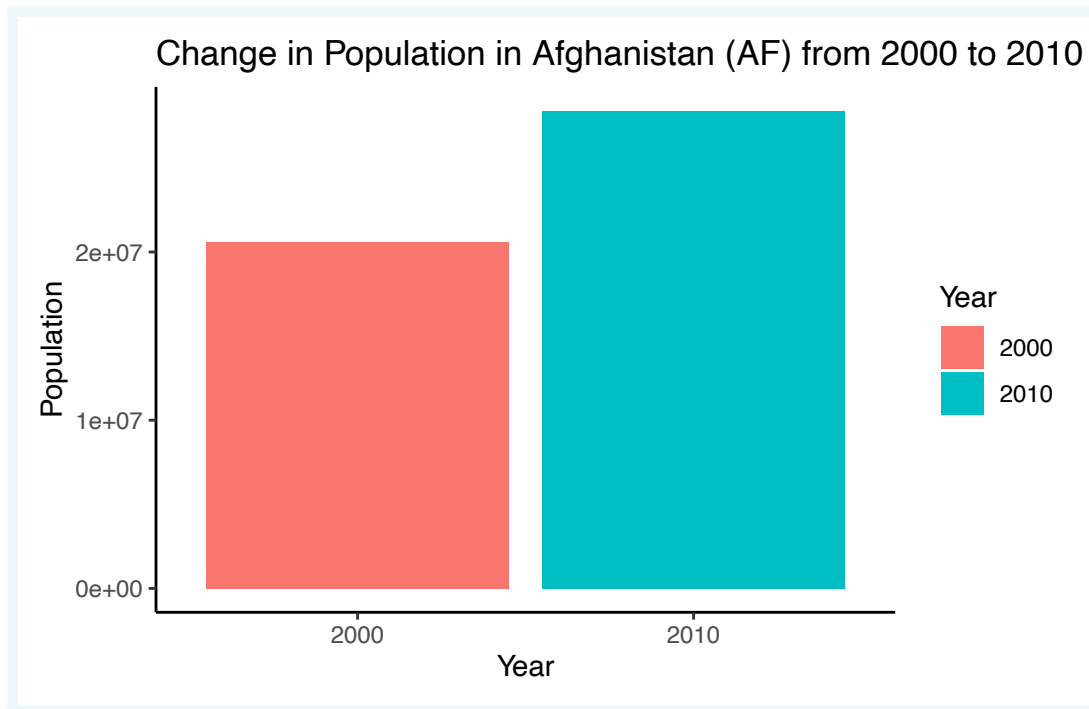
afghanistan_2010 <- population %>%
  filter(country == "Afghanistan", year == 2010) %>%
  pull(population)

afghanistan_population <- data.frame(
  year = c(2000, 2010),
  population = c(afghanistan_2000, afghanistan_2010)
)

percent_change <-
  ((afghanistan_2010 - afghanistan_2000) / afghanistan_2000) * 100

ggplot(afghanistan_population,
  aes(
    x = factor(year),
    y = population,
    fill = factor(year)
  )) +
  geom_bar(stat = "identity") +
  labs(
    x = "Year",
    y = "Population",
    fill = "Year",
    title = paste0(
      "Change in Population in Afghanistan (AF) from 2000 to 2010 (",
      round(percent_change, 2),
      "%)"
    )
  )
```

```
)
) +
theme_classic()
```



ChatGPT 3.5 response:

```
ghana_2000 <- population %>%
  filter(country == "Ghana", year == 2000) %>%
  pull(population)

ghana_2010 <- population %>%
  filter(country == "Ghana", year == 2010) %>%
  pull(population)

ghana_population <- data.frame(
  year = c(2000, 2010),
  population = c(ghana_2000, ghana_2010)
)

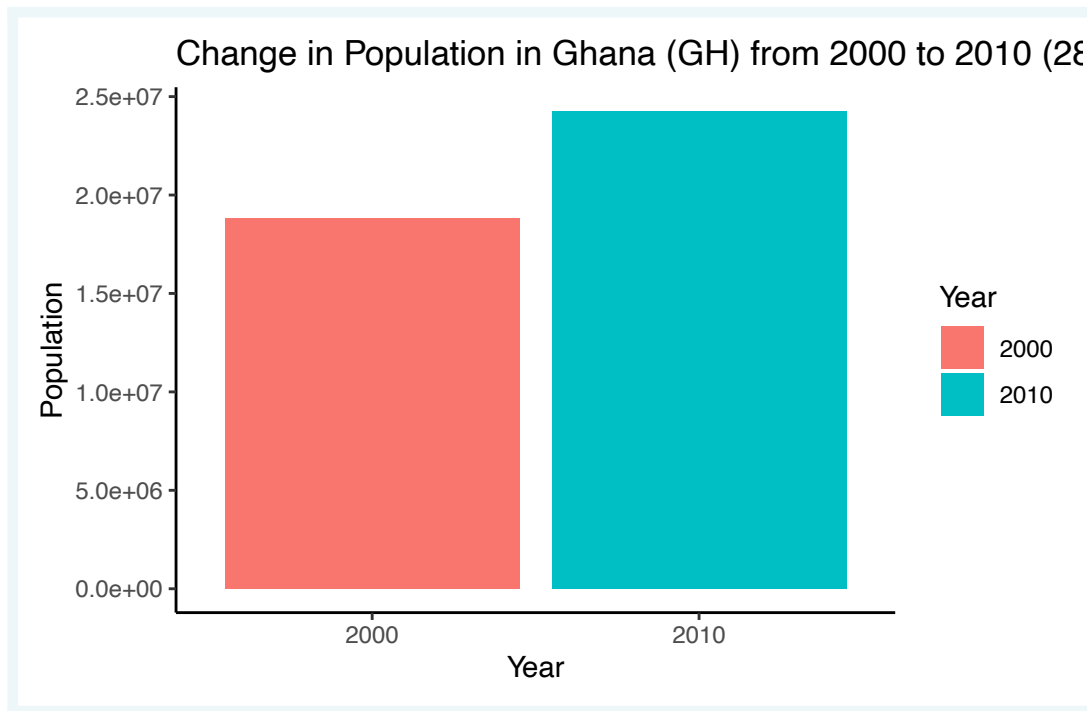
percent_change <- ((ghana_2010 - ghana_2000) / ghana_2000) * 100

ggplot(ghana_population, aes(
  x = factor(year),
  y = population,
  fill = factor(year)
)) +
  geom_bar(stat = "identity") +
  labs(
    x = "Year",
    y = "Population",
```

```

fill = "Year",
title = paste0("Change in Population in Ghana (GH) from 2000 to 2010 (",
               round(percent_change, 2),
               "%)")
) +
theme_classic()

```



This refactored code now refers to "Ghana" instead of "Afghanistan".

9.12 Limitations of ChatGPT

While ChatGPT is a powerful tool for data analysts, it has some limitations:

1. **Lag in Learning:** ChatGPT may struggle with newer software or libraries.
2. **Hallucinations:** Always verify the output of ChatGPT as it can sometimes generate outputs that are incorrect or nonsensical.
3. **Limited Input Length:** ChatGPT cannot process very long prompts. To avoid this, start new conversations frequently.
4. **Weak Math Skills:** At the moment, ChatGPT is not ideal for complex calculations or data analysis.

Chapter 10

Selecting and renaming columns

10.1 Introduction

Today we will begin our exploration of the {dplyr} package! Our first verb on the list is `select` which allows to keep or drop variables from your dataframe. Choosing your variables is the first step in cleaning your data.



Figure 10.1: Fig: the `select()` function.

Let's go !

10.2 Learning objectives

- You can keep or drop columns from a dataframe using the `dplyr::select()` function from the {dplyr} package.
- You can select a range or combination of columns using operators like the colon (:), the exclamation mark (!), and the `c()` function.
- You can select columns based on patterns in their names with helper functions like `starts_with()`, `ends_with()`, `contains()`, and `everything()`.
- You can use `rename()` and `select()` to change column names.

10.3 The Yaounde COVID-19 dataset

In this lesson, we analyse results from 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 [Zenodo](#), and the paper can be viewed [here](#).

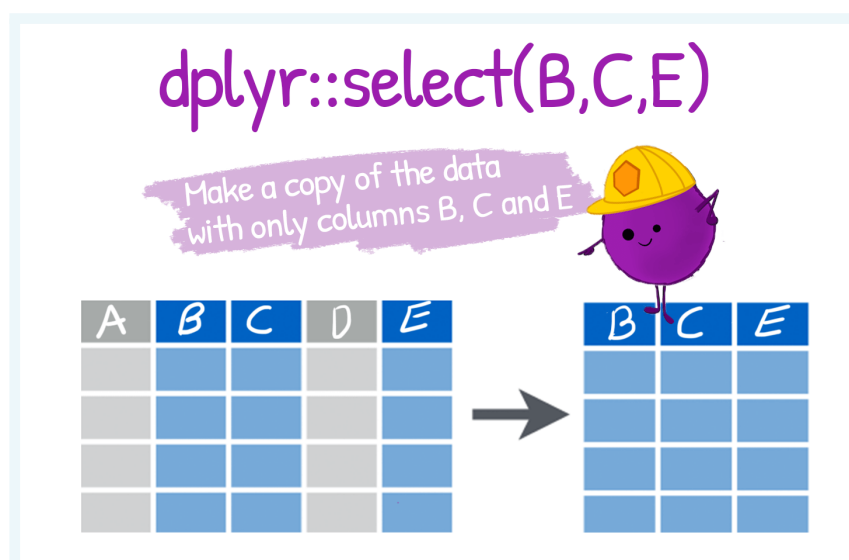
Spend some time browsing through this dataset. Each line corresponds to one patient surveyed. There are some demographic, socio-economic and COVID-related variables. The results of the IgG and IgM antibody tests are in the columns `igg_result` and `igm_result`.

```
yaounde <- read_csv(here::here("data/yaounde_data.csv"))
yaounde
```

```
# A tibble: 5 x 53
  id      date_surveyed  age age_category age_category_3 sex  highest_education
  <chr>   <date>          <dbl> <chr>          <chr>          <chr> <chr>
1 BRIQU~ 2020-10-22      45 45 - 64      Adult          Fema~ Secondary
2 BRIQU~ 2020-10-24      55 45 - 64      Adult          Male  University
3 BRIQU~ 2020-10-24      23 15 - 29      Adult          Male  University
4 BRIQU~ 2020-10-22      20 15 - 29      Adult          Fema~ Secondary
5 BRIQU~ 2020-10-22      55 45 - 64      Adult          Fema~ Primary
# i 46 more variables: occupation <chr>, weight_kg <dbl>, height_cm <dbl>,
# is_smoker <chr>, is_pregnant <chr>, is_medicated <chr>, neighborhood <chr>,
# household_with_children <chr>, breadwinner <chr>, source_of_revenue <chr>,
# has_contact_covid <chr>, igg_result <chr>, igm_result <chr>,
# symptoms <chr>, symp_fever <chr>, symp_headache <chr>, symp_cough <chr>,
# symp_rhinitis <chr>, symp_sneezing <chr>, symp_fatigue <chr>,
# symp_muscle_pain <chr>, symp_nausea_or_vomiting <chr>, ...
```



Figure 10.2: Left: the Yaounde survey team. Right: an antibody test being administered.

Figure 10.3: Fig: the `select()` function. (Drawing adapted from Allison Horst).

10.4 Introducing `select()`

`dplyr::select()` lets us pick which columns (variables) to keep or drop.

We can select a column **by name**:

```
yaounde %>% select(age)
```

```
# A tibble: 5 x 1
  age
<dbl>
1   45
2   55
3   23
4   20
5   55
```

Or we can select a column **by position**:

```
yaounde %>% select(3) # `age` is the 3rd column
```

```
# A tibble: 5 x 1
  age
<dbl>
1   45
2   55
3   23
4   20
5   55
```

To select **multiple variables**, we separate them with commas:

```
yaounde %>% select(age, sex, igg_result)
```

```
# A tibble: 971 x 3
   age sex    igg_result
<dbl> <chr> <chr>
1    45 Female Negative
2    55 Male   Positive
3    23 Male   Negative
4    20 Female Positive
5    55 Female Positive
6    17 Female Negative
7    13 Female Positive
8    28 Male   Negative
9    30 Male   Negative
10   13 Female Positive
# i 961 more rows
```

Practice

- Select the weight and height variables in the yaounde data frame.
- Select the 16th and 22nd columns in the yaounde data frame.

For the next part of the tutorial, let's create a smaller subset of the data, called yao.

```
yao <-
  yaounde %>% select(age,
                    sex,
                    highest_education,
                    occupation,
                    is_smoker,
                    is_pregnant,
                    igg_result,
                    igm_result)

yao
```

```
# A tibble: 5 x 8
   age sex    highest_education occupation    is_smoker is_pregnant igg_result
<dbl> <chr> <chr>                <chr>        <chr>      <chr>      <chr>
1    45 Female Secondary      Informal work~ Non-smok~ No        Negative
2    55 Male   University    Salaried work~ Ex-smoker <NA>      Positive
3    23 Male   University    Student      Smoker    <NA>      Negative
4    20 Female Secondary    Student      Non-smok~ No        Positive
5    55 Female Primary      Trader--Farmer Non-smok~ No        Positive
# i 1 more variable: igm_result <chr>
```

10.4.1 Selecting column ranges with :

The `:` operator selects a **range of consecutive variables**:

```
yao %>% select(age:occupation) # Select all columns from `age` to `occupation`
```

```
# A tibble: 5 x 4
```

	age	sex	highest_education	occupation
	<dbl>	<chr>	<chr>	<chr>
1	45	Female	Secondary	Informal worker
2	55	Male	University	Salaried worker
3	23	Male	University	Student
4	20	Female	Secondary	Student
5	55	Female	Primary	Trader--Farmer

We can also specify a range with column numbers:

```
yao %>% select(1:4) # Select columns 1 to 4
```

```
# A tibble: 5 x 4
```

	age	sex	highest_education	occupation
	<dbl>	<chr>	<chr>	<chr>
1	45	Female	Secondary	Informal worker
2	55	Male	University	Salaried worker
3	23	Male	University	Student
4	20	Female	Secondary	Student
5	55	Female	Primary	Trader--Farmer

Practice

- With the `yaounde` data frame, select the columns between `symptoms` and `sequelae`, inclusive. (“Inclusive” means you should also include `symptoms` and `sequelae` in the selection.)

10.4.2 Excluding columns with !

The **exclamation point** negates a selection:

```
yao %>% select(!age) # Select all columns except `age`
```

```
# A tibble: 5 x 7
```

	sex	highest_education	occupation	is_smoker	is_pregnant	igg_result	igm_result
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	Fema~	Secondary	Informal ~	Non-smok~	No	Negative	Negative
2	Male	University	Salaried ~	Ex-smoker	<NA>	Positive	Negative
3	Male	University	Student	Smoker	<NA>	Negative	Negative
4	Fema~	Secondary	Student	Non-smok~	No	Positive	Negative
5	Fema~	Primary	Trader--F~	Non-smok~	No	Positive	Negative

To drop a range of consecutive columns, we use, for example, `!age:occupation`:

```
yao %>% select(!age:occupation) # Drop columns from `age` to `occupation`
```

```
# A tibble: 5 x 4
  is_smoker is_pregnant igg_result igm_result
  <chr>      <chr>      <chr>      <chr>
1 Non-smoker No          Negative Negative
2 Ex-smoker <NA>        Positive Negative
3 Smoker    <NA>        Negative Negative
4 Non-smoker No          Positive Negative
5 Non-smoker No          Positive Negative
```

To drop several non-consecutive columns, place them inside `!c()`:

```
yao %>% select(!c(age, sex, igg_result))
```

```
# A tibble: 5 x 5
  highest_education occupation is_smoker is_pregnant igm_result
  <chr>             <chr>      <chr>      <chr>      <chr>
1 Secondary         Informal worker Non-smoker No          Negative
2 University        Salaried worker Ex-smoker <NA>        Negative
3 University        Student      Smoker    <NA>        Negative
4 Secondary         Student      Non-smoker No          Negative
5 Primary           Trader--Farmer Non-smoker No          Negative
```

Practice

- From the `yaounde` data frame, **remove** all columns between `highest_education` and `consultation`, inclusive.

10.5 Helper functions for `select()`

`dplyr` has a number of helper functions to make selecting easier by using patterns from the column names. Let's take a look at some of these.

10.5.1 `starts_with()` and `ends_with()`

These two helpers work exactly as their names suggest!

```
yao %>% select(starts_with("is_")) # Columns that start with "is"
```

```
# A tibble: 5 x 2
  is_smoker is_pregnant
  <chr>      <chr>
1 Non-smoker No
2 Ex-smoker <NA>
3 Smoker    <NA>
4 Non-smoker No
5 Non-smoker No
```

```
yao %>% select(ends_with("_result")) # Columns that end with "result"
```

```
# A tibble: 5 x 2
  igg_result igm_result
  <chr>      <chr>
1 Negative  Negative
2 Positive  Negative
3 Negative  Negative
4 Positive  Negative
5 Positive  Negative
```

10.5.2 contains()

`contains()` helps select columns that contain a certain string:

```
yaounde %>% select(contains("drug")) # Columns that contain the string "drug"
```

```
# A tibble: 5 x 12
  drugsource is_drug_parac is_drug_antibio is_drug_hydrocortisone
  <chr>      <dbl>      <dbl>      <dbl>
1 Self or familial      1          0          0
2 <NA>                  NA          NA          NA
3 <NA>                  NA          NA          NA
4 Self or familial      0          1          0
5 <NA>                  NA          NA          NA
# i 8 more variables: is_drug_other_anti_inflam <dbl>, is_drug_antiviral <dbl>,
#   is_drug_chloro <dbl>, is_drug_tradn <dbl>, is_drug_oxygen <dbl>,
#   is_drug_other <dbl>, is_drug_no_resp <dbl>, is_drug_none <dbl>
```

10.5.3 everything()

Another helper function, `everything()`, matches all variables that have not yet been selected.

```
## First, `is_pregnant`, then every other column.
yao %>% select(is_pregnant, everything())
```

```
# A tibble: 5 x 8
  is_pregnant age sex highest_education occupation is_smoker igg_result
  <chr>      <dbl> <chr> <chr>      <chr>      <chr>      <chr>
1 No        45 Female Secondary Informal work~ Non-smok~ Negative
2 <NA>      55 Male University Salaried work~ Ex-smoker Positive
3 <NA>      23 Male University Student Smoker Negative
4 No        20 Female Secondary Student Non-smok~ Positive
5 No        55 Female Primary Trader--Farmer Non-smok~ Positive
# i 1 more variable: igm_result <chr>
```

It is often useful for establishing the order of columns.

Say we wanted to bring the `is_pregnant` column to the start of the `yao` data frame, we could type out all the column names manually:

```
yao %>% select(is_pregnant,
              age,
              sex,
              highest_education,
              occupation,
              is_smoker,
              igg_result,
              igm_result)
```

```
# A tibble: 5 x 8
  is_pregnant age sex highest_education occupation is_smoker igg_result
  <chr>      <dbl> <chr> <chr>                <chr>        <chr>    <chr>
1 No         45 Female Secondary Informal work~ Non-smok~ Negative
2 <NA>       55 Male University Salaried work~ Ex-smoker Positive
3 <NA>       23 Male University Student Smoker Negative
4 No         20 Female Secondary Student Non-smok~ Positive
5 No         55 Female Primary Trader--Farmer Non-smok~ Positive
# i 1 more variable: igm_result <chr>
```

But this would be painful for larger data frames, such as our original `yaounde` data frame. In such a case, we can use `everything()`:

```
## Bring `is_pregnant` to the front of the data frame
yaounde %>% select(is_pregnant, everything())
```

```
# A tibble: 5 x 53
  is_pregnant id date_surveyed age age_category age_category_3 sex
  <chr>      <chr>      <date>      <dbl> <chr>        <chr>        <chr>
1 No        BRIQUETERIE~ 2020-10-22 45 45 - 64 Adult Fema~
2 <NA>      BRIQUETERIE~ 2020-10-24 55 45 - 64 Adult Male
3 <NA>      BRIQUETERIE~ 2020-10-24 23 15 - 29 Adult Male
4 No        BRIQUETERIE~ 2020-10-22 20 15 - 29 Adult Fema~
5 No        BRIQUETERIE~ 2020-10-22 55 45 - 64 Adult Fema~
# i 46 more variables: highest_education <chr>, occupation <chr>,
# weight_kg <dbl>, height_cm <dbl>, is_smoker <chr>, is_medicated <chr>,
# neighborhood <chr>, household_with_children <chr>, breadwinner <chr>,
# source_of_revenue <chr>, has_contact_covid <chr>, igg_result <chr>,
# igm_result <chr>, symptoms <chr>, symp_fever <chr>, symp_headache <chr>,
# symp_cough <chr>, symp_rhinitis <chr>, symp_sneezing <chr>,
# symp_fatigue <chr>, symp_muscle_pain <chr>, ...
```

This helper can be combined with many others.

```
## Bring columns that end with "result" to the front of the data frame
yaounde %>% select(ends_with("result"), everything())
```

```
# A tibble: 5 x 53
  igg_result igm_result id      date_surveyed  age age_category age_category_3
  <chr>      <chr>      <chr>      <date>          <dbl> <chr>          <chr>
1 Negative   Negative   BRIQUET~ 2020-10-22      45 45 - 64      Adult
2 Positive   Negative   BRIQUET~ 2020-10-24      55 45 - 64      Adult
3 Negative   Negative   BRIQUET~ 2020-10-24      23 15 - 29      Adult
4 Positive   Negative   BRIQUET~ 2020-10-22      20 15 - 29      Adult
5 Positive   Negative   BRIQUET~ 2020-10-22      55 45 - 64      Adult
# i 46 more variables: sex <chr>, highest_education <chr>, occupation <chr>,
# weight_kg <dbl>, height_cm <dbl>, is_smoker <chr>, is_pregnant <chr>,
# is_medicated <chr>, neighborhood <chr>, household_with_children <chr>,
# breadwinner <chr>, source_of_revenue <chr>, has_contact_covid <chr>,
# symptoms <chr>, symp_fever <chr>, symp_headache <chr>, symp_cough <chr>,
# symp_rhinitis <chr>, symp_sneezing <chr>, symp_fatigue <chr>,
# symp_muscle_pain <chr>, symp_nausea_or_vomiting <chr>, ...
```

Practice

- Select all columns in the yaounde data frame that start with "is_".
- Move the columns that start with "is_" to the beginning of the yaounde data frame.

10.6 Change column names with rename()

`dplyr::rename()` is used to change column names:

```
## Rename `age` and `sex` to `patient_age` and `patient_sex`
yaounde %>%
  rename(patient_age = age,
         patient_sex = sex)
```

```
# A tibble: 5 x 53
  id      date_surveyed patient_age age_category age_category_3 patient_sex
  <chr>      <date>          <dbl> <chr>          <chr>          <chr>
1 BRIQUETERIE~ 2020-10-22      45 45 - 64      Adult      Female
2 BRIQUETERIE~ 2020-10-24      55 45 - 64      Adult      Male
3 BRIQUETERIE~ 2020-10-24      23 15 - 29      Adult      Male
4 BRIQUETERIE~ 2020-10-22      20 15 - 29      Adult      Female
5 BRIQUETERIE~ 2020-10-22      55 45 - 64      Adult      Female
# i 47 more variables: highest_education <chr>, occupation <chr>,
# weight_kg <dbl>, height_cm <dbl>, is_smoker <chr>, is_pregnant <chr>,
# is_medicated <chr>, neighborhood <chr>, household_with_children <chr>,
# breadwinner <chr>, source_of_revenue <chr>, has_contact_covid <chr>,
# igg_result <chr>, igm_result <chr>, symptoms <chr>, symp_fever <chr>,
# symp_headache <chr>, symp_cough <chr>, symp_rhinitis <chr>,
# symp_sneezing <chr>, symp_fatigue <chr>, symp_muscle_pain <chr>, ...
```

RENAME COLUMNS

```
dplyr::rename(enemies = species)
```

species enemies	status	diet
Dog	active	carnivore
House cat		carnivore
Osprey		carnivore

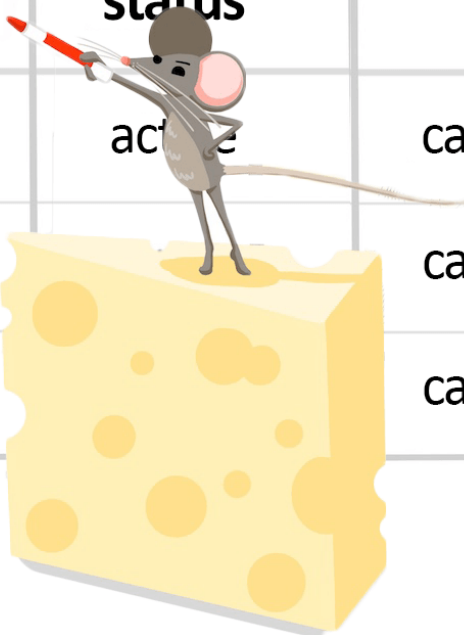


Figure 10.4: Fig: the `rename()` function. (Drawing adapted from Allison Horst)

Watch Out

The fact that the new name comes first in the function (`rename(NEWNAME = OLDNAME)`) is sometimes confusing. You should get used to this with time.

10.6.1 Rename within `select()`

You can also rename columns while selecting them:

```
## Select `age` and `sex`, and rename them to `patient_age` and `patient_sex`
yaounde %>%
  select(patient_age = age,
         patient_sex = sex)
```

```
# A tibble: 5 x 2
  patient_age patient_sex
    <dbl>    <chr>
1      45 Female
2      55 Male
3      23 Male
4      20 Female
5      55 Female
```

10.7 Wrap up

I hope this first lesson has allowed you to see how intuitive and useful the {dplyr} verbs are! This is the first of a series of basic data wrangling verbs: see you in the next lesson to learn more.

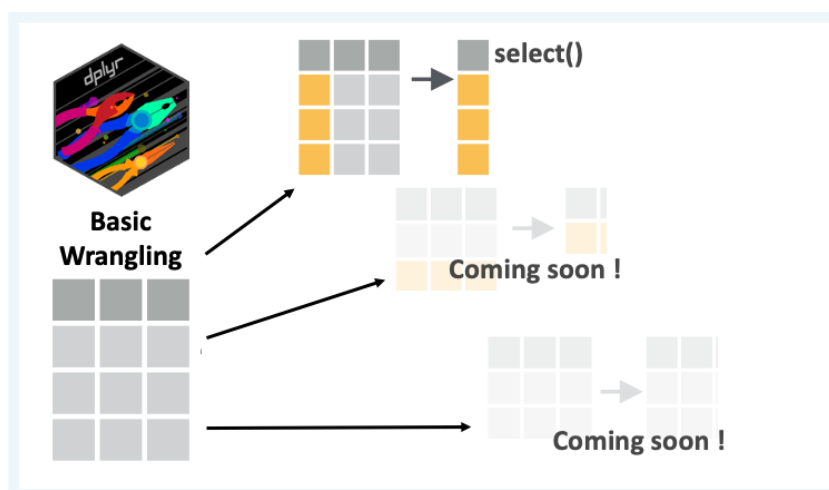


Figure 10.5: Fig: Basic Data Wrangling Dplyr Verbs.

References

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

- Horst, A. (2021). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)
- *Subset columns using their names and types—Select*. (n.d.). Retrieved 31 December 2021, from <https://dplyr.tidyverse.org/reference/select.html>

Artwork was adapted from:

- Horst, A. (2021). *R & stats illustrations by Allison Horst*. <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

10.8 Solutions

```
.SOLUTION_Q_weight_height()
```

```
yaounde %>% select(weight_kg, height_cm)
```

```
.SOLUTION_Q_cols_16_22()
```

```
yaounde %>% select(16, 22)
```

```
.SOLUTION_Q_symp_to_sequel()
```

```
yaounde %>% select(symptoms:sequelae)
```

```
.SOLUTION_Q_educ_consult()
```

```
yaounde %>% select(!c(highest_education:consultation))
```

```
.SOLUTION_Q_starts_with_is()
```

```
yaounde %>% select(starts_with("is"))
```

```
.SOLUTION_Q_rearrange()
```

```
yaounde %>% select(starts_with("is_"), everything())
```

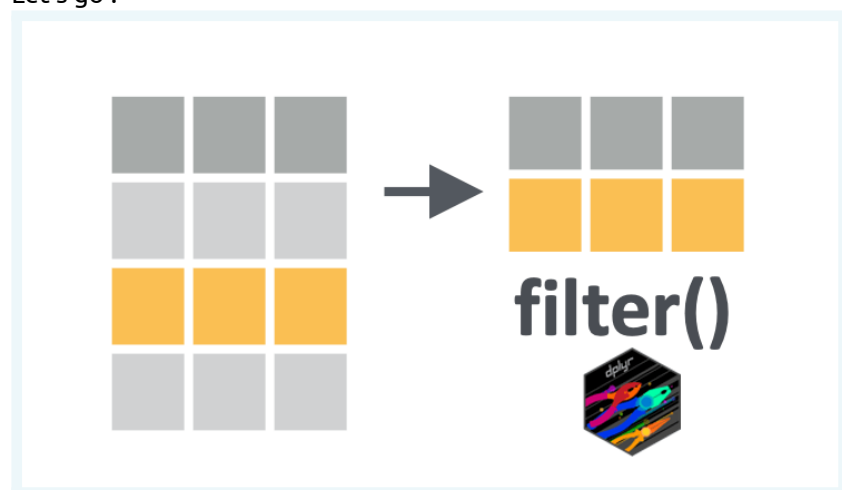
Chapter 11

Filtering rows

11.1 Intro

Onward with the {dplyr} package, discovering the `filter` verb. Last time we saw how to select variables (columns) and today we will see how to keep or drop data entries, rows, using `filter`. Dropping abnormal data entries or keeping subsets of your data points is another essential aspect of data wrangling.

Let's go !



11.2 Learning objectives

1. You can use `dplyr::filter()` to keep or drop rows from a dataframe.
2. You can filter rows by specifying conditions on numbers or strings using relational operators like greater than (`>`), less than (`<`), equal to (`==`), and not equal to (`!=`).
3. You can filter rows by combining conditions using logical operators like the ampersand (`&`) and the vertical bar (`|`).
4. You can filter rows by negating conditions using the exclamation mark (`!`) logical operator.
5. You can filter rows with missing values using the `is.na()` function.

11.3 The Yaounde COVID-19 dataset

In this lesson, we will again use the data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
yaounde <- read_csv(here::here('data/yaounde_data.csv'))
### a smaller subset of variables
yao <- yaounde %>%
  select(age, sex, weight_kg, highest_education, neighborhood,
         occupation, is_smoker, is_pregnant,
         igg_result, igm_result)
yao
```

```
# A tibble: 5 x 10
   age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1   45 Female      95 Secondary      Briqueterie Informal work~ Non-smok~
2   55 Male       96 University    Briqueterie Salaried work~ Ex-smoker
3   23 Male       74 University    Briqueterie Student      Smoker
4   20 Female     70 Secondary    Briqueterie Student      Non-smok~
5   55 Female     67 Primary      Briqueterie Trader--Farmer Non-smok~
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

11.4 Introducing filter()

We use `filter()` to keep rows that satisfy a set of conditions. Let's take a look at a simple example. If we want to keep just the male records, we run:

```
yao %>% filter(sex == "Male")
```

```
# A tibble: 5 x 10
   age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1   55 Male      96 University    Briqueterie Salaried worker Ex-smoker
2   23 Male      74 University    Briqueterie Student      Smoker
3   28 Male      62 Doctorate    Briqueterie Student      Non-smok~
4   30 Male      73 Secondary    Briqueterie Trader      Non-smok~
5   42 Male      71 Secondary    Briqueterie Trader      Ex-smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

Note the use of the double equal sign `==` rather than the single equal sign `=`. The `==` sign tests for equality, as demonstrated below:

```
### create the object `sex_vector` with three elements
sex_vector <- c("Male", "Female", "Female")
### test which elements are equal to "Male"
sex_vector == "Male"
```

```
[1] TRUE FALSE FALSE
```

So the code `yao %>% filter(sex == "Male")` will keep all rows where the equality test `sex == "Male"` evaluates to `TRUE`.

It is often useful to chain `filter()` with `nrow()` to get the number of rows fulfilling a condition.

```
### how many respondents were male?
yao %>%
  filter(sex == "Male") %>%
  nrow()
```

```
[1] 422
```

Key Point

The double equal sign, `==`, tests for equality, while the single equals sign, `=`, is used for specifying values to arguments inside functions.

Practice

Filter the `yao` data frame to respondents who were pregnant during the survey. How many respondents were female? (Use `filter()` and `nrow()`)

11.5 Relational operators

The `==` operator introduced above is an example of a “relational” operator, as it tests the relation between two values. Here is a list of some of these operators:

Operator	is TRUE if
<code>A < B</code>	A is less than B
<code>A <= B</code>	A is less than or equal to B
<code>A > B</code>	A is greater than B
<code>A >= B</code>	A is greater than or equal to B
<code>A == B</code>	A is equal to B
<code>A != B</code>	A is not equal to B
<code>A %in% B</code>	A is an element of B

Let’s see how to use these within `filter()`:

```
yao %>% filter(sex != "Male") ## keep rows where `sex` is not "Male"
```

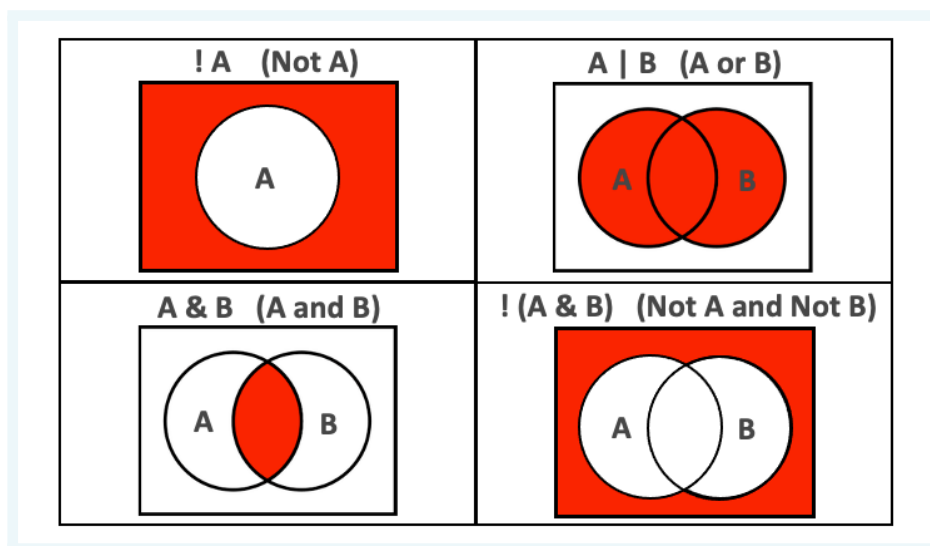


Figure 11.1: Fig: AND and OR operators visualized.

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1   45 Female      95 Secondary      Briqueterie Informal work~ Non-smok~
2   20 Female      70 Secondary      Briqueterie Student      Non-smok~
3   55 Female      67 Primary        Briqueterie Trader--Farmer Non-smok~
4   17 Female      65 Secondary      Briqueterie Student      Non-smok~
5   13 Female      65 Secondary      Briqueterie Student      Non-smok~
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

```
yao %>% filter(age < 6) ## keep respondents under 6
```

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1    5 Female      19 Primary      Carriere      Student    Non-smoker
2    5 Female      26 Primary      Carriere      No response Non-smoker
3    5 Male       16 Primary      Cité Verte    Student    Non-smoker
4    5 Female      21 Primary      Ekoudou       Student    Non-smoker
5    5 Male       15 Primary      Ekoudou       Student    Non-smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

```
yao %>% filter(age >= 70) ## keep respondents aged at least 70
```

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1   78 Male      95 Secondary      Briqueterie Retired--Info~ Ex-smoker
2   79 Female     40 Primary      Briqueterie Retired      Non-smok~
3   78 Female     60 Primary      Briqueterie Unemployed   Non-smok~
4   75 Male      74 Primary      Briqueterie Informal work~ Non-smok~
```

```
5 72 Male 65 Secondary Carriere Retired Non-smok~
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

```
### keep respondents whose highest education is "Primary" or "Secondary"
yao %>% filter(highest_education %in% c("Primary", "Secondary"))
```

```
# A tibble: 5 x 10
  age sex weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr> <dbl> <chr> <chr> <chr> <chr>
1 45 Female 95 Secondary Briqueterie Informal work~ Non-smok~
2 20 Female 70 Secondary Briqueterie Student Non-smok~
3 55 Female 67 Primary Briqueterie Trader--Farmer Non-smok~
4 17 Female 65 Secondary Briqueterie Student Non-smok~
5 13 Female 65 Secondary Briqueterie Student Non-smok~
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

Practice

From yao, keep only respondents who were children (under 18).
With %in%, keep only respondents who live in the “Tsinga” or “Messa” neighborhoods.

11.6 Combining conditions with & and |

We can pass multiple conditions to a single filter() statement separated by commas:

```
### keep respondents who are pregnant and are ex-smokers
yao %>% filter(is_pregnant == "Yes", is_smoker == "Ex-smoker") ## only one row
```

```
# A tibble: 1 x 10
  age sex weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr> <dbl> <chr> <chr> <chr> <chr>
1 25 Female 90 Secondary Carriere Home-maker Ex-smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

When multiple conditions are separated by a comma, they are implicitly combined with an **and** (&).

It is best to replace the comma with & to make this more explicit.

```
### same result as before, but `&` is more explicit
yao %>% filter(is_pregnant == "Yes" & is_smoker == "Ex-smoker")
```

```
# A tibble: 1 x 10
  age sex weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr> <dbl> <chr> <chr> <chr> <chr>
1 25 Female 90 Secondary Carriere Home-maker Ex-smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

i Side Note

Don't confuse:

- the “,” in listing several conditions in filter filter(A,B) i.e. filter based on condition A and (&) condition B
- the “,” in lists c(A,B) which is listing different components of the list (and has nothing to do with the & operator)

If we want to combine conditions with an **or**, we use the vertical bar symbol, |.

```
### respondents who are pregnant OR who are ex-smokers
yao %>% filter(is_pregnant == "Yes" | is_smoker == "Ex-smoker")
```

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1    55 Male      96 University          Briqueterie    Salaried worker Ex-smoker
2    42 Male      71 Secondary          Briqueterie    Trader        Ex-smoker
3    38 Male      71 University          Briqueterie    Informal worker Ex-smoker
4    69 Male     108 University          Briqueterie    Retired        Ex-smoker
5    65 Male      93 Secondary          Briqueterie    Retired        Ex-smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

💡 Practice

Filter yao to only keep men who tested IgG positive.

Filter yao to keep both children (under 18) and anyone whose highest education is primary school.

11.7 Negating conditions with !

To negate conditions, we wrap them in !().

Below, we drop respondents who are children (less than 18 years) or who weigh less than 30kg:

```
### drop respondents < 18 years OR < 30 kg
yao %>% filter(!(age < 18 | weight_kg < 30))
```

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1    45 Female      95 Secondary          Briqueterie    Informal work~ Non-smok~
2    55 Male      96 University          Briqueterie    Salaried work~ Ex-smoker
3    23 Male      74 University          Briqueterie    Student        Smoker
4    20 Female      70 Secondary          Briqueterie    Student        Non-smok~
5    55 Female      67 Primary            Briqueterie    Trader--Farmer Non-smok~
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

The ! operator is also used to negate %in% since R does not have an operator for **NOT in**.

```
### drop respondents whose highest education is NOT "Primary" or "Secondary"
yao %>% filter(!(highest_education %in% c("Primary", "Secondary")))
```

```
# A tibble: 5 x 10
  age sex    weight_kg highest_education neighborhood occupation is_smoker
<dbl> <chr>    <dbl> <chr>                <chr>          <chr>      <chr>
1   55 Male      96 University          Briqueterie Salaried worker Ex-smoker
2   23 Male      74 University          Briqueterie Student       Smoker
3   28 Male      62 Doctorate          Briqueterie Student       Non-smok~
4   38 Male      71 University          Briqueterie Informal worker Ex-smoker
5   54 Male      71 University          Briqueterie Salaried worker Smoker
# i 3 more variables: is_pregnant <chr>, igg_result <chr>, igm_result <chr>
```

Key Point

It is easier to read `filter()` statements as **keep** statements, to avoid confusion over whether we are filtering **in** or filtering **out**!

So the code below would read: “**keep** respondents who are under 18 or who weigh less than 30kg”.

```
yao %>% filter(age < 18 | weight_kg < 30)
```

And when we wrap conditions in `!()`, we can then read `filter()` statements as **drop** statements. So the code below would read: “**drop** respondents who are under 18 or who weigh less than 30kg”.

```
yao %>% filter(!(age < 18 | weight_kg < 30))
```

Practice

From `yao`, drop respondents who live in the Tsinga or Messa neighborhoods.

11.8 NA values

The relational operators introduced so far do not work with NA.

Let’s make a data subset to illustrate this.

```
yao_mini <- yao %>%
  select(sex, is_pregnant) %>%
  slice(1,11,50,2) ## custom row order

yao_mini
```

```
# A tibble: 4 x 2
  sex    is_pregnant
<chr>  <chr>
1 Female No
2 Female No response
3 Female Yes
4 Male    <NA>
```

In `yao_mini`, the last respondent has an NA for the `is_pregnant` column, because he is male.

Trying to select this row using `== NA` will not work.

```
yao_mini %>% filter(is_pregnant == NA) ## does not work
```

```
# A tibble: 0 x 2
# i 2 variables: sex <chr>, is_pregnant <chr>
```

```
yao_mini %>% filter(is_pregnant == "NA") ## does not work
```

```
# A tibble: 0 x 2
# i 2 variables: sex <chr>, is_pregnant <chr>
```

This is because NA is a non-existent value. So R cannot evaluate whether it is “equal to” or “not equal to” anything.

The special function `is.na()` is therefore necessary:

```
### keep rows where `is_pregnant` is NA
yao_mini %>% filter(is.na(is_pregnant))
```

```
# A tibble: 1 x 2
  sex    is_pregnant
<chr> <chr>
1 Male    <NA>
```

This function can be negated with `!`:

```
### drop rows where `is_pregnant` is NA
yao_mini %>% filter(!is.na(is_pregnant))
```

```
# A tibble: 3 x 2
  sex    is_pregnant
<chr> <chr>
1 Female No
2 Female No response
3 Female Yes
```

Side Note

For tibbles, RStudio will highlight NA values bright red to distinguish them from other values:

```
# A tibble: 5 × 3
  age sex    is_pregnant
<dbl> <chr> <chr>
1   32 Male      NA
2   23 Female Yes
3   35 Male      NA
4   31 Female No
5   17 Female No response
```

Figure 11.2: A common error with NA

i Side Note

NA values can be identified but any other encoding such as "NA" or "NaN", which are encoded as strings, will be imperceptible to the functions (they are strings, like any others).

💡 Practice

From the yao dataset, keep all the respondents who had missing records for the report of their smoking status.

💡 Practice

For some respondents the respiration rate, in breaths per minute, was recorded in the `respiration_frequency` column.
From yaounde, drop those with a respiration frequency under 20. Think about NAs while doing this! You should avoid also dropping the NA values.

11.9 Wrap up

Now you know the two essential verbs to `select()` columns and to `filter()` rows. This way you keep the variables you are interested in by selecting your columns and you keep the data entries you judge relevant by filtering your rows.

But what about modifying, transforming your data? We will learn about this in the next lesson. See you there!

References

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

- Horst, A. (2021). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)

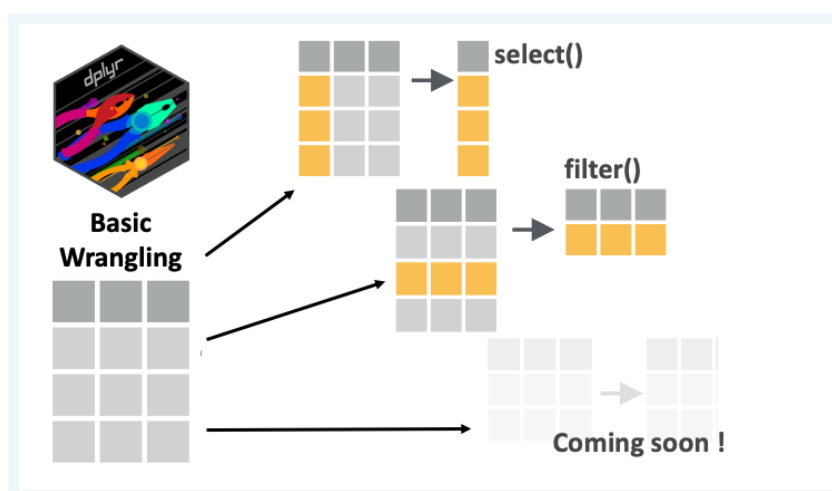


Figure 11.3: Fig: Basic Data Wrangling: `select()` and `filter()`.

- *Subset rows using column values—Filter.* (n.d.). Retrieved 12 January 2022, from <https://dplyr.tidyverse.org/reference/filter.html>

Artwork was adapted from:

- Horst, A. (2021). *R & stats illustrations by Allison Horst.* <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

11.10 Solutions

```
.SOLUTION_Q_is_pregnant()
```

```
yao %>% filter(is_pregnant == 'Yes')
```

```
.SOLUTION_Q_female_nrow()
```

```
yao %>%
  filter(sex == 'Female') %>%
  nrow()
```

```
.SOLUTION_Q_under_18()
```

```
yao %>% filter(age < 18)
```

```
.SOLUTION_Q_tsinga_messa()
```

```
yao %>%  
filter(neighborhood %in% c("Tsinga", "Messa"))
```

```
.SOLUTION_Q_male_positive()
```

```
yao %>%  
filter(sex == "Male" & igg_result == "Positive")
```

```
.SOLUTION_Q_child_primary()
```

```
yao %>% filter(age < 18 | highest_education == "Primary")
```

```
.SOLUTION_Q_not_tsinga_messa()
```

```
yao %>%  
filter(!(neighborhood %in% c("Tsinga", "Messa")))
```

```
.SOLUTION_Q_na_smoker()
```

```
yao %>% filter(is.na(is_smoker))
```

Chapter 12

Mutating columns

12.1 Intro

You now know how to keep or drop columns and rows from your dataset. Today you will learn how to modify existing variables or create new ones, using the `mutate()` verb from `{dplyr}`. This is an essential step in most data analysis projects.

Let's go!

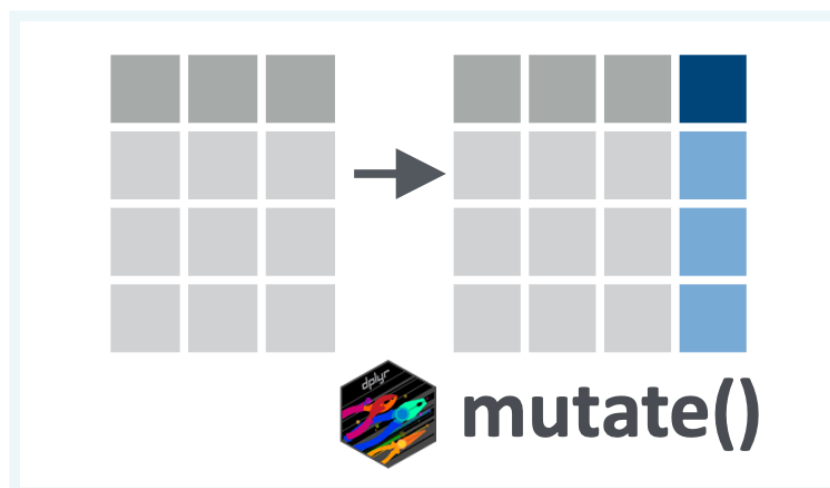


Figure 12.1: Fig: the `mutate()` verb.

12.2 Learning objectives

1. You can use the `mutate()` function from the `{dplyr}` package to create new variables or modify existing variables.
2. You can create new numeric, character, factor, and boolean variables

12.3 Packages

This lesson will require the packages loaded below:

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

12.4 Datasets

In this lesson, we will again use the data from the COVID-19 serological survey conducted in Yaounde, Cameroon. Below, we import the dataset `yaounde` and create a smaller subset called `yao`. Note that this dataset is slightly different from the one used in the previous lesson.

```
yaounde <- read_csv(here::here('data/yaounde_data.csv'))

### a smaller subset of variables
yao <- yaounde %>% select(date_surveyed,
                        age,
                        weight_kg, height_cm,
                        symptoms, is_smoker)

yao
```

```
# A tibble: 10 x 6
  date_surveyed   age weight_kg height_cm symptoms is_smoker
  <date>         <dbl>   <dbl>   <dbl> <chr>      <chr>
1 2020-10-22      45      95     169 Muscle pain Non-smok~
2 2020-10-24      55      96     185 No symptoms Ex-smoker
3 2020-10-24      23      74     180 No symptoms Smoker
4 2020-10-22      20      70     164 Rhinitis--Sneezing--Anosmi~ Non-smok~
5 2020-10-22      55      67     147 No symptoms Non-smok~
6 2020-10-25      17      65     162 Fever--Cough--Rhinitis--Na~ Non-smok~
7 2020-10-25      13      65     150 Sneezing Non-smok~
8 2020-10-24      28      62     173 Headache Non-smok~
9 2020-10-24      30      73     170 Fever--Rhinitis--Anosmia o~ Non-smok~
10 2020-10-24      13      56     153 No symptoms Non-smok~
```

We will also use a dataset from a cross-sectional study that aimed to determine the prevalence of sarcopenia in the elderly population (>60 years) in in Karnataka, India. Sarcopenia is a condition that is common in elderly people and is characterized by progressive and generalized loss of skeletal muscle mass and strength. The data was obtained from Zenodo [here](#), and the source publication can be found [here](#).

Below, we import and view this dataset:

```
sarcopenia <- read_csv(here::here('data/sarcopenia_elderly.csv'))

sarcopenia
```

```
# A tibble: 10 x 9
  number   age age_group sex_male_1_female_0 marital_status height_meters
  <dbl> <dbl> <chr>          <dbl> <chr>          <dbl>
```

```

1      7 60.8 Sixties      0 married      1.57
2      8 72.3 Seventies   1 married      1.65
3      9 62.6 Sixties     0 married      1.59
4     12 72  Seventies    0 widow       1.47
5     13 60.1 Sixties     0 married      1.55
6     19 60.6 Sixties     0 married      1.42
7     45 60.1 Sixties     1 widower     1.68
8     46 60.2 Sixties     0 married      1.8
9     51 63  Sixties      0 married      1.6
10    56 60.4 Sixties     0 married      1.6
# i 3 more variables: weight_kg <dbl>, grip_strength_kg <dbl>,
#   skeletal_muscle_index <dbl>

```

12.5 Introducing mutate()



Figure 12.2: The mutate() function. (Drawing adapted from Allison Horst)

We use `dplyr::mutate()` to create new variables or modify existing variables. The syntax is quite intuitive, and generally looks like `df %>% mutate(new_column_name = what_it_contains)`.

Let's see a quick example.

The `yaounde` dataset currently contains a column called `height_cm`, which shows the height, in centimeters, of survey respondents. Let's create a data frame, `yao_height`, with just this column, for easy illustration:

```

yao_height <- yaounde %>% select(height_cm)
yao_height

# A tibble: 5 x 1
  height_cm
    <dbl>
1     169
2     185

```

```
3      180
4      164
5      147
```

What if you wanted to **create a new variable**, called `height_meters` where heights are converted to meters? You can use `mutate()` for this, with the argument `height_meters = height_cm/100`:

```
yao_height %>%
  mutate(height_meters = height_cm/100)
```

```
# A tibble: 5 x 2
  height_cm height_meters
    <dbl>         <dbl>
1      169          1.69
2      185          1.85
3      180          1.8
4      164          1.64
5      147          1.47
```

Great. The syntax is beautifully simple, isn't it?

Now, imagine there was a small error in the equipment used to measure respondent heights, and all heights are 5cm too small. You therefore like to add 5cm to all heights in the dataset. To do this, rather than creating a new variable as you did before, you can **modify the existing variable** with `mutate`:

```
yao_height %>%
  mutate(height_cm = height_cm + 5)
```

```
# A tibble: 5 x 1
  height_cm
    <dbl>
1      174
2      190
3      185
4      169
5      152
```

Again, very easy to do!

Practice

The sarcopenia data frame has a variable `weight_kg`, which contains respondents' weights in kilograms. Create a new column, called `weight_grams`, with respondents' weights in grams. Store your answer in the `Q_weight_to_g` object. (1 kg equals 1000 grams.)

```
## Complete the code with your answer:
Q_weight_to_g <-
  sarcopenia %>%
  -----
```

Hopefully you now see that the mutate function is quite user-friendly. In theory, we could end the lesson here, because you now know how to use mutate(). But of course, the devil will be in the details—the interesting thing is not mutate() itself but what goes *inside* the mutate() call.

The rest of the lesson will go through a few use cases for the mutate() verb. In the process, we'll touch on several new functions you have not yet encountered.

12.6 Creating a Boolean variable

You can use mutate() to create a Boolean variable to categorize part of your population.

Below we create a Boolean variable, is_child which is either TRUE if the subject is a child or FALSE if the subject is an adult (first, we select just the age variable so it's easy to see what is being done; you will likely not need this pre-selection for your own analyses).

```
yao %>%
  select(age) %>%
  mutate(is_child = age <= 18)
```

```
# A tibble: 5 x 2
   age is_child
<dbl> <lgl>
1    45 FALSE
2    55 FALSE
3    23 FALSE
4    20 FALSE
5    55 FALSE
```

The code age <= 18 evaluates whether each age is less than or equal to 18. Ages that match that condition (ages 18 and under) are TRUE and those that fail the condition are FALSE.

Such a variable is useful to, for example, count the number of children in the dataset. The code below does this with the janitor::tabyl() function:

```
yao %>%
  mutate(is_child = age <= 18) %>%
  tabyl(is_child)
```

```
is_child  n  percent
FALSE 662 0.6817714
TRUE 309 0.3182286
```

You can observe that 31.8% (0.318...) of respondents in the dataset are children.

Let's see one more example, since the concept of Boolean variables can be a bit confusing. The symptoms variable reports any respiratory symptoms experienced by the patient:

```
yao %>%
  select(symptoms)
```

```
# A tibble: 5 x 1
  symptoms
  <chr>
1 Muscle pain
2 No symptoms
3 No symptoms
4 Rhinitis--Sneezing--Anosmia or ageusia
5 No symptoms
```

You could create a Boolean variable, called `has_no_symptoms`, that is set to `TRUE` if the respondent reported no symptoms:

```
yao %>%
  select(symptoms) %>%
  mutate(has_no_symptoms = symptoms == "No symptoms")
```

```
# A tibble: 5 x 2
  symptoms                has_no_symptoms
  <chr>                  <lgl>
1 Muscle pain          FALSE
2 No symptoms          TRUE
3 No symptoms          TRUE
4 Rhinitis--Sneezing--Anosmia or ageusia FALSE
5 No symptoms          TRUE
```

Similarly, you could create a Boolean variable called `has_any_symptoms` that is set to `TRUE` if the respondent reported any symptoms. For this, you'd simply swap the `symptoms == "No symptoms"` code for `symptoms != "No symptoms"`:

```
yao %>%
  select(symptoms) %>%
  mutate(has_any_symptoms = symptoms != "No symptoms")
```

```
# A tibble: 5 x 2
  symptoms                has_any_symptoms
  <chr>                  <lgl>
1 Muscle pain          TRUE
2 No symptoms          FALSE
3 No symptoms          FALSE
4 Rhinitis--Sneezing--Anosmia or ageusia TRUE
5 No symptoms          FALSE
```

Still confused by the Boolean examples? That's normal. Pause and play with the code above a little. Then try the practice question below

Practice

Women with a grip strength below 20kg are considered to have low grip strength. With a female subset of the sarcopenia data frame, add a variable called `low_grip_strength` that is `TRUE` for women with a grip strength < 20 kg and `FALSE` for other women.

```
## Complete the code with your answer:
Q_women_low_grip_strength <-
  sarcopenia %>%
  filter(sex_male_1_female_0 == 0) # first we filter the dataset to only women
  # mutate code here
```

What percentage of women surveyed have a low grip strength according to the definition above? Enter your answer as a number without quotes (e.g. 43.3 or 12.2), to one decimal place.

```
Q_prop_women_low_grip_strength <- YOUR_ANSWER_HERE
```

12.7 Creating a numeric variable based on a formula

Now, let's look at an example of creating a numeric variable, the body mass index (BMI), which is a commonly used health indicator. The formula for the body mass index can be written as:

$$BMI = \frac{weight(kilograms)}{height(meters)^2}$$

You can use `mutate()` to calculate BMI in the `yao` dataset as follows:

```
yao %>%
  select(weight_kg, height_cm) %>%

  # first obtain the height in meters
  mutate(height_meters = height_cm/100) %>%

  # then use the BMI formula
  mutate(bmi = weight_kg / (height_meters)^2)
```

```
# A tibble: 5 x 4
  weight_kg height_cm height_meters  bmi
    <dbl>    <dbl>      <dbl> <dbl>
1      95      169         1.69  33.3
2      96      185         1.85  28.0
3      74      180         1.8   22.8
4      70      164         1.64  26.0
5      67      147         1.47  31.0
```

Let's save the data frame with BMIs for later. We will use it in the next section.

```
yao_bmi <-
  yao %>%
  select(weight_kg, height_cm) %>%
  # first obtain the height in meters
  mutate(height_meters = height_cm/100) %>%
  # then use the BMI formula
  mutate(bmi = weight_kg / (height_meters)^2)
```

 Practice

Appendicular muscle mass (ASM), a useful health indicator, is the sum of muscle mass in all 4 limbs. It can be predicted with the following formula, called Lee's equation:

$$ASM(kg) = (0.244 \times weight(kg)) + (7.8 \times height(m)) + (6.6 \times sex) - (0.098 \times age) - 4.5$$

The sex variable in the formula assumes that men are coded as 1 and women are coded as 0 (which is already the case for our sarcopenia dataset.) The $- 4.5$ at the end is a constant used for Asians. Calculate the ASM value for all individuals in the sarcopenia dataset. This value should be in a new column called `asm`

```
## Complete the code with your answer:
Q_asm_calculation <-
  sarcopenia #_____
  #_____
```

12.8 Changing a variable's type

In your data analysis workflow, you often need to redefine variable *types*. You can do so with functions like `as.integer()`, `as.factor()`, `as.character()` and `as.Date()` within your `mutate()` call. Let's see one example of this.

12.8.1 Integer: `as.integer`

`as.integer()` converts any numeric values to integers:

```
yao_bmi %>%
  mutate(bmi_integer = as.integer(bmi))
```

```
# A tibble: 5 x 5
  weight_kg height_cm height_meters   bmi bmi_integer
    <dbl>      <dbl>        <dbl> <dbl>      <int>
1      95        169          1.69  33.3         33
2      96        185          1.85  28.0         28
3      74        180          1.8   22.8         22
4      70        164          1.64  26.0         26
5      67        147          1.47  31.0         31
```

Note that this *truncates* integers rather than rounding them up or down, as you might expect. For example the BMI 22.8 in the third row is truncated to 22. If you want rounded numbers, you can use the `round` function from base R

 Pro Tip

Using `as.integer()` on a factor variable is a fast way of encoding strings into numbers. It can be essential to do so for some machine learning data processing.

```
yao_bmi %>%
  mutate(bmi_integer = as.integer(bmi),
         bmi_rounded = round(bmi))
```

A tibble: 5 x 6

	weight_kg <dbl>	height_cm <dbl>	height_meters <dbl>	bmi <dbl>	bmi_integer <int>	bmi_rounded <dbl>
1	95	169	1.69	33.3	33	33
2	96	185	1.85	28.0	28	28
3	74	180	1.8	22.8	22	23
4	70	164	1.64	26.0	26	26
5	67	147	1.47	31.0	31	31

Side Note

The base R `round()` function rounds “half down”. That is, the number 3.5, for example, is rounded down to 3 by `round()`. This is weird. Most people expect 3.5 to be rounded *up* to 4, not down to 3. So most of the time, you’ll actually want to use the `round_half_up()` function from `janitor`.

Challenge

In future lessons, you will discover how to manipulate dates and how to convert to a date type using `as.Date()`.

Practice

Use `as_integer()` to convert the ages of respondents in the `sarcopenia` dataset to integers (truncating them in the process). This should go in a new column called `age_integer`

```
## Complete the code with your answer:
Q_age_integer <-
  sarcopenia #_____
  #_____
```

12.9 Wrap up

As you can imagine, transforming data is an essential step in any data analysis workflow. It is often required to clean data and to prepare it for further statistical analysis or for making plots. And as you have seen, it is quite simple to transform data with `dplyr`’s `mutate()` function, although certain transformations are trickier to achieve than others.

Congrats on making it through.

But your data wrangling journey isn’t over yet! In our next lessons, we will learn how to create complex data summaries and how to create and work with data frame groups. Intrigued? See you in the next lesson.

References

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

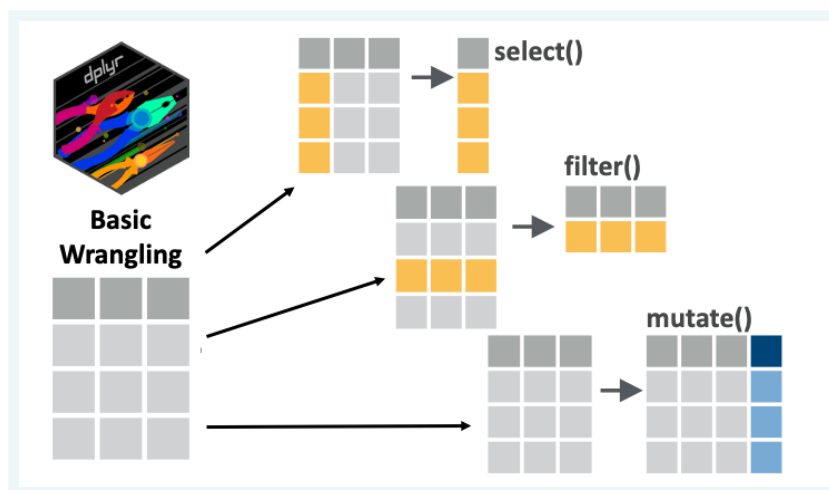


Figure 12.3: Fig: Basic Data Wrangling with `select()`, `filter()`, and `mutate()`.

- Horst, A. (2022). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)
- *Create, modify, and delete columns — Mutate*. (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/mutate.html>
- *Apply a function (or functions) across multiple columns — Across*. (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/across.html>

Artwork was adapted from:

- Horst, A. (2022). *R & stats illustrations by Allison Horst*. <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

Other references:

- Lee, Robert C, ZiMian Wang, Moonseong Heo, Robert Ross, Ian Janssen, and Steven B Heymsfield. "Total-Body Skeletal Muscle Mass: Development and Cross-Validation of Anthropometric Prediction Models." *The American Journal of Clinical Nutrition* 72, no. 3 (2000): 796–803. <https://doi.org/10.1093/ajcn/72.3.796>.

12.10 Solutions

```
.SOLUTION_Q_weight_to_g()
```

```
Q_weight_to_g <-
  sarcopenia %>%
  mutate(weight_grams = weight_kg*1000)
```

```
.SOLUTION_Q_sarcopenia_resp_id()
```

```
Q_sarcopenia_resp_id <-
  sarcopenia %>%
  mutate(respondent_id = 1:n())
```

```
.SOLUTION_Q_women_low_grip_strength()
```

```
Q_women_low_grip_strength <-
  sarcopenia %>%
  filter(sex_male_1_female_0 == 0) %>%
  mutate(low_grip_strength = grip_strength_kg < 20)
```

```
.SOLUTION_Q_prop_women_low_grip_strength()
```

```
Q_prop_women_low_grip_strength <-
  sarcopenia %>%
  filter(sex_male_1_female_0 == 0) %>%
  mutate(low_grip_strength = grip_strength_kg < 20) %>%
  tabyl(low_grip_strength) %>%
  .[2,3] * 100
```

```
.SOLUTION_Q_asm_calculation()
```

```
Q_asm_calculation <-
  sarcopenia %>%
  mutate(asm = 0.244 * weight_kg + 7.8 * height_meters + 6.6 * sex_male_1_female_0 - 0.098 * age)
```

```
.SOLUTION_Q_age_integer()
```

```
Q_age_integer <-
  sarcopenia %>%
  mutate(age_integer = as.integer(age))
```

Chapter 13

Conditional mutating

13.1 Introduction

In the last lesson, you learned the basics of data transformation using the {dplyr} function `mutate()`.

In that lesson, we mostly looked at *global* transformations; that is, transformations that did the same thing to an entire variable. In this lesson, we will look at how to *conditionally* manipulate certain rows based on whether or not they meet defined criteria.

For this, we will mostly use the `case_when()` function, which you will likely come to see as one of the most important functions in {dplyr} for data wrangling tasks.

Let's get started.

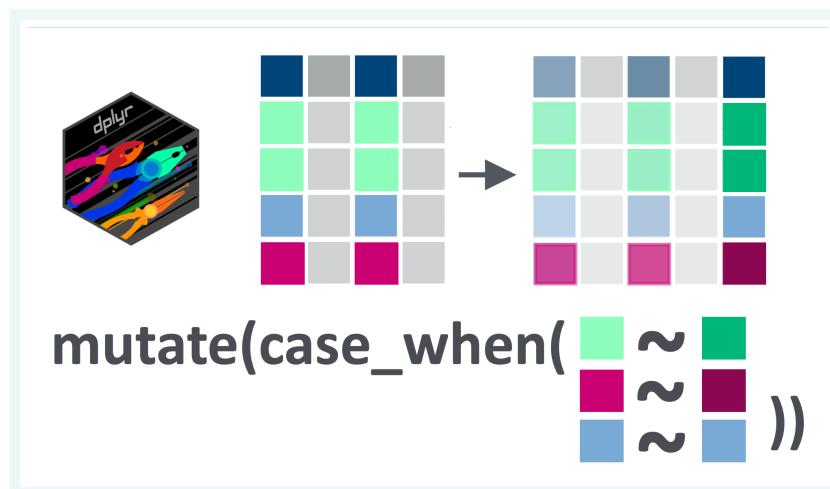


Figure 13.1: Fig: the `case_when()` conditions.

13.2 Learning objectives

1. You can transform or create new variables based on conditions using `dplyr::case_when()`
2. You know how to use the `TRUE` condition in `case_when()` to match unmatched cases.
3. You can handle `NA` values in `case_when()` transformations.
4. You understand how to keep the default values of a variable in a `case_when()` formula

5. You can write `case_when()` conditions involving multiple comparators and multiple variables.
6. You understand `case_when()` conditions priority order.
7. You can use `dplyr::if_else()` for binary conditional assignment.

13.3 Packages

This lesson will require the tidyverse suite of packages:

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

13.4 Datasets

In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
## Import and view the dataset
yaounde <-
  read_csv(here::here('data/yaounde_data.csv')) %>%
  ## make every 5th age missing
  mutate(age = case_when(row_number() %in% seq(5, 900, by = 5) ~ NA_real_,
                        TRUE ~ age)) %>%
  ## rename the age variable
  rename(age_years = age) %>%
  # drop the age category column
  select(-age_category)

yaounde
```

A tibble: 10 x 52

	id	date_surveyed	age_years	age_category_3	sex	highest_education
	<chr>	<date>	<dbl>	<chr>	<chr>	<chr>
1	BRIQUETERIE_0~	2020-10-22	45	Adult	Fema~	Secondary
2	BRIQUETERIE_0~	2020-10-24	55	Adult	Male	University
3	BRIQUETERIE_0~	2020-10-24	23	Adult	Male	University
4	BRIQUETERIE_0~	2020-10-22	20	Adult	Fema~	Secondary
5	BRIQUETERIE_0~	2020-10-22	NA	Adult	Fema~	Primary
6	BRIQUETERIE_0~	2020-10-25	17	Child	Fema~	Secondary
7	BRIQUETERIE_0~	2020-10-25	13	Child	Fema~	Secondary
8	BRIQUETERIE_0~	2020-10-24	28	Adult	Male	Doctorate
9	BRIQUETERIE_0~	2020-10-24	30	Adult	Male	Secondary
10	BRIQUETERIE_0~	2020-10-24	NA	Child	Fema~	Secondary

i 46 more variables: occupation <chr>, weight_kg <dbl>, height_cm <dbl>,
 # is_smoker <chr>, is_pregnant <chr>, is_medicated <chr>, neighborhood <chr>,
 # household_with_children <chr>, breadwinner <chr>, source_of_revenue <chr>,
 # has_contact_covid <chr>, igg_result <chr>, igm_result <chr>,
 # symptoms <chr>, symp_fever <chr>, symp_headache <chr>, symp_cough <chr>,
 # symp_rhinitis <chr>, symp_sneezing <chr>, symp_fatigue <chr>,
 # symp_muscle_pain <chr>, symp_nausea_or_vomiting <chr>, ...

Note that in the code chunk above, we slightly modified the age column, artificially introducing some missing values, and we also dropped the age_category column. This is to help illustrate some key points in the tutorial.

For practice questions, we will also use an outbreak linelist of 136 cases of influenza A H7N9 from a [2013 outbreak](#) in China. This is a modified version of a dataset compiled by Kucharski et al. (2014).

```
## Import and view the dataset
flu_linelist <- read_csv(here::here('data/flu_h7n9_china_2013.csv'))
flu_linelist
```

```
# A tibble: 10 x 8
  case_id date_of_onset date_of_hospitalisation date_of_outcome outcome gender
  <dbl>   <date>         <date>         <date>         <chr>   <chr>
1     1 2013-02-19      NA             2013-03-04      Death   m
2     2 2013-02-27      2013-03-03      2013-03-10      Death   m
3     3 2013-03-09      2013-03-19      2013-04-09      Death   f
4     4 2013-03-19      2013-03-27      NA             <NA>     f
5     5 2013-03-19      2013-03-30      2013-05-15      Recover f
6     6 2013-03-21      2013-03-28      2013-04-26      Death   f
7     7 2013-03-20      2013-03-29      2013-04-09      Death   m
8     8 2013-03-07      2013-03-18      2013-03-27      Death   m
9     9 2013-03-25      2013-03-25      NA             <NA>     m
10    10 2013-03-28      2013-04-01      2013-04-03      Death   m
# i 2 more variables: age <dbl>, province <chr>
```

13.5 Reminder: relational operators (comparators) in R

Throughout this lesson, you will use a lot of relational operators in R. Recall that relational operators, sometimes called “comparators”, test the relation between two values, and return TRUE, FALSE or NA.

A list of the most common operators is given below:

Operator	is TRUE if
A < B	A is less than B
A <= B	A is less than or equal to B
A > B	A is greater than B
A >= B	A is greater than or equal to B
A == B	A is equal to B
A != B	A is not equal to B
A %in% B	A is an element of B

13.6 Introduction to case_when()

To get familiar with case_when(), let’s begin with a simple conditional transformation on the age_years column of the yaounde dataset. First we subset the data frame to just the age_years column for easy illustration:

```
yaounde_age <-
  yaounde %>%
    select(age_years)

yaounde_age
```

```
# A tibble: 10 x 1
  age_years
  <dbl>
1       45
2       55
3       23
4       20
5       NA
6       17
7       13
8       28
9       30
10      NA
```

Now, using `case_when()`, we can make a new column, called “age_group”, that has the value “Child” if the person is below 18, and “Adult” if the person is 18 and up:

```
yaounde_age %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                                age_years >= 18 ~ "Adult"))
```

```
# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1       45 Adult
2       55 Adult
3       23 Adult
4       20 Adult
5       NA <NA>
6       17 Child
7       13 Child
8       28 Adult
9       30 Adult
10      NA <NA>
```

The `case_when()` syntax may seem a bit foreign, but it is quite simple: on the left-hand side (LHS) of the `~` sign (called a “tilde”), you provide the condition(s) you want to evaluate, and on the right-hand side (RHS), you provide a value to put in if the condition is true.

So the statement `case_when(age_years < 18 ~ "Child", age_years >= 18 ~ "Adult")` can be read as: “if `age_years` is below 18, input ‘Child’, else if `age_years` is greater than or equal to 18, input ‘Adult’”.

i Vocab**Formulas, LHS and RHS**

Each line of a `case_when()` call is termed a “formula” or, sometimes, a “two-sided formula”. And each formula has a left-hand side (abbreviated LHS) and right-hand side (abbreviated RHS).

For example, the code `age_years < 18 ~ "Child"` is a “formula”, its LHS is `age_years < 18` while its RHS is `"Child"`.

You are likely to come across these terms when reading the documentation for the `case_when()` function, and we will also refer to them in this lesson.

After creating a new variable with `case_when()`, it is a good idea to inspect it thoroughly to make sure it worked as intended.

To inspect the variable, you can pipe your data frame into the `View()` function to view it in spreadsheet form:

```
yaounde_age %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                               age_years >= 18 ~ "Adult")) %>%
  View()
```

This would open up a new tab in RStudio where you should manually scan through the new column, `age_group` and the referenced column `age_years` to make sure your `case_when()` statement did what you wanted it to do.

You could also pass the new column into the `tabyl()` function to ensure that the proportions “make sense”:

```
yaounde_age %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                               age_years >= 18 ~ "Adult")) %>%
  tabyl(age_group)
```

age_group	n	percent	valid_percent
Adult	558	0.5746653	0.7054362
Child	233	0.2399588	0.2945638
<NA>	180	0.1853759	NA

💡 Practice

With the `flu_linelist` data, make a new column, called `age_group`, that has the value “Below 50” for people under 50 and “50 and above” for people aged 50 and up. Use the `case_when()` function.

```
## Complete the code with your answer:
Q_age_group <-
  flu_linelist %>%
  mutate(age_group = _____)
```

Out of the entire sample of individuals in the `flu_linelist` dataset, what percentage are confirmed to be below 60? (Repeat the above procedure but with the 60 cutoff, then call `tabyl()` on the age group variable. Use the `percent` column, not the `valid_percent` column.)

```
## Enter your answer as a WHOLE number without quotes:
Q_age_group_percentage <- YOUR_ANSWER_HERE
```

13.7 The TRUE default argument

In a `case_when()` statement, you can use a literal `TRUE` condition to match any rows not yet matched with provided conditions.

For example, if we only keep only the first condition from the previous example, `age_years < 18`, and define the default value to be `TRUE ~ "Not child"` then all adults and NA values in the data set will be labeled "Not child" by default.

```
yaounde_age %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                                TRUE ~ "Not child"))
```

```
# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1      45 Not child
2      55 Not child
3      23 Not child
4      20 Not child
5      NA Not child
6      17 Child
7      13 Child
8      28 Not child
9      30 Not child
10     NA Not child
```

This `TRUE` condition can be read as “for everything else...”.

So the full `case_when()` statement used above, `age_years < 18 ~ "Child"`, `TRUE ~ "Not child"`, would then be read as: “if age is below 18, input ‘Child’ and *for everyone else not yet matched*, input ‘Not child’”.

Watch Out

It is important to use `TRUE` as the *final* condition in `case_when()`. If you use it as the first condition, it will take precedence over all others, as seen here:

```
yaounde_age %>%
  mutate(age_group = case_when(TRUE ~ "Not child",
                                age_years < 18 ~ "Child"))
```

```
# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1      45 Not child
```

```

2      55 Not child
3      23 Not child
4      20 Not child
5      NA Not child
6      17 Not child
7      13 Not child
8      28 Not child
9      30 Not child
10     NA Not child

```

As you can observe, all individuals are now coded with “Not child”, because the TRUE condition was placed first, and therefore took precedence. We will explore the issue of precedence further below.

13.8 Matching NA's with is.na()

We can match missing values manually with `is.na()`. Below we match NA ages with `is.na()` and set their age group to “Missing age”:

```

yaounde_age %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                                age_years >= 18 ~ "Adult",
                                is.na(age_years) ~ "Missing age"))

```

```

# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1      45 Adult
2      55 Adult
3      23 Adult
4      20 Adult
5      NA Missing age
6      17 Child
7      13 Child
8      28 Adult
9      30 Adult
10     NA Missing age

```

Practice

As before, using the `flu_linelist` data, make a new column, called `age_group`, that has the value “Below 60” for people under 60 and “60 and above” for people aged 60 and up. But this time, also set those with missing ages to “Missing age”.

```

## Complete the code with your answer:
Q_age_group_nas <-
  flu_linelist %>%

```

 Practice

The gender column of the flu_linelist dataset contains the values “f”, “m” and NA:

```
flu_linelist %>%
  tabyl(gender)
```

```
gender  n    percent valid_percent
f 39 0.28676471      0.2910448
m 95 0.69852941      0.7089552
<NA>  2 0.01470588           NA
```

Recode “f”, “m” and NA to “Female”, “Male” and “Missing gender” respectively. You should modify the existing gender column, not create a new column.

```
## Complete the code with your answer:
Q_gender_recode <-
  flu_linelist %>%
```

13.9 Keeping default values of a variable

The right-hand side (RHS) of a `case_when()` formula can also take in a variable from your data frame. This is often useful when you want to change just a few values in a column.

Let’s see an example with the `highest_education` column, which contains the highest education level attained by a respondent:

```
yaounde_educ <-
  yaounde %>%
    select(highest_education)
yaounde_educ
```

```
# A tibble: 10 x 1
  highest_education
  <chr>
1 Secondary
2 University
3 University
4 Secondary
5 Primary
6 Secondary
7 Secondary
8 Doctorate
9 Secondary
10 Secondary
```

Below, we create a new column, `highest_educ_recode`, where we recode both “University” and “Doctorate” to the value “Post-secondary”:

```
yaounde_educ %>%
  mutate(
    highest_educ_recode =
      case_when(
        highest_education %in% c("University", "Doctorate") ~ "Post-secondary"
      )
  )
```

```
# A tibble: 10 x 2
  highest_education highest_educ_recode
  <chr>             <chr>
1 Secondary         <NA>
2 University        Post-secondary
3 University        Post-secondary
4 Secondary         <NA>
5 Primary           <NA>
6 Secondary         <NA>
7 Secondary         <NA>
8 Doctorate         Post-secondary
9 Secondary         <NA>
10 Secondary        <NA>
```

It worked, but now we have NAs for all other rows. To keep these other rows at their default values, we can add the line `TRUE ~ highest_education` (with a variable, `highest_education`, on the right-hand side of a formula):

```
yaounde_educ %>%
  mutate(
    highest_educ_recode =
      case_when(
        highest_education %in% c("University", "Doctorate") ~ "Post-secondary",
        TRUE ~ highest_education
      )
  )
```

```
# A tibble: 10 x 2
  highest_education highest_educ_recode
  <chr>             <chr>
1 Secondary         Secondary
2 University        Post-secondary
3 University        Post-secondary
4 Secondary         Secondary
5 Primary           Primary
6 Secondary         Secondary
7 Secondary         Secondary
8 Doctorate         Post-secondary
9 Secondary         Secondary
10 Secondary        Secondary
```

Now the `case_when()` statement reads: 'If highest education is "University" or "Doctorate", input "Post-secondary". For everyone else, input the value from `highest_education`'.

Above we have been putting the recoded values in a separate column, `highest_educ_recode`, but for this kind of replacement, it is more common to simply overwrite the existing column:

```
yaounde_educ %>%
  mutate(
    highest_education =
      case_when(
        highest_education %in% c("University", "Doctorate") ~ "Post-secondary",
        TRUE ~ highest_education
      )
  )
```

```
# A tibble: 10 x 1
  highest_education
  <chr>
1 Secondary
2 Post-secondary
3 Post-secondary
4 Secondary
5 Primary
6 Secondary
7 Secondary
8 Post-secondary
9 Secondary
10 Secondary
```

We can read this last `case_when()` statement as: ‘If highest education is “University” or “Doctorate”, *change the value to “Post-secondary”*. For everyone else, *leave in the value from highest_education*’.

Practice

Using the `flu_linelist` data, modify the existing column `outcome` by replacing the value “Recover” with “Recovery”.

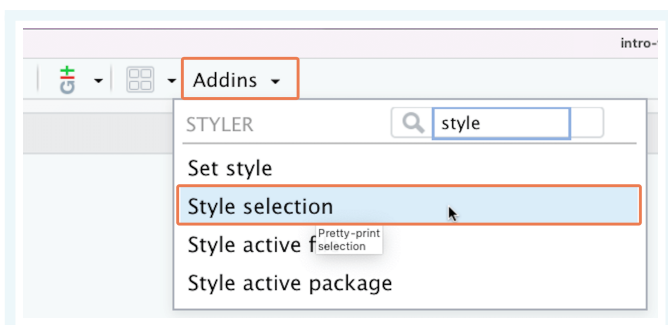
```
## Complete the code with your answer:
Q_recode_recovery <-
  flu_linelist
```

(We know it’s a lot of code for such a simple change. Later you will see easier ways to do this.)

Pro Tip

Avoiding long code lines As you start to write increasingly complex `case_when()` statements, it will become helpful to use line breaks to avoid long lines of code.

To assist with creating line breaks, you can use the `{styler}` package. Install it with `pacman::p_load(styler)`. Then to reformat any piece of code, highlight the code, click the “Addins” button in RStudio, then click on “Style selection”:



Alternatively, you could highlight the code and use the shortcut Shift + Command/Control + A to use RStudio's built-in code reformatter.

Sometimes {styler} does a better job at reformatting. Sometimes the built-in reformatter does a better job.

13.10 Multiple conditions on a single variable

LHS conditions in `case_when()` formulas can have multiple parts. Let's see an example of this.

But first, we will inspire ourselves from what we learnt in the `mutate()` lesson and recreate the BMI variable. This involves first converting the `height_cm` variable to meters, then calculating BMI.

```
yaounde_BMI <-
  yaounde %>%
  mutate(height_m = height_cm/100,
         BMI = (weight_kg / (height_m)^2)) %>%
  select(BMI)

yaounde_BMI
```

```
# A tibble: 10 x 1
  BMI
<dbl>
1  33.3
2  28.0
3  22.8
4  26.0
5  31.0
6  24.8
7  28.9
8  20.7
9  25.3
10 23.9
```

Recall the following BMI categories:

- If the BMI is inferior to 18.5, the person is considered underweight.
- A normal BMI is greater than or equal to 18.5 and less than 25.
- An overweight BMI is greater than or equal to 25 and less than 30.

- An obese BMI is BMI is greater than or equal to 30.

The condition `BMI >= 18.5 & BMI < 25` to define Normal weight is a compound condition because it has *two* comparators: `>=` and `<`.

```
yaounde_BMI <-
  yaounde_BMI %>%
  mutate(BMI_classification = case_when(
    BMI < 18.5 ~ 'Underweight',
    BMI >= 18.5 & BMI < 25 ~ 'Normal weight',
    BMI >= 25 & BMI < 30 ~ 'Overweight',
    BMI >= 30 ~ 'Obese'))

yaounde_BMI
```

```
# A tibble: 10 x 2
  BMI BMI_classification
<dbl> <chr>
1  33.3 Obese
2  28.0 Overweight
3  22.8 Normal weight
4  26.0 Overweight
5  31.0 Obese
6  24.8 Normal weight
7  28.9 Overweight
8  20.7 Normal weight
9  25.3 Overweight
10 23.9 Normal weight
```

Let's use `tabyl()` to have a look at our data:

```
yaounde_BMI %>%
  tabyl(BMI_classification)
```

But you can see that the levels of BMI are defined in alphabetical order from Normal weight to Underweight, instead of from lightest (Underweight) to heaviest (Obese). Remember that if you want to have a certain order you can make `BMI_classification` a factor using `mutate()` and define its levels.

```
yaounde_BMI %>%
  mutate(BMI_classification = factor(
    BMI_classification,
    levels = c("Obese",
               "Overweight",
               "Normal weight",
               "Underweight")
  )) %>%
  tabyl(BMI_classification)
```

 Watch Out

With compound conditions, you should remember to input the variable name *everytime* there is a comparator. R learners often forget this and will try to run code that looks like this:

```
yaounde_BMI %>%
  mutate(BMI_classification = case_when(BMI < 18.5 ~ 'Underweight',
                                         BMI >= 18.5 & < 25 ~ 'Normal weight',
                                         BMI >= 25 & < 30 ~ 'Overweight',
                                         BMI >= 30 ~ 'Obese'))
```

The definitions for the “Normal weight” and “Overweight” categories are mistaken. Do you see the problem? Try to run the code to spot the error.

 Practice

With the `flu_linelist` data, make a new column, called `adolescent`, that has the value “Yes” for people in the 10-19 (at least 10 and less than 20) age group, and “No” for everyone else.

```
## Complete the code with your answer:
Q_adolescent_grouping <-
  flu_linelist %>%
```

13.11 Multiple conditions on multiple variables

In all examples seen so far, you have only used conditions involving a single variable at a time. But LHS conditions often refer to multiple variables at once.

Let’s see a simple example with age and sex in the `yaounde` data frame. First, we select just these two variables for easy illustration:

```
yaounde_age_sex <-
  yaounde %>%
  select(age_years, sex)

yaounde_age_sex
```

```
# A tibble: 10 x 2
  age_years sex
  <dbl> <chr>
1      45 Female
2      55 Male
3      23 Male
4      20 Female
5      NA Female
6      17 Female
7      13 Female
8      28 Male
9      30 Male
10     NA Female
```

Now, imagine we want to recruit women and men in the 20-29 age group into two studies. For this we'd like to create a column, called `recruit`, with the following schema:

- Women aged 20-29 should have the value "Recruit to female study"
- Men aged 20-29 should have the value "Recruit to male study"
- Everyone else should have the value "Do not recruit"

To do this, we run the following `case_when` statement:

```
yaounde_age_sex %>%
  mutate(recruit = case_when(
    sex == "Female" & age_years >= 20 & age_years <= 29 ~ "Recruit to female study",
    sex == "Male" & age_years >= 20 & age_years <= 29 ~ "Recruit to male study",
    TRUE ~ "Do not recruit"
  ))
```

A tibble: 10 x 3

	age_years	sex	recruit
	<dbl>	<chr>	<chr>
1	45	Female	Do not recruit
2	55	Male	Do not recruit
3	23	Male	Recruit to male study
4	20	Female	Recruit to female study
5	NA	Female	Do not recruit
6	17	Female	Do not recruit
7	13	Female	Do not recruit
8	28	Male	Recruit to male study
9	30	Male	Do not recruit
10	NA	Female	Do not recruit

You could also add extra pairs of parentheses around the age criteria within each condition:

```
yaounde_age_sex %>%
  mutate(recruit = case_when(
    sex == "Female" & (age_years >= 20 & age_years <= 29) ~ "Recruit to female study",
    sex == "Male" & (age_years >= 20 & age_years <= 29) ~ "Recruit to male study",
    TRUE ~ "Do not recruit"
  ))
```

This extra pair of parentheses does not change the code output, but it improves coherence because the reader can visually see that your condition is made of two parts, one for gender, `sex == "Female"`, and another for age, `(age_years >= 20 & age_years <= 29)`.

Practice

With the `flu_linelist` data, make a new column, called `recruit` with the following schema:

- Individuals aged 30-59 (at least 30, younger than 60) from the Jiangsu province should have the value "Recruit to Jiangsu study"
- Individuals aged 30-59 from the Zhejiang province should have the value "Recruit to Zhejiang study"
- Everyone else should have the value "Do not recruit"

```
## Complete the code with your answer:
Q_age_province_grouping <-
  flu_linelist %>%
  mutate(recruit = _____)
```

13.12 Order of priority of conditions in case_when()

Note that the order of conditions is important, because conditions listed at the top of your `case_when()` statement take priority over others.

To understand this, run the example below:

```
yaounde_age_sex %>%
  mutate(age_group = case_when(
    age_years < 18 ~ "Child",
    age_years < 30 ~ "Young adult",
    age_years < 120 ~ "Older adult"))
```

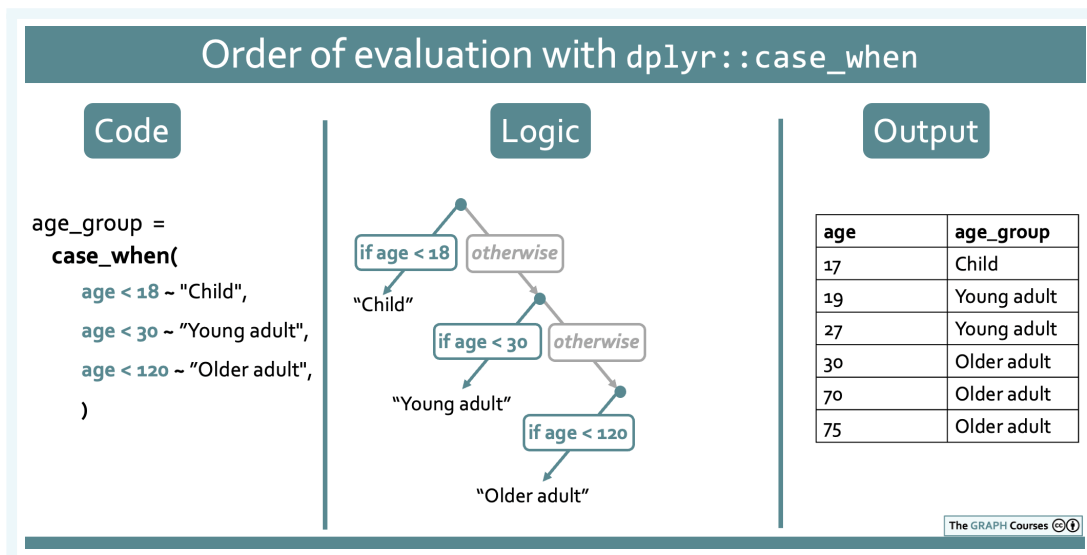
```
# A tibble: 10 x 3
  age_years sex    age_group
  <dbl> <chr> <chr>
1      45 Female Older adult
2      55 Male  Older adult
3      23 Male  Young adult
4      20 Female Young adult
5      NA Female <NA>
6      17 Female Child
7      13 Female Child
8      28 Male  Young adult
9      30 Male  Older adult
10     NA Female <NA>
```

This initially looks like a faulty `case_when()` statement because the age conditions overlap. For example, the statement `age_years < 120 ~ "Older adult"` (which reads “if age is below 120, input ‘Older adult’”) suggests that *anyone* between ages 0 and 120 (even a 1-year old baby!, would be coded as “Older adult”.

But as you saw, the code actually works fine! People under 18 are still coded as “Child”.

What’s going on? Essentially, the `case_when()` statement is interpreted as a series of branching logical steps, starting with the first condition. So this particular statement can be read as: “If age is below 18, input ‘Child’, *and otherwise*, if age is below 30, input ‘Young adult’, *and otherwise*, if age is below 120, input ‘Older adult’”.

This is illustrated in the schematic below:



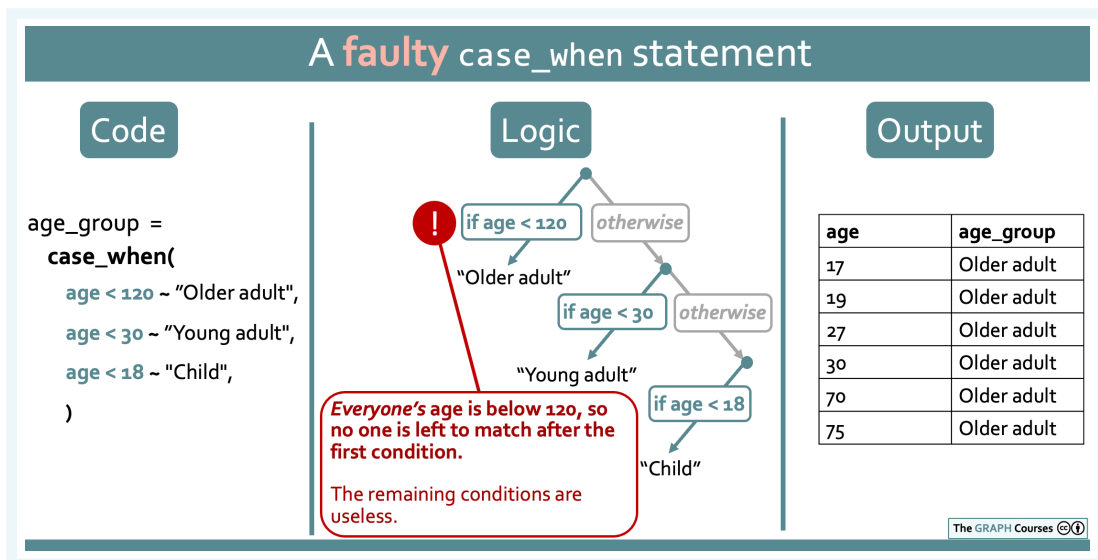
This means that if you swap the order of the conditions, you will end up with a faulty `case_when()` statement:

```
yaounde_age %>%
  mutate(age_group = case_when(
    age_years < 120 ~ "Older adult",
    age_years < 30 ~ "Young adult",
    age_years < 18 ~ "Child"))
```

```
# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1      45 Older adult
2      55 Older adult
3      23 Older adult
4      20 Older adult
5      NA <NA>
6      17 Older adult
7      13 Older adult
8      28 Older adult
9      30 Older adult
10     NA <NA>
```

As you can see, everyone is coded as "Older adult". This happens because the first condition matches everyone, so there is no one left to match with the subsequent conditions. The statement can be read "If age is below 120, input 'Older adult', and otherwise if age is below 30...." But there is no "otherwise" because everyone has already been matched!

This is illustrated in the diagram below:



Although we have spent much time explaining the importance of the order of conditions, in this specific example, there would be a much clearer way to write this code that would not depend on the order of conditions. Rather than leave the age groups open-ended like this:

```
age_years < 120 ~ "Older adult"
```

you should actually use *closed* age bounds like this:

```
age_years >= 30 & age_years < 120 ~ "Older adult"
```

which is read: "if age is greater than or equal to 30 and less than 120, input 'Older adult'".

With such closed conditions, the order of conditions no longer matters. You get the same result no matter how you arrange the conditions:

```
## start with "Older adult" condition
yaounde_age %>%
  mutate(age_group = case_when(
    age_years >= 30 & age_years < 120 ~ "Older adult",
    age_years >= 18 & age_years < 30 ~ "Young adult",
    age_years >= 0 & age_years < 18 ~ "Child"
  ))
```

```
# A tibble: 10 x 2
  age_years age_group
  <dbl> <chr>
1      45 Older adult
2      55 Older adult
3      23 Young adult
4      20 Young adult
5      NA <NA>
6      17 Child
7      13 Child
8      28 Young adult
9      30 Older adult
10     NA <NA>
```

```
## start with "Child" condition
yaounde_age %>%
  mutate(age_group = case_when(
    age_years >= 0 & age_years < 18 ~ "Child",
    age_years >= 18 & age_years < 30 ~ "Young adult",
    age_years >= 30 & age_years < 120 ~ "Older adult"
  ))
```

```
# A tibble: 10 x 2
  age_years age_group
    <dbl>   <chr>
1      45 Older adult
2      55 Older adult
3      23 Young adult
4      20 Young adult
5      NA <NA>
6      17 Child
7      13 Child
8      28 Young adult
9      30 Older adult
10     NA <NA>
```

Nice and clean!

So why did we spend so much time explaining the importance of condition order if you can simply avoid open-ended categories and not have to worry about condition order?

One reason is that understanding condition order should now help you see why it is important to put the TRUE condition as the final line in your `case_when()` statement. The TRUE condition matches *every row that has not yet been matched*, so if you use it first in the `case_when()`, it will match *everyone*!

The other reason is that there are certain cases where you *may* want to use open-ended overlapping conditions, and so you will have to pay attention to the order of conditions. Let's see one such example now: identifying COVID-like symptoms. Note that this is somewhat advanced material, likely a bit above your current needs. We are introducing it now so you are aware and can stay vigilant with `case_when()` in the future.

13.12.1 Overlapping conditions within `case_when()`

We want to identify COVID-like symptoms in our data. Consider the symptoms columns in the `yaounde` data frame, which indicates which symptoms were experienced by respondents over a 6-month period:

```
yaounde %>%
  select(starts_with("symp_"))
```

```
# A tibble: 10 x 13
  symp_fever symp_headache symp_cough symp_rhinitis symp_sneezing symp_fatigue
  <chr>      <chr>          <chr>      <chr>          <chr>      <chr>
1 No        No            No         No            No        No
2 No        No            No         No            No        No
3 No        No            No         No            No        No
4 No        No            No         Yes           Yes       No
```

5	No	No	No	No	No	No
6	Yes	No	Yes	Yes	No	No
7	No	No	No	No	Yes	No
8	No	Yes	No	No	No	No
9	Yes	No	No	Yes	No	No
10	No	No	No	No	No	No

```
# i 7 more variables: symp_muscle_pain <chr>, symp_nausea_or_vomiting <chr>,
#   symp_diarrhoea <chr>, symp_short_breath <chr>, symp_sore_throat <chr>,
#   symp_anosmia_or_ageusia <chr>, symp_stomach_ache <chr>
```

We would like to use this to assess whether a person may have had COVID, partly following guidelines recommended by the [WHO](#).

- Individuals with cough are to be classed as “possible COVID cases”
- Individuals with anosmia/ageusia (loss of smell or loss of taste) are to be classed as “probable COVID cases”.

Now, keeping these criteria in mind, consider an individual, let’s call her Osma, who has cough AND anosmia/ageusia? How should we classify Osma?

She meets the criteria for “possible COVID” (because she has cough), but she *also* meets the criteria for “probable COVID” (because she has anosmia/ageusia). So which group should she be classed as, “possible COVID” or “probable COVID”? Think about it for a minute.

Hopefully you guessed that she should be classed as a “probable COVID case”. “Probable” is more likely than “Possible”; and the anosmia/ageusia symptom is more *significant* than the cough symptom. One might say that the criterion for “probable COVID” has a higher specificity or a higher *precedence* than the criterion for “possible COVID”.

Therefore, when constructing a `case_when()` statement, the “probable COVID” condition should also take higher precedence—it should come *first* in the conditions provided to `case_when()`. Let’s see this now.

First we select the relevant variables, for easy illustration. We also identify and `slice()` specific rows that are useful for the demonstration:

```
yaounde_symptoms_slice <-
  yaounde %>%
  select(symp_cough, symp_anosmia_or_ageusia) %>%
  # slice of specific rows useful for demo
  # Once you find the right code, you would remove this slice
  slice(32, 711, 625, 651 )

yaounde_symptoms_slice
```

```
# A tibble: 4 x 2
  symp_cough symp_anosmia_or_ageusia
  <chr>      <chr>
1 No       No
2 Yes      No
3 No       Yes
4 Yes      Yes
```

Now, the correct `case_when()` statement, which has the “Probable COVID” condition first:

```
yaounde_symptoms_slice %>%
  mutate(covid_status = case_when(
    symp_anosmia_or_ageusia == "Yes" ~ "Probable COVID",
    symp_cough == "Yes" ~ "Possible COVID"
  ))
```

```
# A tibble: 4 x 3
  symp_cough symp_anosmia_or_ageusia covid_status
  <chr>      <chr>                        <chr>
1 No        No                            <NA>
2 Yes       No                            Possible COVID
3 No        Yes                            Probable COVID
4 Yes       Yes                            Probable COVID
```

This `case_when()` statement can be read in simple terms as ‘If the person has anosmia/ageusia, input “Probable COVID”, and otherwise, if the person has cough, input “Possible COVID”’.

Now, spend some time looking through the output data frame, especially the last three individuals. The individual in row 2 meets the criterion for “Possible COVID” because they have cough (`symp_cough == “Yes”`), and the individual in row 3 meets the criterion for “Probable COVID” because they have anosmia/ageusia (`symp_anosmia_or_ageusia == “Yes”`).

The individual in row 4 is Osma, who both meets the criteria for “possible COVID” *and* for “probable COVID”. And because we arranged our `case_when()` conditions in the right order, she is coded correctly as “probable COVID”. Great!

But notice what happens if we swap the order of the conditions:

```
yaounde_symptoms_slice %>%
  mutate(covid_status = case_when(
    symp_cough == "Yes" ~ "Possible COVID",
    symp_anosmia_or_ageusia == "Yes" ~ "Probable COVID"
  ))
```

```
# A tibble: 4 x 3
  symp_cough symp_anosmia_or_ageusia covid_status
  <chr>      <chr>                        <chr>
1 No        No                            <NA>
2 Yes       No                            Possible COVID
3 No        Yes                            Probable COVID
4 Yes       Yes                            Possible COVID
```

Oh no! Osma in row 4 is now misclassified as “Possible COVID” even though she has the more significant anosmia/ageusia symptom. This is because the first condition `symp_cough == “Yes”` matched her first, and so the second condition was not able to match her!

So now you see why you sometimes need to think deeply about the order of your `case_when()` conditions. It is a minor point, but it can bite you at unexpected times. Even experienced analysts tend to make mistakes that can be traced to improper arrangement of `case_when()` statements.

i Challenge

In reality, there *is* still another solution to avoid misclassifying the person with cough and anosmia/ageusia. That is to add `symp_anosmia_or_ageusia != "Yes"` (not equal to “Yes”) to the conditions for “Possible COVID”. Can you think of why this works?

```
yaounde_symptoms_slice %>%
  mutate(covid_status = case_when(
    symp_cough == "Yes" & symp_anosmia_or_ageusia != "Yes" ~ "Possible COVID",
    symp_anosmia_or_ageusia == "Yes" ~ "Probable COVID"))
```

```
# A tibble: 4 x 3
  symp_cough symp_anosmia_or_ageusia covid_status
  <chr>      <chr>                      <chr>
1 No        No                        <NA>
2 Yes       No                        Possible COVID
3 No        Yes                        Probable COVID
4 Yes       Yes                        Probable COVID
```

💡 Practice

With the `flu_linelist` dataset, create a new column called `follow_up_priority` that implements the following schema:

- Women should be considered “High priority”
- All children (under 18 years) of any gender should be considered “Highest priority”.
- Everyone else should have the value “No priority”

```
## Complete the code with your answer:
Q_priority_groups <-
  flu_linelist %>%
  mutate(follow_up_priority = -----
  )
```

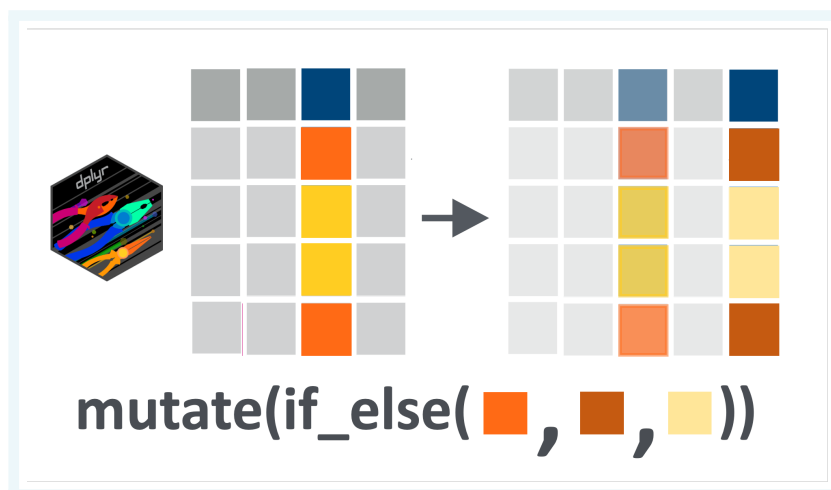
13.13 Binary conditions: `dplyr::if_else()`

There is another {dplyr} verb similar to `case_when()` for when we want to apply a binary condition to a variable: `if_else()`. A binary condition is either TRUE or FALSE.

`if_else()` has a similar application as `case_when()` : if the condition is true, then one operation is applied, if the condition is false, the alternative is applied. The syntax is: `if_else(CONDITION, IF_TRUE, IF_FALSE)`. As you can see, this only allows for a binary condition (not multiple cases, such as handled by `case_when()`).

If we take one of the first examples about recoding the `highest_education` variable, we can write it either with `case_when()` or with `if_else()`.

Here is the version we already explored:

Figure 13.2: Fig: the `if_else()` conditions.

```
yaounde_educ %>%
  mutate(
    highest_education =
      case_when(
        highest_education %in% c("University", "Doctorate") ~ "Post-secondary",
        TRUE ~ highest_education
      )
  )
```

```
# A tibble: 10 x 1
  highest_education
  <chr>
1 Secondary
2 Post-secondary
3 Post-secondary
4 Secondary
5 Primary
6 Secondary
7 Secondary
8 Post-secondary
9 Secondary
10 Secondary
```

And this is how we would write it using `if_else()`:

```
yaounde_educ %>%
  mutate(highest_education =
    if_else(
      highest_education %in% c("University", "Doctorate"),
      # if TRUE then we recode
      "Post-secondary",
      # if FALSE then we keep default value
      highest_education
    )
  )
```

```
# A tibble: 10 x 1
  highest_education
  <chr>
1 Secondary
2 Post-secondary
3 Post-secondary
4 Secondary
5 Primary
6 Secondary
7 Secondary
8 Post-secondary
9 Secondary
10 Secondary
```

As you can see, we get the same output, whether we use `if_else()` or `case_when()`.

Practice

With the `flu_linelist` data, make a new column, called `age_group`, that has the value “Below 50” for people under 50 and “50 and above” for people aged 50 and up. Use the `if_else()` function. This is exactly the same question as your first practice question, but this time you need to use `if_else()`.

```
## Complete the code with your answer:
Q_age_group_if_else <-
  flu_linelist %>%
  mutate(age_group = if_else(_____))
```

13.14 Wrap up

Changing or constructing your variables based on conditions on other variables is one of the most repeated data wrangling tasks. To the point it deserved its very own lesson !

I hope now that you will feel comfortable using `case_when()` and `if_else()` within `mutate()` and that you are excited to learn more complex {dplyr} operations such as grouping variables and summarizing them.

See you next time!

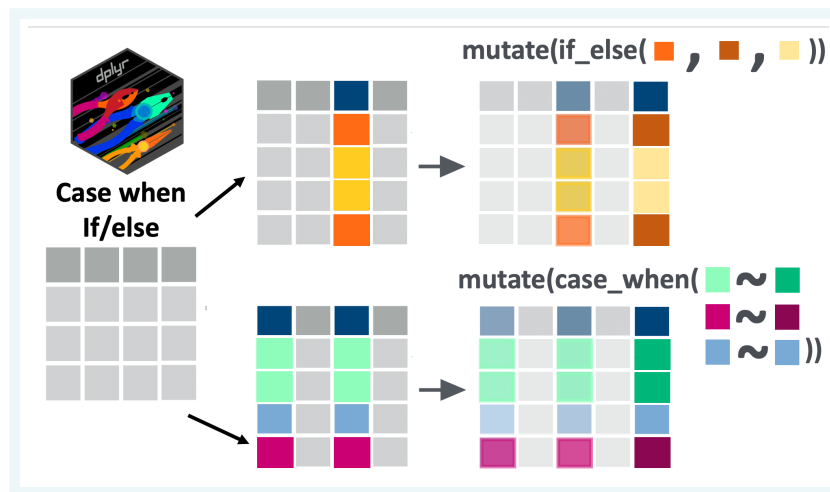
References

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

- Horst, A. (2022). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)
- *Create, modify, and delete columns — Mutate*. (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/mutate.html>

Artwork was adapted from:

- Horst, A. (2022). *R & stats illustrations by Allison Horst*. <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

Figure 13.3: Fig: the `if_else()` and the `'case_when()'` conditions.

13.15 Solutions

```
.SOLUTION_Q_age_group()
```

```
Q_age_group <-
  flu_linelist %>%
  mutate(age_group = case_when(age < 50 ~ "Below 50",
                                age >= 50 ~ "50 and above"))
```

```
.SOLUTION_Q_age_group_percentage()
```

Here is one way (not the only way) to get it:

```
Q_age_group_percentage <-
  flu_linelist %>%
  mutate(age_group_percentage = case_when(age < 60 ~ "Below 60",
                                           age >= 60 ~ "60 and above")) %>%
  tabyl(age_group_percentage) %>%
  filter(age_group_percentage == "Below 60") %>%
  pull(percent) * 100
```

```
.SOLUTION_Q_age_group_nas()
```

```
Q_age_group_nas <-
  flu_linelist %>%
  mutate(age_group = case_when(age < 60 ~ "Below 60",
```

```
age >= 60 ~ "60 and above",
is.na(age) ~ "Missing age"))
```

```
.SOLUTION_Q_gender_recode()
```

```
Q_gender_recode <-
  flu_linelist %>%
    mutate(gender = case_when(gender == "f" ~ "Female",
                              gender == "m" ~ "Male",
                              is.na(gender) ~ "Missing gender"))
```

```
.SOLUTION_Q_recode_recovery()
```

```
Q_recode_recovery <-
  flu_linelist %>%
    mutate(outcome = case_when(outcome == "Recover" ~ "Recovery",
                              TRUE ~ outcome))
```

```
.SOLUTION_Q_adolescent_grouping()
```

```
Q_adolescent_grouping <-
  flu_linelist %>%
    mutate(adolescent = case_when(
      age >= 10 & age < 20 ~ "Yes",
      TRUE ~ "No"))
```

```
.SOLUTION_Q_age_province_grouping()
```

```
Q_age_province_grouping <-
  flu_linelist %>%
    mutate(recruit = case_when(
      province == "Jiangsu" & (age >= 30 & age < 60) ~ "Recruit to Jiangsu study",
      province == "Zhejiang" & (age >= 30 & age < 60) ~ "Recruit to Zhejiang study",
      TRUE ~ "Do not recruit"
    ))
```

```
.SOLUTION_Q_priority_groups()
```

```
Q_priority_groups <-  
  flu_linelist %>%  
    mutate(follow_up_priority = case_when(  
      age < 18 ~ "Highest priority",  
      gender == "f" ~ "High priority",  
      TRUE ~ "No priority"  
    ))
```

```
.SOLUTION_Q_age_group_if_else()
```

```
Q_age_group_if_else <-  
  flu_linelist %>%  
    mutate(age_group = if_else(age < 50, "Below 50", "50 and above"))
```

Chapter 14

Grouping and summarizing data

14.1 Introduction

You currently know how to keep your data entries of interest, how keep relevant variables and how to modify them or create new ones.

Now, we will take your data wrangling skills one step further by understanding how to easily extract summary statistics, through the verb `summarize()`, such as calculating the mean of a variable.

Moreover, we will begin exploring a crucial verb, `group_by()`, capable of grouping your variables together to perform grouped operations on your data set.

Let's go !

14.2 Learning objectives

1. You can use `dplyr::summarize()` to extract summary statistics from datasets.
 2. You can use `dplyr::group_by()` to group data by one or more variables before performing operations on them.
 3. You understand why and how to ungroup grouped data frames.
 4. You can use `dplyr::n()` together with `group_by()-summarize()` to count rows per group.
 5. You can use `sum()` together with `group_by()-summarize()` to count rows that meet a condition.
 6. You can use `dplyr::count()` as a handy function to count rows per group.
-

14.3 The Yaounde COVID-19 dataset

In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
yaounde <- read_csv(here::here('data/yaounde_data.csv'))

## A smaller subset of variables
yao <- yaounde %>% select(
  age, age_category_3, sex, weight_kg, height_cm,
  neighborhood, is_smoker, is_pregnant, occupation,
  treatment_combinations, symptoms, n_days_miss_work, n_bedridden_days,
  highest_education, igg_result)

yao
```

```
# A tibble: 971 x 15
```

	age	age_category_3	sex	weight_kg	height_cm	neighborhood	is_smoker
	<dbl>	<chr>	<chr>	<dbl>	<dbl>	<chr>	<chr>
1	45	Adult	Female	95	169	Briqueterie	Non-smoker
2	55	Adult	Male	96	185	Briqueterie	Ex-smoker
3	23	Adult	Male	74	180	Briqueterie	Smoker
4	20	Adult	Female	70	164	Briqueterie	Non-smoker
5	55	Adult	Female	67	147	Briqueterie	Non-smoker
6	17	Child	Female	65	162	Briqueterie	Non-smoker
7	13	Child	Female	65	150	Briqueterie	Non-smoker
8	28	Adult	Male	62	173	Briqueterie	Non-smoker
9	30	Adult	Male	73	170	Briqueterie	Non-smoker
10	13	Child	Female	56	153	Briqueterie	Non-smoker

```
# i 961 more rows
```

```
# i 8 more variables: is_pregnant <chr>, occupation <chr>,
# treatment_combinations <chr>, symptoms <chr>, n_days_miss_work <dbl>,
# n_bedridden_days <dbl>, highest_education <chr>, igg_result <chr>
```

See the first lesson in this chapter for more information about this dataset.

14.4 What are summary statistics?

A summary statistic is a single value (such as a mean or median) that describes a sequence of values (typically a column in your dataset).

What is a summary statistic?

Summary statistics
describe a sequence
of values...

...with a single value

Age
9
1
3
4
1
1
3



Mean: 3.375

Summary statistics can describe the center, spread or range of a variable, or the counts and positions of values within that variable. Some common summary statistics are shown in the diagram below:

Examples of summary statistics

```
age <- (9, 1, 4, 2, 2, 2)
```

Summary statistic	R code	Output
Counts		
No. of elements	<code>dplyr::n(age)</code>	6
No. of distinct elements	<code>dplyr::n_distinct(age)</code>	4
Position		
First element	<code>dplyr::first(age)</code>	9
Last element	<code>dplyr::last(age)</code>	2
3rd element	<code>dplyr::nth(age, 3)</code>	4
Center		
Mean	<code>mean(age)</code>	3.3
Median	<code>median(age)</code>	2
Spread		
Standard deviation	<code>sd(age)</code>	2.9
Interquartile range	<code>IQR(age)</code>	1.5
Range		
Minimum	<code>min(age)</code>	1
Maximum	<code>max(age)</code>	9
25th quantile	<code>quantile(age, 0.25)</code>	2

Computing summary statistics is a very common operation in most data analysis workflows, so it will be important to become fluent in extracting them from your datasets. And for this task, there is no better tool than the `{dplyr}` function `summarize()`! So let's see how to use this powerful function.

14.5 Introducing dplyr::summarize()

To get started, it is best to first consider how to get simple summary statistics *without* using `summarize()`, then we will consider why you *should* actually use `summarize()`.

Imagine you were asked to find the mean age of respondents in the `yao` data frame. How might you do this in base R?

First, recall that the dollar sign function, `$`, allows you to extract a data frame column to a vector:

```
yao$age # extract the `age` column from `yao`
```

To obtain the mean, you simply pass this `yao$age` vector into the `mean()` function:

```
mean(yao$age)
```

```
[1] 29.01751
```

And that's it! You now have a simple summary statistic. Extremely easy, right?

So why do we need `summarize()` to get summary statistics if the process is already so simple without it? We'll come back to the *why* question soon. First let's see *how* to obtain summary statistics with `summarize()`.

Going back to the previous example, the correct syntax to get the mean age with `summarize()` would be:

```
yao %>%  
  summarize(mean_age = mean(age))
```

```
# A tibble: 1 x 1  
  mean_age  
    <dbl>  
1    29.0
```

The anatomy of this syntax is shown below. You simply need to input name of the new column (e.g. `mean_age`), the summary function (e.g. `mean()`), and the column to summarize (e.g. `age`).

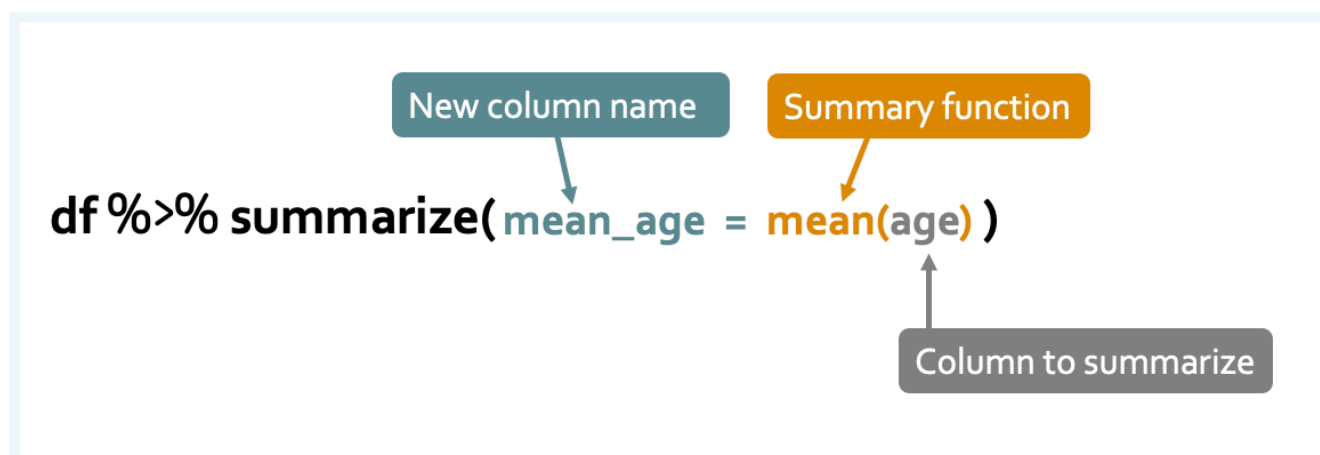


Figure 14.1: Fig. Basic syntax for the `summarize()` function.

You can also compute multiple summary statistics in a single `summarize()` statement. For example, if you wanted both the mean and the median age, you could run:

```
yao %>%
  summarize(mean_age = mean(age),
            median_age = median(age))
```

```
# A tibble: 1 x 2
  mean_age median_age
  <dbl>      <dbl>
1    29.0         26
```

Nice!

Now, you should be wondering why `summarize()` puts the summary statistics into a data frame, with each statistic in a different column.

The main benefit of this data frame structure is to make it easy to produce *grouped* summaries (and creating such grouped summaries will be the primary benefit of using `summarize()`).

We will look at these grouped summaries in the next section. For now, attempt the practice questions below.

Practice

Use `summarize()` and the relevant summary functions to obtain the mean, median and standard deviation of respondent weights from the `weight_kg` variable of the `yao` data frame. Your output should be a data frame with three columns named as shown below:

mean_weight_kg	median_weight_kg	sd_weight_kg
----------------	------------------	--------------

```
Q_weight_summary <-
yao %>%
  -----
```

Practice

Use `summarize()` and the relevant summary functions to obtain the minimum and maximum respondent heights from the `height_cm` variable of the `yao` data frame. Your output should be a data frame with two columns named as shown below:

min_height_cm	max_height_cm
---------------	---------------

```
Q_height_summary <-
  yao %>%
  -----

.CHECK_Q_height_summary()
.HINT_Q_height_summary()
```

14.6 Grouped summaries with dplyr::group_by()

As its name suggests, `dplyr::group_by()` lets you group a data frame by the values in a variable (e.g. male vs female sex). You can then perform operations that are split according to these groups.

What effect does `group_by()` have on a data frame? Let's try to group the `yao` data frame by sex and observe the effect:

```
yao %>%
  group_by(sex)
```

```
# A tibble: 971 x 15
# Groups:   sex [2]
   age age_category_3 sex    weight_kg height_cm neighborhood is_smoker
<dbl> <chr>          <chr>    <dbl>    <dbl> <chr>          <chr>
1    45 Adult      Female      95      169 Briqueterie Non-smoker
2    55 Adult      Male       96      185 Briqueterie Ex-smoker
3    23 Adult      Male       74      180 Briqueterie Smoker
4    20 Adult      Female     70      164 Briqueterie Non-smoker
5    55 Adult      Female     67      147 Briqueterie Non-smoker
6    17 Child      Female     65      162 Briqueterie Non-smoker
7    13 Child      Female     65      150 Briqueterie Non-smoker
8    28 Adult      Male       62      173 Briqueterie Non-smoker
9    30 Adult      Male       73      170 Briqueterie Non-smoker
10   13 Child      Female     56      153 Briqueterie Non-smoker
# i 961 more rows
# i 8 more variables: is_pregnant <chr>, occupation <chr>,
#   treatment_combinations <chr>, symptoms <chr>, n_days_miss_work <dbl>,
#   n_bedridden_days <dbl>, highest_education <chr>, igg_result <chr>
```

Hmm. Apparently nothing happened. The one thing you *might* notice is a new section in the header that tells you the grouped-by variable—sex—and the number of groups—2:

```
# A tibble: 971 x 10
# Groups:   sex [2]
```

Apart from this header however, the data frame appears unchanged.

But watch what happens when we chain the `group_by()` with the `summarize()` call we used in the previous section:

```
yao %>%
  group_by(sex) %>%
  summarize(mean_age = mean(age))
```

```
# A tibble: 2 x 2
  sex    mean_age
<chr>    <dbl>
1 Female    29.5
2 Male     28.4
```

You get a different summary statistic for each group! The statistics for women are in one row and those for men are in another. (From this output data frame, you can tell that, for example, the mean age for female respondents is 29.5, while that for male respondents is 28.4)

As was mentioned earlier, this kind of grouped summary is the primary reason the `summarize()` function is so useful!

Let's see another example of a simple `group_by()` + `summarize()` operation.

Suppose you were asked to obtain the maximum and minimum weights for individuals in different neighborhoods in the `yao` data frame. First you would `group_by()` the neighbourhood variable, then call the `max()` and `min()` functions inside `summarize()`:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(max_weight = max(weight_kg),
            min_weight = min(weight_kg))
```

```
# A tibble: 9 x 3
  neighborhood max_weight min_weight
<chr>          <dbl>      <dbl>
1 Briqueterie    128         20
2 Carriere       129         14
3 Cité Verte    118         16
4 Ekoudou       135         15
5 Messa          96         19
6 Mokolo       162         16
7 Nkomkana      161         15
8 Tsinga        105         15
9 Tsinga Oliga  100         17
```

Great! With just a few code lines you are able to extract quite a lot of information.

Let's see one more example for good measure. The variable `n_days_miss_work` tells us the number of days that respondents missed work due to COVID-like symptoms. Individuals who reported no COVID-like symptoms have an NA for this variable:

```
yao %>%
  select(n_days_miss_work)
```

```
# A tibble: 971 x 1
  n_days_miss_work
      <dbl>
1             0
2            NA
3            NA
4             7
5            NA
6             7
7             0
8             0
9             0
10            NA
# i 961 more rows
```

To count the total number of work days missed for each sex group, you could try to run the `sum()` function on the `n_days_miss_work` variable:

```
yao %>%
  group_by(sex) %>%
  summarise(total_days_missed = sum(n_days_miss_work))
```

```
# A tibble: 2 x 2
  sex      total_days_missed
  <chr>          <dbl>
1 Female            NA
2 Male              NA
```

Hmmm. This gives you NA results because some rows in the `n_days_miss_work` column have NAs in them, and R cannot find the sum of values containing an NA. To solve this, the argument `na.rm = TRUE` is needed:

```
yao %>%
  group_by(sex) %>%
  summarise(total_days_missed = sum(n_days_miss_work, na.rm = TRUE))
```

```
# A tibble: 2 x 2
  sex      total_days_missed
  <chr>          <dbl>
1 Female        256
2 Male          272
```

The output tells us that across all women in the sample, 256 work days were missed due to COVID-like symptoms, and across all men, 272 days.

So hopefully now you see why `summarize()` is so powerful. In combination with `group_by()`, it lets you obtain highly informative grouped summaries of your datasets with very few lines of code.

Producing such summaries is a very important part of most data analysis workflows, so this skill is likely to come in handy soon!

Vocab

`summarize()` **produces “Pivot Tables”**

The summary data frames created by `summarize()` are often called Pivot Tables in the context of spreadsheet software like Microsoft Excel.

Practice

Use `group_by()` and `summarize()` to obtain the mean weight (kg) by smoking status in the `yao` data frame. Name the average weight column `weight_mean`

The output data frame should look like this:

is_smoker	weight_mean
Ex-smoker	
Non-smoker	
Smoker	
NA	

```
Q_weight_by_smoking_status <-
  yao %>%
  -----
  -----
```

Practice

Use `group_by()`, `summarize()`, and the relevant summary functions to obtain the minimum and maximum heights for each sex in the `yao` data frame.

Your output should be a data frame with three columns named as shown below:

sex	min_height_cm	max_height_cm
Female		
Male		

```
Q_min_max_height_by_sex <-
  yao %>%
  -----
  -----
```

Practice

Use `group_by()`, `summarize()`, and the `sum()` function to calculate the total number of bedridden days (from the `n_bedridden_days` variable) reported by respondents of each sex.

Your output should be a data frame with two columns named as shown below:

sex	total_bedridden_days
Female	
Male	

```
Q_sum_bedridden_days <-
  yao %>%
  -----
  -----
```

14.7 Grouping by multiple variables (nested grouping)

It is possible to group a data frame by more than one variable. This is sometimes called “nested” grouping.

Let’s see an example. Suppose you want to know the mean age of men and women *in each neighbourhood* (rather than the mean age of *all* women), you could put both `sex` and `neighborhood` in the `group_by()` statement:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age))
```

``summarise()`` has grouped output by `'sex'`. You can override using the `` .groups`` argument.

```
# A tibble: 18 x 3
# Groups:   sex [2]
   sex    neighborhood mean_age
<chr> <chr>          <dbl>
1 Female Briqueterie    31.6
2 Female Carriere      28.2
3 Female Cité Verte    31.8
4 Female Ekoudou       29.3
5 Female Messa         30.2
6 Female Mokolo        28.0
7 Female Nkomkana      33.0
8 Female Tsinga        30.6
9 Female Tsinga Oliga  24.3
10 Male  Briqueterie    33.7
11 Male  Carriere      30.0
12 Male  Cité Verte    27.0
13 Male  Ekoudou       25.2
14 Male  Messa         23.9
15 Male  Mokolo        30.5
16 Male  Nkomkana      29.8
17 Male  Tsinga        28.8
18 Male  Tsinga Oliga    24.3
```

From this output data frame you can tell that, for example, women from Briqueterie have a mean age of 31.6 years, while men from Briqueterie have a mean age of 33.7 years.

The order of the columns listed in `group_by()` is interchangeable. So if you run `group_by(neighborhood, sex)` instead of `group_by(sex, neighborhood)`, you'll get the same result, although it will be ordered differently:

```
yao %>%
  group_by(neighborhood, sex) %>%
  summarize(mean_age = mean(age))
```

``summarise()`` has grouped output by 'neighborhood'. You can override using the ``groups`` argument.

```
# A tibble: 18 x 3
# Groups:   neighborhood [9]
  neighborhood sex    mean_age
  <chr>         <chr>    <dbl>
1 Briqueterie  Female    31.6
2 Briqueterie  Male     33.7
3 Carriere     Female    28.2
4 Carriere     Male     30.0
5 Cité Verte   Female    31.8
6 Cité Verte   Male     27.0
7 Ekoudou      Female    29.3
8 Ekoudou      Male     25.2
9 Messa        Female    30.2
10 Messa       Male     23.9
11 Mokolo      Female    28.0
12 Mokolo      Male     30.5
13 Nkomkana    Female    33.0
14 Nkomkana    Male     29.8
15 Tsinga      Female    30.6
16 Tsinga      Male     28.8
17 Tsinga Olig Female    24.3
18 Tsinga Olig Male     24.3
```

Now the column order is different: `neighborhood` is the first column, and `sex` is the second. And the row order is also different: rows are first ordered by `neighborhood`, then ordered by `sex` within each neighborhood.

But the actual summary statistics are the same. For example, you can again see that women from Briqueterie have a mean age of 31.6 years, while men from Briqueterie have a mean age of 33.7 years.

Practice

Using the `yao` data frame, group your data by gender (`sex`) and treatments (`treatment_combinations`) using `group_by`. Then, using `summarize()` and the relevant summary function, calculate the mean weight (`weight_kg`) for each group. Your output should be a data frame with three columns named as shown below:

sex	treatment_combinations	mean_weight_kg
-----	------------------------	----------------

```
Q_weight_by_sex_treatments <-
  yao %>%
  -----
```

Using the yao data frame, group your data by age category (age_category_3), gender (sex), and IgG results (igg_result) using group_by(). Then, using summarize() and the relevant summary function, calculate the mean number of bedridden days (n_bedridden_days) for each group. Your output should be a data frame with four columns named as shown below:

age_category_3	sex	igg_result	mean_n_bedridden_days
----------------	-----	------------	-----------------------

```
Q_bedridden_by_age_sex_igresult <-
  yao %>%
  -----
```

14.8 Ungrouping with dplyr::ungroup() (why and how)

When you group_by() more than one variable before using summarize(), the output data frame is still grouped. This persistent grouping can have unwanted downstream effects, so you will sometimes need to use dplyr::ungroup() to ungroup the data before doing further analysis.

To understand *why* you should ungroup() data, first consider the following example, where we group by only one variable before summarizing:

```
yao %>%
  group_by(sex) %>%
  summarize(mean_age = mean(age))
```

```
# A tibble: 2 x 2
  sex    mean_age
<chr>    <dbl>
1 Female    29.5
2 Male     28.4
```

The data comes out like a normal data frame; it is not grouped. You can tell this because there is no information about groups in the header.

But now consider when you group by two variables before summarizing:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age))
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

```
# A tibble: 18 x 3
# Groups:   sex [2]
  sex    neighborhood mean_age
<chr>   <chr>         <dbl>
1 Female Briqueterie    31.6
2 Female Carriere      28.2
3 Female Cité Verte    31.8
4 Female Ekoudou       29.3
5 Female Messa         30.2
6 Female Mokolo        28.0
7 Female Nkomkana      33.0
8 Female Tsinga        30.6
9 Female Tsinga Oliga  24.3
10 Male  Briqueterie    33.7
11 Male  Carriere      30.0
12 Male  Cité Verte    27.0
13 Male  Ekoudou       25.2
14 Male  Messa         23.9
15 Male  Mokolo        30.5
16 Male  Nkomkana      29.8
17 Male  Tsinga        28.8
18 Male  Tsinga Oliga   24.3
```

Now the header tells you that the data is still grouped by the first variable in `group_by()`, `sex`:

```
# A tibble: 18 x 3
# Groups:   sex [2]
```

What is the implication of this persistent grouping in the data frame? It means that the data frame may exhibit what seems like weird behavior when you try to apply some {dplyr} functions on it.

For example, if you try to `select()` a single variable, perhaps the `mean_age` variable, you should normally be able to just use `select(mean_age)`:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  select(mean_age) # doesn't work as expected
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

Adding missing grouping variables: ``.sex``

```
# A tibble: 18 x 2
# Groups:   sex [2]
  sex    mean_age
<chr>    <dbl>
1 Female    31.6
```

2 Female	28.2
3 Female	31.8
4 Female	29.3
5 Female	30.2
6 Female	28.0
7 Female	33.0
8 Female	30.6
9 Female	24.3
10 Male	33.7
11 Male	30.0
12 Male	27.0
13 Male	25.2
14 Male	23.9
15 Male	30.5
16 Male	29.8
17 Male	28.8
18 Male	24.3

But as you can see, the grouped-by variable, `sex`, is *still* selected, even though we only asked for `mean_age` in the `select()` statement.

This is one of the many examples of unique behaviors of grouped data frames. Other dplyr verbs like `filter()`, `mutate()` and `arrange()` also act in special ways on grouped data. We will address this in detail in a future lesson.

So you now know *why* you should ungroup data when you no longer need it grouped. Let's now see *how* to ungroup data. It's quite simple: just add the `ungroup()` function to your pipe chain. For example:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  ungroup()
```

``summarise()`` has grouped output by `'sex'`. You can override using the ``.groups`` argument.

```
# A tibble: 18 x 3
   sex    neighborhood mean_age
<chr> <chr>          <dbl>
1 Female Briqueterie      31.6
2 Female Carriere       28.2
3 Female Cité Verte     31.8
4 Female Ekoudou        29.3
5 Female Messa          30.2
6 Female Mokolo         28.0
7 Female Nkomkana       33.0
8 Female Tsinga         30.6
9 Female Tsinga Oliga   24.3
10 Male  Briqueterie      33.7
```

11	Male	Carriere	30.0
12	Male	Cité Verte	27.0
13	Male	Ekoudou	25.2
14	Male	Messa	23.9
15	Male	Mokolo	30.5
16	Male	Nkomkana	29.8
17	Male	Tsinga	28.8
18	Male	Tsinga Oliga	24.3

Now that the data frame is ungrouped, it will behave like a normal data frame again. For example, you can `select()` any column(s) you want; you won't have some unwanted columns tagging along:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  ungroup() %>%
  select(mean_age)
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

```
# A tibble: 18 x 1
  mean_age
  <dbl>
1     31.6
2     28.2
3     31.8
4     29.3
5     30.2
6     28.0
7     33.0
8     30.6
9     24.3
10    33.7
11    30.0
12    27.0
13    25.2
14    23.9
15    30.5
16    29.8
17    28.8
18    24.3
```

14.9 Counting rows

You can do a lot of data science by just *counting* and occasionally *dividing*. - Hadley Wickham, Chief Scientist at RStudio

A common data summarization task is counting how many observations (rows) there are for each group. You can achieve this with the special `n()` function from `{dplyr}`, which is specifically designed to be used within `summarize()`.

For example, if you want to count how many individuals are in each neighborhood group, you would run:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(count = n())
```

```
# A tibble: 9 x 2
  neighborhood count
  <chr>         <int>
1 Briqueterie    106
2 Carriere       236
3 Cité Verte     72
4 Ekoudou        190
5 Messa          48
6 Mokolo         96
7 Nkomkana       75
8 Tsinga         81
9 Tsinga Oliga   67
```

As you can see, the `n()` function does not require any arguments. It just “knows its job” in the data frame!

Of course, you can include other summary statistics in the same `summarize()` call. For example, below we also calculate the mean age per neighborhood.

```
yao %>%
  group_by(neighborhood) %>%
  summarize(count = n(),
            mean_age = mean(age))
```

```
# A tibble: 9 x 3
  neighborhood count mean_age
  <chr>         <int>   <dbl>
1 Briqueterie    106    32.5
2 Carriere       236    28.9
3 Cité Verte     72    29.9
4 Ekoudou        190    27.6
5 Messa          48    27.3
6 Mokolo         96    29.1
7 Nkomkana       75    31.7
8 Tsinga         81    29.7
9 Tsinga Oliga   67    24.3
```

 Practice

Group your yao data frame by the respondents' occupation (occupation) and use `summarize()` to create columns that show:

- how many individuals there are with each occupation (think of the `n()` function)
- the mean number of work days missed (`n_days_miss_work`) by those in that occupation

Your output should be a data frame with three columns named as shown below:

occupation	count	mean_n_days_miss_work
------------	-------	-----------------------

```
Q_occupation_summary <-
  yao %>%
  -----
```

14.9.1 Counting rows that meet a condition

Rather than counting *all* rows as above, it is sometimes more useful to count just the rows that meet specific conditions. This can be done easily by placing the required conditions within the `sum()` function.

For example, to count the number of people under 18 in each neighborhood, you place the condition `age < 18` inside `sum()`:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(count_under_18 = sum(age < 18))
```

```
# A tibble: 9 x 2
  neighborhood count_under_18
  <chr>          <int>
1 Briqueterie    28
2 Carriere       58
3 Cité Verte     19
4 Ekoudou        66
5 Messa          18
6 Mokolo         32
7 Nkomkana       22
8 Tsinga         23
9 Tsinga Oliga   25
```

Similarly, to count the number of people with doctorate degrees in each neighborhood, you place the condition `highest_education == "Doctorate"` inside `sum()`:

```
yao %>%
  group_by(neighborhood) %>%
```

```
summarize(count_with_doctorates = sum(highest_education == "Doctorate"))
```

```
# A tibble: 9 x 2
  neighborhood count_with_doctorates
  <chr>          <int>
1 Briqueterie      2
2 Carriere         1
3 Cité Verte       1
4 Ekoudou          1
5 Messa           2
6 Mokolo           0
7 Nkomkana         4
8 Tsinga           3
9 Tsinga Oliga     3
```

Challenge

Under the hood: counting with conditions

Why are you able to use `sum()` which is meant to add numbers, on a condition like `highest_education == "Doctorate"`?

Using `sum()` on a condition works because the condition evaluates to the Boolean values `TRUE` and `FALSE`. And these Boolean values are treated as numbers (where `TRUE` equals 1 and `FALSE` equals 0), and numbers can, of course, be summed.

The code below demonstrates what is going on under the hood in a step-by-step way. Run through it and see if you can follow.

```
demo_of_condition_sums <- yao %>%
  select(highest_education) %>%
  mutate(with_doctorate = highest_education == "Doctorate") %>%
  mutate(numeric_with_doctorate = as.numeric(with_doctorate))

demo_of_condition_sums
```

```
# A tibble: 971 x 3
  highest_education with_doctorate numeric_with_doctorate
  <chr>             <lgl>                <dbl>
1 Secondary        FALSE                0
2 University       FALSE                0
3 University       FALSE                0
4 Secondary        FALSE                0
5 Primary          FALSE                0
6 Secondary        FALSE                0
7 Secondary        FALSE                0
8 Doctorate        TRUE                 1
9 Secondary        FALSE                0
10 Secondary       FALSE                0
# i 961 more rows
```

The numeric values can then be added to produce a count of rows fulfilling the condition `highest_education == "Doctorate"`:

```
demo_of_condition_sums %>%
  summarize(count_with_doctorate = sum(numeric_with_doctorate))
```

```
# A tibble: 1 x 1
  count_with_doctorate
          <dbl>
1                17
```

For a final illustration of counting with conditions, consider the `treatment_combinations` variable, which lists the treatments received by people with COVID-like symptoms. People who received no treatments have an NA value:

```
yao %>%
  select(treatment_combinations)
```

```
# A tibble: 971 x 1
  treatment_combinations
          <chr>
1 Paracetamol
2 <NA>
3 <NA>
4 Antibiotics
5 <NA>
6 Paracetamol--Antibiotics
7 Traditional meds.
8 Paracetamol
9 Paracetamol--Traditional meds.
10 <NA>
# i 961 more rows
```

If you want to count the number of people who received *no treatment*, you would sum up those who meet the `is.na(treatment_combinations)` condition:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(unknown_treatments = sum(is.na(treatment_combinations)))
```

```
# A tibble: 9 x 2
  neighborhood unknown_treatments
          <chr>          <int>
1 Briqueterie           82
2 Carriere             192
3 Cité Verte            46
4 Ekoudou             133
5 Messa                35
6 Mokolo               65
```

7 Nkomkana	53
8 Tsinga	56
9 Tsinga Oliga	47

These are the people with NA values for the `treatment_combinations` column.

To count the people who *did* receive some treatment, you can simply negate the `is.na()` function with `!`:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(known_treatments = sum(!is.na(treatment_combinations)))
```

```
# A tibble: 9 x 2
  neighborhood known_treatments
  <chr>          <int>
1 Briqueterie    24
2 Carriere       44
3 Cité Verte     26
4 Ekoudou        57
5 Messa          13
6 Mokolo         31
7 Nkomkana       22
8 Tsinga         25
9 Tsinga Oliga   20
```

PLEASE SKIP THE PRACTICE QUESTION ON CHECKING SYMPTOMS FOR ADULTS. WE DECIDED TO REMOVE IT.

14.9.2 dplyr::count()

The `dplyr::count()` function wraps a bunch of things into one beautiful friendly line of code to help you find counts of observations by group.

Let's use `dplyr::count()` on our occupation variable:

```
yao %>%
  count(occupation)
```

```
# A tibble: 28 x 2
  occupation          n
  <chr>              <int>
1 Farmer            5
2 Farmer--Other      1
3 Home-maker        65
4 Home-maker--Farmer  2
5 Home-maker--Informal worker  3
6 Home-maker--Informal worker--Farmer  1
7 Home-maker--Trader  3
8 Informal worker    189
9 Informal worker--Other  2
10 Informal worker--Trader  4
# i 18 more rows
```

Note that this is the same output as:

```
yao %>%
  group_by(occupation) %>%
  summarize(n = n())
```

```
# A tibble: 28 x 2
  occupation      n
  <chr>      <int>
1 Farmer      5
2 Farmer--Other 1
3 Home-maker  65
4 Home-maker--Farmer 2
5 Home-maker--Informal worker 3
6 Home-maker--Informal worker--Farmer 1
7 Home-maker--Trader 3
8 Informal worker 189
9 Informal worker--Other 2
10 Informal worker--Trader 4
# i 18 more rows
```

You can also apply `dplyr::count()` in a nested fashion:

```
yao %>%
  count(sex, occupation)
```

```
# A tibble: 40 x 3
  sex      occupation      n
  <chr>   <chr>      <int>
1 Female Farmer      3
2 Female Home-maker  65
3 Female Home-maker--Farmer 2
4 Female Home-maker--Informal worker 3
5 Female Home-maker--Informal worker--Farmer 1
6 Female Home-maker--Trader 3
7 Female Informal worker 77
8 Female Informal worker--Trader 1
9 Female No response  8
10 Female Other      6
# i 30 more rows
```

Practice

The `count()` verb gives you key information about your dataset in a very quick manner. Let's look at our IgG results stratified by age category and sex in one line of code.

Using the `yao` data frame, count the different combinations of gender (`sex`), age categories (`age_category_3`) and IgG results (`igg_result`).

Your output should be a data frame with four columns named as shown below:

sex	age_category_3	igg_result	n
-----	----------------	------------	---

```
Q_count_iggresults_stratified_by_sex_agecategories <-
yao %>%
```

```
-----
```

Using the `yao` data frame, count the different combinations of age categories (`age_category_3`) and number of bedridden days (`n_bedridden_days`).

Your output should be a data frame with three columns named as shown below:

age_category_3	n_bedridden_days	n
----------------	------------------	---

```
Q_count_bedridden_age_categories <-
yao %>%
```

```
-----
```

The downside of `count()` is that it can only give you a single summary statistic in the data frame. When you use `summarize()` and `n()` you can include multiple summary statistics. For example:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(count = n(),
            median_age = median(age))
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

```
# A tibble: 18 x 4
# Groups:   sex [2]
  sex    neighborhood count median_age
<chr> <chr>         <int>    <dbl>
1 Female Briqueterie     61      28
2 Female Carriere      140     25.5
3 Female Cité Verte     44      28
4 Female Ekoudou      110     26.5
5 Female Messa         26     27.5
6 Female Mokolo        53      23
7 Female Nkomkana      43      28
8 Female Tsinga        42      29
9 Female Tsinga Oliga   30     23.5
10 Male  Briqueterie     45      28
11 Male  Carriere       96      27
12 Male  Cité Verte     28     22.5
```

13	Male	Ekoudou	80	21.5
14	Male	Messa	22	24.5
15	Male	Mokolo	43	32
16	Male	Nkomkana	32	27
17	Male	Tsinga	39	27
18	Male	Tsinga Oliga	37	21

But `count()` can only yield counts:

```
yao %>%
  group_by(sex, neighborhood) %>%
  count()
```

```
# A tibble: 18 x 3
# Groups:   sex, neighborhood [18]
   sex    neighborhood     n
  <chr>   <chr>         <int>
1 Female Briqueterie      61
2 Female Carriere       140
3 Female Cité Verte      44
4 Female Ekoudou       110
5 Female Messa          26
6 Female Mokolo         53
7 Female Nkomkana       43
8 Female Tsinga         42
9 Female Tsinga Oliga    30
10 Male  Briqueterie      45
11 Male  Carriere        96
12 Male  Cité Verte       28
13 Male  Ekoudou          80
14 Male  Messa             22
15 Male  Mokolo            43
16 Male  Nkomkana           32
17 Male  Tsinga             39
18 Male  Tsinga Oliga        37
```

14.10 Including missing combinations in summaries

When you use `group_by()` and `summarize()` on multiple variables, you obtain a summary statistic for every unique combination of the grouped variables. For instance, consider the code and output below, which counts the number of individuals in each age-sex group:

```
yao %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n())
```

``summarise()`` has grouped output by `'sex'`. You can override using the ``.groups`` argument.

```
# A tibble: 6 x 3
# Groups:   sex [2]
  sex    age_category_3 number_of_individuals
<chr>  <chr>                <int>
1 Female Adult              368
2 Female Child              155
3 Female Senior              26
4 Male   Adult              267
5 Male   Child              136
6 Male   Senior              19
```

In the output data frame, there is one row for each combination of sex and age group (Female—Adult, Female—Child and so on).

But what happens if one of these combinations is not present in the data?

Let's create an artificial example to observe this. With the code below, we artificially drop all male children from the yao data frame:

```
yao_no_male_children <-
  yao %>%
  filter(!(sex == "Male" & age_category_3 == "Child"))
```

Now if you run the same `group_by()` and `summarize()` call on `yao_no_male_children`, you'll notice the missing combination:

```
yao_no_male_children %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n())
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

```
# A tibble: 5 x 3
# Groups:   sex [2]
  sex    age_category_3 number_of_individuals
<chr>  <chr>                <int>
1 Female Adult              368
2 Female Child              155
3 Female Senior              26
4 Male   Adult              267
5 Male   Senior              19
```

Indeed, there is no row for male children.

But sometimes it is useful to include such missing combinations in the output data frame, with an NA or 0 value for the summary statistic.

To do this, you can run the following code instead:

```
yao_no_male_children %>%
  # convert variables to factors
  mutate(sex = as.factor(sex),
```

```

    age_category_3 = as.factor(age_category_3)) %>%
# Note the the .drop = FALSE argument
group_by(sex, age_category_3, .drop = FALSE) %>%
summarise(number_of_individuals = n())

```

``summarise()`` has grouped output by 'sex'. You can override using the `` .groups`` argument.

```

# A tibble: 6 x 3
# Groups:   sex [2]
  sex    age_category_3 number_of_individuals
<fct> <fct>                <int>
1 Female Adult                368
2 Female Child                155
3 Female Senior                26
4 Male    Adult                267
5 Male    Child                 0
6 Male    Senior               19

```

What does the code do?

- First it converts the grouping variables to factors with `as.factor()` (inside a `mutate()` call)
- Then it uses the argument `.drop = FALSE` in the `group_by()` function to avoid dropping the missing combinations.

Now you have a clear 0 count for the number of male children!

Let's see one more example, this time without artificially modifying our data.

The code below calculates the average age by sex and education group:

```

yao %>%
  group_by(sex, highest_education) %>%
  summarise(mean_age = mean(age))

```

``summarise()`` has grouped output by 'sex'. You can override using the `` .groups`` argument.

```

# A tibble: 13 x 3
# Groups:   sex [2]
  sex    highest_education    mean_age
<chr> <chr>                <dbl>
1 Female Doctorate          28
2 Female No formal instruction 45.6
3 Female No response         35
4 Female Primary            26.8
5 Female Secondary          28.8
6 Female University         31.5

```

7 Male	Doctorate	42.2
8 Male	No formal instruction	37.9
9 Male	No response	22
10 Male	Other	5.5
11 Male	Primary	22.9
12 Male	Secondary	29.4
13 Male	University	31.9

Notice that in the output data frame, there are 7 rows for men but only 6 rows for women, because no woman answered “Other” to the question on highest education level.

If you nonetheless want to include the “Female—Other” row in the output data frame, you would run:

```
yao %>%
  mutate(sex = as.factor(sex),
         highest_education = as.factor(highest_education)) %>%
  group_by(sex, highest_education, .drop = FALSE) %>%
  summarise(mean_age = mean(age))
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

```
# A tibble: 14 x 3
# Groups:   sex [2]
  sex    highest_education    mean_age
  <fct>   <fct>              <dbl>
1 Female Doctorate          28
2 Female No formal instruction 45.6
3 Female No response         35
4 Female Other              NaN
5 Female Primary            26.8
6 Female Secondary          28.8
7 Female University         31.5
8 Male   Doctorate          42.2
9 Male   No formal instruction 37.9
10 Male  No response         22
11 Male  Other              5.5
12 Male  Primary            22.9
13 Male  Secondary          29.4
14 Male  University         31.9
```

Practice

Using the `yao` data frame, let’s calculate the median age when grouping by neighborhood, age_category, and gender

Note, we want all possible combinations of these three variables (not just those present in our data). Pay attention to two data wrangling imperatives!

- convert your grouping variables to factors beforehand using `mutate()`
- calculate your statistic, the median, while removing any NA values.

Your output should be a data frame with four columns named as shown below:

neighborhood	age_category_3	sex	median_age
--------------	----------------	-----	------------

```
Q_median_age_by_neighborhood_agecategory_sex <-
  yao %>%
  -----
```

i Side Note

Why include missing combinations?

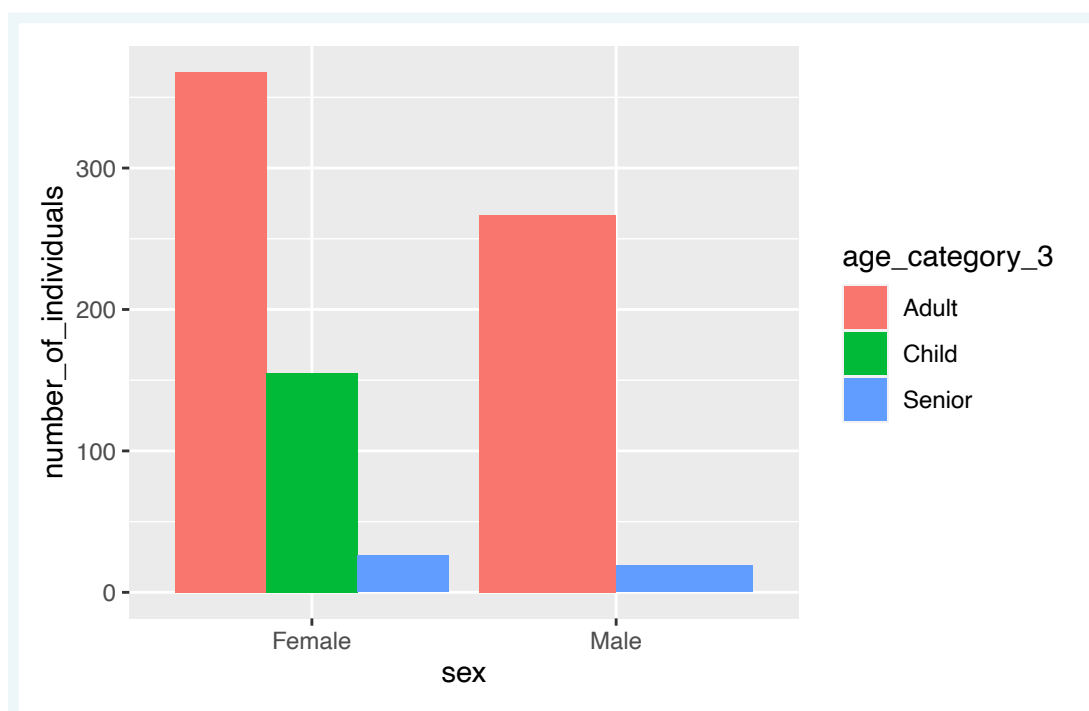
Above, we mentioned that including missing combinations is often useful in the data analysis workflow. Let's see one use case: plotting with {ggplot}. If you have not yet learned {ggplot}, that is okay, just focus on the plot outputs.

To make a dodged bar chart with the age-sex counts of `yao_no_male_children`, you could run:

```
yao_no_male_children %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

# pass the output to ggplot
ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill = age_category_3),
           position = "dodge")
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

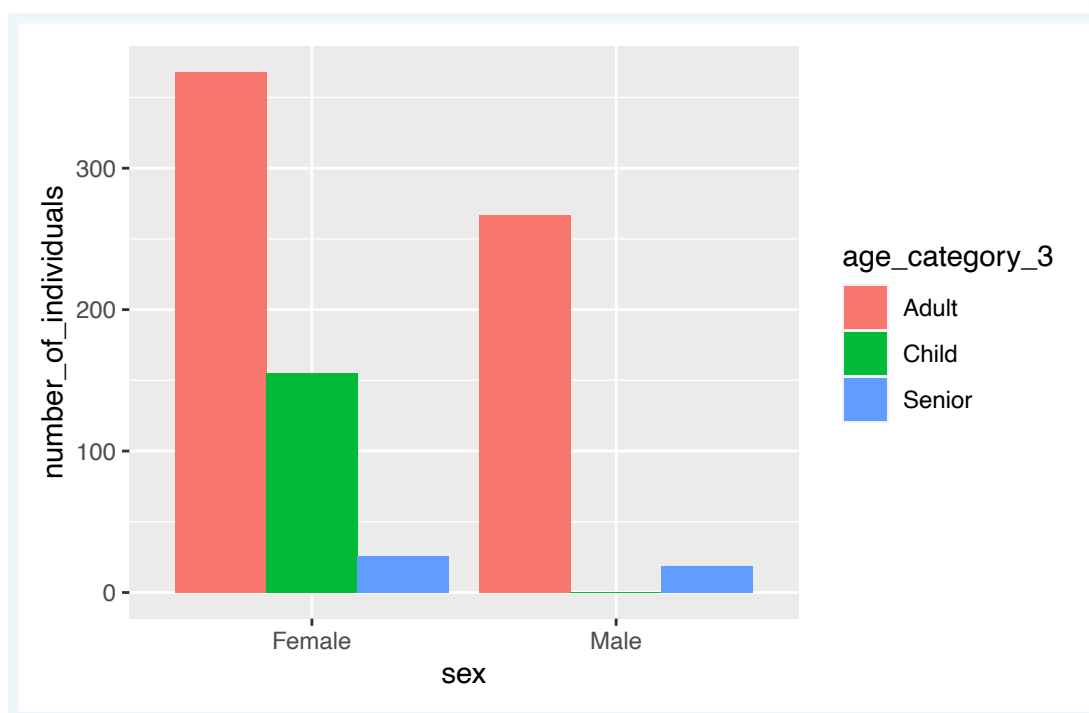


Not very elegant! Ideally there should be an empty space indicating 0 for the number of male children. If you instead implement the procedure to include missing combinations, you get a more natural dodged bar plot, with an empty space for male children:

```
yao_no_male_children %>%
  mutate(sex = as.factor(sex),
         age_category_3 = as.factor(age_category_3)) %>%
  group_by(sex, age_category_3, .drop = FALSE) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

# pass the output to ggplot
ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill = age_category_3),
          position = "dodge")
```

`summarise()` has grouped output by 'sex'. You can override using the `.groups` argument.



Much better!

By the way, this output can be improved slightly by setting the factor levels for age to their proper ascending order: first "Child", then "Adult" then "Senior":

```

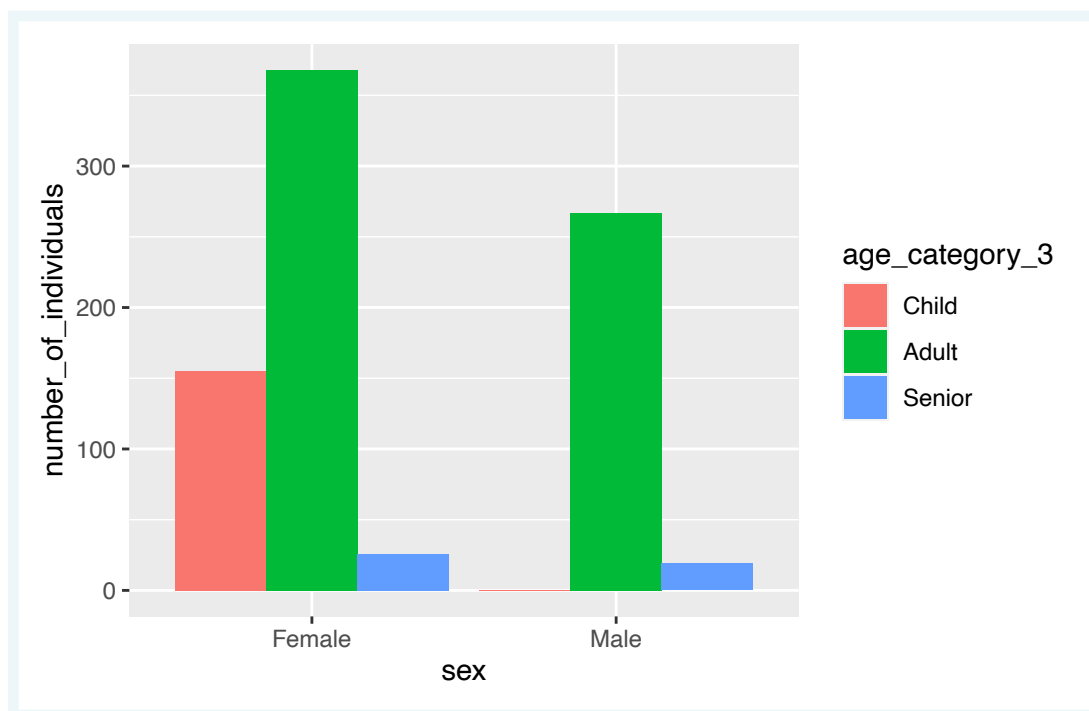
yao_no_male_children %>%
  mutate(sex = as.factor(sex),
         age_category_3 = factor(age_category_3,
                                levels = c("Child",
                                             "Adult",
                                             "Senior"))) %>%

  group_by(sex, age_category_3, .drop = FALSE) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

  # pass the output to ggplot
  ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill = age_category_3),
           position = "dodge")

```

`summarise()` has grouped output by 'sex'. You can override using the `.groups` argument.



14.11 Wrap up

You have now seen how to obtain quick summary statistics from your data, either for exploratory data or for further data presentation or plotting.

Additionally, you have discovered one of the marvels of {dplyr}, the possibility to group your data using `group_by()`.

`group_by()` combined with `summarize()` is one of the most common grouping manipulations.

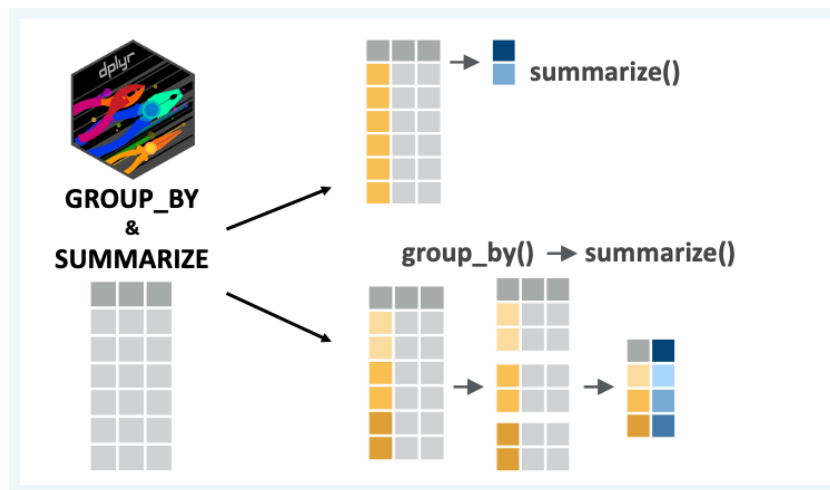


Figure 14.2: Fig: `summarize()` and its use combined with `group_by()`.

However, you can also combine `group_by()` with many of the other {dplyr} verbs: this is what we will cover in our next lesson. See you soon !

Thank you to [Alice Osmaston](#) and [Saifeldin Shehata](#) for their comments and review.

References

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

- Horst, A. (2022). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)
- *Group by one or more variables*. (n.d.). Retrieved 21 February 2022, from https://dplyr.tidyverse.org/reference/group_by.html
- *Summarise each group to fewer rows*. (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/summarize.html>
- The Carpentries. (n.d.). *Grouped operations using 'dplyr'*. Grouped operations using 'dplyr' – Introduction to R/tidyverse for Exploratory Data Analysis. Retrieved July 28, 2022, from https://tavareshugo.github.io/r-intro-tidyverse-gapminder/06-grouped_operations_dplyr/index.html

Artwork was adapted from:

- Horst, A. (2022). *R & stats illustrations by Allison Horst*. <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

14.12 Solutions

```
.SOLUTION_Q_weight_summary()
```

```
Q_weight_summary <-  
  yao %>%  
    summarize(mean_weight_kg = mean(weight_kg),  
              median_weight_kg = median(weight_kg),  
              sd_weight_kg = sd(weight_kg))
```

```
.SOLUTION_Q_height_summary()
```

```
Q_height_summary <-  
  yao %>%  
    summarize(min_height_cm = min(height_cm),  
              max_height_cm = max(height_cm))
```

```
.SOLUTION_Q_weight_by_smoking_status()
```

```
Q_weight_by_smoking_status <-  
  yao %>%  
    group_by(is_smoker) %>%  
    summarise(weight_mean = mean(weight_kg))
```

```
.SOLUTION_Q_min_max_height_by_sex()
```

```
Q_min_max_height_by_sex <-  
  yao %>%  
    group_by(sex) %>%  
    summarise(min_height_cm = min(height_cm),  
              max_height_cm = max(height_cm))
```

```
.SOLUTION_Q_sum_bedridden_days()
```

```
Q_sum_bedridden_days <-  
  yao %>%  
    group_by(sex) %>%  
    summarise(total_bedridden_days = sum(n_bedridden_days, na.rm = T))
```

```
.SOLUTION_Q_weight_by_sex_treatments()
```

```
Q_weight_by_sex_treatments <-
  yao %>%
  group_by(sex, treatment_combinations) %>%
  summarise(mean_weight_kg = mean(weight_kg, na.rm = T))
```

```
.SOLUTION_Q_bedridden_by_age_sex_iggresult()
```

```
Q_bedridden_by_age_sex_iggresult <-
  yao %>%
  group_by(age_category_3, sex, igg_result) %>%
  summarise(mean_n_bedridden_days = mean(n_bedridden_days, na.rm = T))
```

```
.SOLUTION_Q_occupation_summary()
```

```
Q_occupation_summary <-
  yao %>%
  group_by(occupation) %>%
  summarise(count = n(),
            mean_n_days_miss_work = mean(n_days_miss_work, na.rm=TRUE))
```

```
.SOLUTION_Q_count_iggresults_stratified_by_sex_agecategories()
```

```
Q_count_iggresults_stratified_by_sex_agecategories <-
  yao %>%
  count(sex, age_category_3, igg_result)
```

```
.SOLUTION_Q_count_bedridden_age_categories()
```

```
Q_count_bedridden_age_categories <-
  yao %>%
  count(age_category_3, n_bedridden_days)
```

```
.SOLUTION_Q_median_age_by_neighborhood_agecategory_sex()
```

```
Q_median_age_by_neighborhood_agecategory_sex <-  
  yao %>%  
  mutate(neighborhood = as.factor(neighborhood),  
         age_category_3 = as.factor(age_category_3),  
         sex = as.factor(sex)) %>%  
  group_by(neighborhood, age_category_3, sex, .drop=FALSE) %>%  
  summarize(median_age = median(age, na.rm=TRUE))
```

Chapter 15

Grouped filter, mutate and arrange

15.1 Introduction

Data wrangling often involves applying the same operations separately to different groups within the data. This pattern, sometimes called “split-apply-combine”, is easily accomplished in {dplyr} by chaining the `group_by()` verb with other wrangling verbs like `filter()`, `mutate()`, and `arrange()` (all of which you have seen before!).

In this lesson, you’ll become confident with these kinds of grouped manipulations.

Let’s get started.

15.2 Learning objectives

1. You can use `group_by()` with `arrange()`, `filter()`, and `mutate()` to conduct grouped operations on a data frame.

15.3 Packages

This lesson will require the {tidyverse} suite of packages and the {here} package:

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

15.4 Datasets

In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon. Below, we import the data, create a small data frame subset, `yao` and an even smaller subset, `yao_sex_weight`.

```
yao <-
  read_csv(here::here('data/yaounde_data.csv')) %>%
  select(sex, age, age_category, weight_kg, occupation, igg_result, igm_result)
```

```
yao
```

```
# A tibble: 5 x 7
  sex      age age_category weight_kg occupation      igg_result igm_result
<chr> <dbl> <chr>          <dbl> <chr>          <chr>      <chr>
1 Female    45 45 - 64             95 Informal worker Negative Negative
2 Male     55 45 - 64             96 Salaried worker Positive Negative
3 Male     23 15 - 29             74 Student          Negative Negative
4 Female    20 15 - 29             70 Student          Positive Negative
5 Female    55 45 - 64             67 Trader--Farmer Positive Negative
```

```
yao_sex_weight <-
  yao %>%
  select(sex, weight_kg)

yao_sex_weight
```

```
# A tibble: 5 x 2
  sex      weight_kg
<chr>      <dbl>
1 Female         95
2 Male          96
3 Male          74
4 Female         70
5 Female         67
```

For practice questions, we will also use the sarcopenia data set that you have seen previously:

```
sarcopenia <- read_csv(here::here('data/sarcopenia_elderly.csv'))

sarcopenia
```

```
# A tibble: 5 x 9
  number  age age_group sex_male_1_female_0 marital_status height_meters
  <dbl> <dbl> <chr>          <dbl> <chr>          <dbl>
1      7  60.8 Sixties             0 married         1.57
2      8  72.3 Seventies          1 married         1.65
3      9  62.6 Sixties             0 married         1.59
4     12  72   Seventies             0 widow          1.47
5     13  60.1 Sixties             0 married         1.55
# i 3 more variables: weight_kg <dbl>, grip_strength_kg <dbl>,
#   skeletal_muscle_index <dbl>
```

15.5 Arranging by group

The `arrange()` function orders the rows of a data frame by the values of selected columns. This function is only sensitive to groupings when we set its argument `.by_group` to `TRUE`. To illustrate this, consider the `yao_sex_weight` data frame:

```
yao_sex_weight

# A tibble: 5 x 2
  sex    weight_kg
<chr>    <dbl>
1 Female      95
2 Male        96
3 Male        74
4 Female      70
5 Female      67
```

We can arrange this data frame by weight like so:

```
yao_sex_weight %>%
  arrange(weight_kg)

# A tibble: 5 x 2
  sex    weight_kg
<chr>    <dbl>
1 Female      14
2 Male        15
3 Male        15
4 Male        15
5 Female      15
```

As expected, lower weights have been brought to the top of the data frame.

If we first group the data, we might expect a different output:

```
yao_sex_weight %>%
  group_by(sex) %>%
  arrange(weight_kg)

# A tibble: 5 x 2
# Groups:   sex [2]
  sex    weight_kg
<chr>    <dbl>
1 Female      14
2 Male        15
3 Male        15
4 Male        15
5 Female      15
```

But as you see, the arrangement is still the same.

Only when we set the `.by_group` argument to `TRUE` do we get something different:

```
yao_sex_weight %>%
  group_by(sex) %>%
  arrange(weight_kg, .by_group = TRUE)
```

```
# A tibble: 5 x 2
# Groups:   sex [1]
  sex    weight_kg
<chr>    <dbl>
1 Female      14
2 Female      15
3 Female      16
4 Female      16
5 Female      18
```

Now, the data is *first* sorted by sex (all women first), and then by weight.

`arrange()` can group automatically

In reality we do not need `group_by()` to arrange by group; we can simply put multiple variables in the `arrange()` function for the same effect.

So this simple `arrange()` statement:

```
yao_sex_weight %>%
  arrange(sex, weight_kg)
```

```
# A tibble: 5 x 2
  sex    weight_kg
<chr>    <dbl>
1 Female      14
2 Female      15
3 Female      16
4 Female      16
5 Female      18
```

is equivalent to the more complex `group_by()`, `arrange()` statement used before:

```
yao_sex_weight %>%
  group_by(sex) %>%
  arrange(weight_kg, .by_group = TRUE)
```

The code `arrange(sex, weight_kg)` tells R to arrange the rows *first* by sex, and then by weight.

Obviously, this syntax, with just `arrange()`, and no `group_by()` is simpler, so you can stick to it.

`desc()` for descending order

Recall that to arrange *in descending order*, we can wrap the target variable in `desc()`. So, for example, to sort by sex and weight, but with the heaviest people on top, we can run:

```
yao_sex_weight %>%
  arrange(sex, desc(weight_kg))
```

```
# A tibble: 5 x 2
  sex    weight_kg
<chr>    <dbl>
1 Female    162
2 Female    161
3 Female    158
4 Female    135
5 Female    129
```

Practice

With an `arrange()` call, sort the `sarcopenia` data first by sex and then by grip strength. (If done correctly, the first row should be of a woman with a grip strength of 1.3 kg). To make the arrangement clear, you should first `select()` the sex and grip strength variables.

```
## Complete the code with your answer:
Q_grip_strength_arranged <-
  sarcopenia %>%
  select(_____) %>%
  arrange(_____)
```

Practice

The `sarcopenia` dataset contains a column, `age_group`, which stores age groups as a string (the age groups are “Sixties”, “Seventies” and “Eighties”). Convert this variable to a factor with the levels in the right order (first “Sixties” then “Seventies” and so on). (Hint: Look back on the `case_when()` lesson if you do not see how to relevel a factor.)

Then, with a nested `arrange()` call, arrange the data first by the newly-created `age_group` factor variable (younger individuals first) and then by `height_meters`, with shorter individuals first.

```
## Complete the code with your answer:
Q_age_group_height <-
  sarcopenia
```

15.6 Filtering by group

The `filter()` function keeps or drops rows based on a condition. If `filter()` is applied to grouped data, the filtering operation is carried out separately for each group.

To illustrate this, consider again the `yao_sex_weight` data frame:

```
yao_sex_weight
```

```
# A tibble: 5 x 2
  sex    weight_kg
```

	<chr>	<dbl>
1	Female	95
2	Male	96
3	Male	74
4	Female	70
5	Female	67

If we want to filter the data for the heaviest person, we could run:

```
yao_sex_weight %>%
  filter(weight_kg == max(weight_kg))
```

```
# A tibble: 1 x 2
  sex    weight_kg
<chr>    <dbl>
1 Female      162
```

But if we want to get heaviest person per sex group (the heaviest man *and* the heaviest woman), we can use `group_by(sex)` then `filter()`:

```
yao_sex_weight %>%
  group_by(sex) %>%
  filter(weight_kg == max(weight_kg))
```

```
# A tibble: 2 x 2
# Groups:   sex [2]
  sex    weight_kg
<chr>    <dbl>
1 Male      128
2 Female    162
```

Great! The code above can be translated as “For each sex group, keep the row with the maximum `weight_kg` value”.

Filtering with nested groupings

`filter()` will work fine with any number of nested groupings.

For example, if we want to see the heaviest man and heaviest woman *per age group* we could run the following on the `yao` data frame:

```
yao %>%
  group_by(sex, age_category) %>%
  filter(weight_kg == max(weight_kg))
```

```
# A tibble: 10 x 7
# Groups:   sex, age_category [10]
  sex    age age_category weight_kg occupation    igg_result igm_result
<chr> <dbl> <chr>          <dbl> <chr>          <chr>      <chr>
```

1 Male	69 65 +	108 Retired	Positive	Negative
2 Male	37 30 - 44	128 Informal worker	Negative	Negative
3 Male	26 15 - 29	91 Trader	Positive	Negative
4 Female	19 15 - 29	109 Student	Negative	Negative
5 Female	64 45 - 64	158 Retired	Negative	Negative
6 Female	32 30 - 44	162 Informal worker	Positive	Negative
7 Male	46 45 - 64	122 Informal worker	Negative	Negative
8 Female	8 5 - 14	161 Student	Negative	Positive
9 Female	68 65 +	109 Retired	Negative	Negative
10 Male	6 5 - 14	99 No response	Negative	Negative

This code groups by sex *and* age category, and then finds the heaviest person in each sub-category.

(Why do we have 10 rows in the output? Well, 2 sex groups x 5 groups age groups = 10 unique groupings.)

The output is a bit scattered though, so we can chain this with the `arrange()` function, to arrange by sex and age group.

```
yao %>%
  group_by(sex, age_category) %>%
  filter(weight_kg == max(weight_kg)) %>%
  arrange(sex, age_category)
```

A tibble: 10 x 7

Groups: sex, age_category [10]

	sex	age	age_category	weight_kg	occupation	igg_result	igm_result
	<chr>	<dbl>	<chr>	<dbl>	<chr>	<chr>	<chr>
1	Female	19	15 - 29	109	Student	Negative	Negative
2	Female	32	30 - 44	162	Informal worker	Positive	Negative
3	Female	64	45 - 64	158	Retired	Negative	Negative
4	Female	8	5 - 14	161	Student	Negative	Positive
5	Female	68	65 +	109	Retired	Negative	Negative
6	Male	26	15 - 29	91	Trader	Positive	Negative
7	Male	37	30 - 44	128	Informal worker	Negative	Negative
8	Male	46	45 - 64	122	Informal worker	Negative	Negative
9	Male	6	5 - 14	99	No response	Negative	Negative
10	Male	69	65 +	108	Retired	Positive	Negative

Now the data is easier to read. All women come first, then men. But we see notice a weird arrangement of the age groups! Those aged 5 to 14 should come *first* in the arrangement. Of course, we've learned how to fix this—the `factor()` function, and its `levels` argument:

```
yao %>%
  mutate(age_category = factor(
    age_category,
    levels = c("5 - 14", "15 - 29", "30 - 44", "45 - 64", "65 +")
  )) %>%
  group_by(sex, age_category) %>%
  filter(weight_kg == max(weight_kg)) %>%
  arrange(sex, age_category)
```

```
# A tibble: 10 x 7
# Groups:   sex, age_category [10]
  sex      age age_category weight_kg occupation      igg_result igm_result
  <chr> <dbl> <fct>          <dbl> <chr>          <chr>      <chr>
1 Female      8 5 - 14           161 Student      Negative Positive
2 Female     19 15 - 29           109 Student      Negative Negative
3 Female     32 30 - 44           162 Informal worker Positive Negative
4 Female     64 45 - 64           158 Retired       Negative Negative
5 Female     68 65 +             109 Retired       Negative Negative
6 Male        6 5 - 14            99 No response    Negative Negative
7 Male       26 15 - 29            91 Trader        Positive Negative
8 Male       37 30 - 44           128 Informal worker Negative Negative
9 Male       46 45 - 64           122 Informal worker Negative Negative
10 Male      69 65 +             108 Retired       Positive Negative
```

Now we have a nice and well-arranged output!

Practice

Group the sarcopenia data frame by age group and sex, then filter for the highest skeletal muscle index in each (nested) group.

```
## Complete the code with your answer:
Q_max_skeletal_muscle_index <-
  sarcopenia
```

15.7 Mutating by group

`mutate()` is used to modify columns or to create new ones. With grouped data, `mutate()` operates over each group independently.

Let's first consider a regular `mutate()` call, not a grouped one. Imagine that you wanted to add a column that ranks respondents by weight. This can be done with the `rank()` function inside a `mutate()` call:

```
yao_sex_weight %>%
  mutate(weight_rank = rank(weight_kg))
```

```
# A tibble: 5 x 3
  sex      weight_kg weight_rank
  <chr>      <dbl>      <dbl>
1 Female      95        901
2 Male       96        908
3 Male       74        640.
4 Female      70        564.
5 Female      67        502.
```

The output shows that the first row is the 901st lightest individual. But it would be more intuitive to rank in descending order with the heaviest person first. We can do this with the `desc()` function:

```
yao_sex_weight %>%
  mutate(weight_rank = rank(desc(weight_kg)))
```

```
# A tibble: 5 x 3
  sex    weight_kg weight_rank
<chr>    <dbl>    <dbl>
1 Female      95         71
2 Male       96         64
3 Male       74        332.
4 Female      70        408.
5 Female      67        470.
```

The output shows that the person in the first row is the 71st heaviest individual.

Now, let's try to write a grouped `mutate()` call. Imagine we want to add this weight rank column *per sex group* in the data frame. That is, we want to know each person's weight rank in their sex category. In this case, we can chain `group_by(sex)` with `mutate()`:

```
yao_sex_weight %>%
  group_by(sex) %>%
  mutate(weight_rank = rank(desc(weight_kg)))
```

```
# A tibble: 5 x 3
# Groups:   sex [2]
  sex    weight_kg weight_rank
<chr>    <dbl>    <dbl>
1 Female      95        53.5
2 Male       96        13.5
3 Male       74        148
4 Female      70        220.
5 Female      67        250.
```

Now we see that the person in the first row is the 53rd heaviest *woman*. (The .5 indicates that this rank is a tie with someone else in the data.)

We could also arrange the data to make things clearer:

```
yao_sex_weight %>%
  group_by(sex) %>%
  mutate(weight_rank = rank(desc(weight_kg))) %>%
  arrange(sex, weight_rank)
```

```
# A tibble: 5 x 3
# Groups:   sex [1]
  sex    weight_kg weight_rank
<chr>    <dbl>    <dbl>
1 Female      162         1
```

2 Female	161	2
3 Female	158	3
4 Female	135	4
5 Female	129	5

Mutating with nested groupings

Of course, as with the other verbs we have seen, `mutate()` also works with nested groups.

For example, below we create the nested grouping of age *and* sex with the yao data frame, then add a rank column with `mutate()`:

```
yao %>%
  group_by(sex, age_category) %>%
  mutate(weight_rank = rank(desc(weight_kg)))
```

```
# A tibble: 5 x 8
# Groups:   sex, age_category [4]
  sex      age age_category weight_kg occupation    igg_result igm_result
<chr> <dbl> <chr>          <dbl> <chr>        <chr>      <chr>
1 Female   45 45 - 64           95 Informal worker Negative Negative
2 Male     55 45 - 64           96 Salaried worker Positive Negative
3 Male     23 15 - 29           74 Student          Negative Negative
4 Female   20 15 - 29           70 Student          Positive Negative
5 Female   55 45 - 64           67 Trader--Farmer Positive Negative
# i 1 more variable: weight_rank <dbl>
```

The output shows that the person in the first row is 20th heaviest *woman in the 45 to 64 age group*.

Practice

With the sarcopenia data, group by `age_group`, then in a new variable called `grip_strength_rank`, compute the per-age-group rank of each individual's grip strength. (To compute the rank, use `mutate()` and the `rank()` function with its default ties method.)

```
## Complete the code with your answer:
Q_rank_grip_strength <-
  sarcopenia
```

Watch Out

Remember to ungroup data before further analysis

As has been mentioned before, it is important ungroup your data before doing further analysis. Consider this last example, where we computed the weight rank of individuals per age and sex group:

```
yao %>%
  group_by(sex, age_category) %>%
  mutate(weight_rank = rank(desc(weight_kg)))
```

```
# A tibble: 5 x 8
# Groups:   sex, age_category [4]
```

```

sex      age age_category weight_kg occupation      igg_result igm_result
<chr>    <dbl> <chr>          <dbl> <chr>          <chr>      <chr>
1 Female   45 45 - 64           95 Informal worker Negative   Negative
2 Male     55 45 - 64           96 Salaried worker Positive   Negative
3 Male     23 15 - 29           74 Student      Negative   Negative
4 Female   20 15 - 29           70 Student      Positive   Negative
5 Female   55 45 - 64           67 Trader--Farmer Positive    Negative
# i 1 more variable: weight_rank <dbl>

```

If, in the process of analysis, you stored this output as a new data frame:

```

yao_modified <-
  yao %>%
    group_by(sex, age_category) %>%
    mutate(weight_rank = rank(desc(weight_kg)))

```

And then, later on, you picked up the data frame and tried some other analysis, for example, filtering to get the oldest person in the data:

```

yao_modified %>%
  filter(age == max(age))

```

```

# A tibble: 5 x 8
# Groups:   sex, age_category [5]
sex      age age_category weight_kg occupation      igg_result igm_result
<chr>    <dbl> <chr>          <dbl> <chr>          <chr>      <chr>
1 Male     65 45 - 64           93 Retired      Negative   Negative
2 Male     78 65 +           95 Retired--Informal w~ Positive   Negative
3 Male     14 5 - 14           44 Student      Negative   Negative
4 Female   44 30 - 44           67 Home-maker   Positive   Negative
5 Female   79 65 +           40 Retired      Negative   Negative
# i 1 more variable: weight_rank <dbl>

```

You might be confused by the output! Why are there 55 rows of “oldest people”?

This would be because you forgot to ungroup the data before storing it for further analysis. Let’s do this properly now

```

yao_modified <-
  yao %>%
    group_by(sex, age_category) %>%
    mutate(weight_rank = rank(desc(weight_kg))) %>%
    ungroup()

```

Now we can correctly obtain the oldest person/people in the data set:

```

yao_modified %>%
  filter(age == max(age))

```

```

# A tibble: 2 x 8
sex      age age_category weight_kg occupation igg_result igm_result
<chr>    <dbl> <chr>          <dbl> <chr>          <chr>      <chr>

```

```
1 Female      79 65 +                40 Retired      Negative  Negative
2 Female      79 65 +                81 Home-maker  Negative  Negative
# i 1 more variable: weight_rank <dbl>
```

15.8 Wrap up

group_by() is a marvelous tool for arranging, mutating, filtering based on the groups within a single or multiple variables.

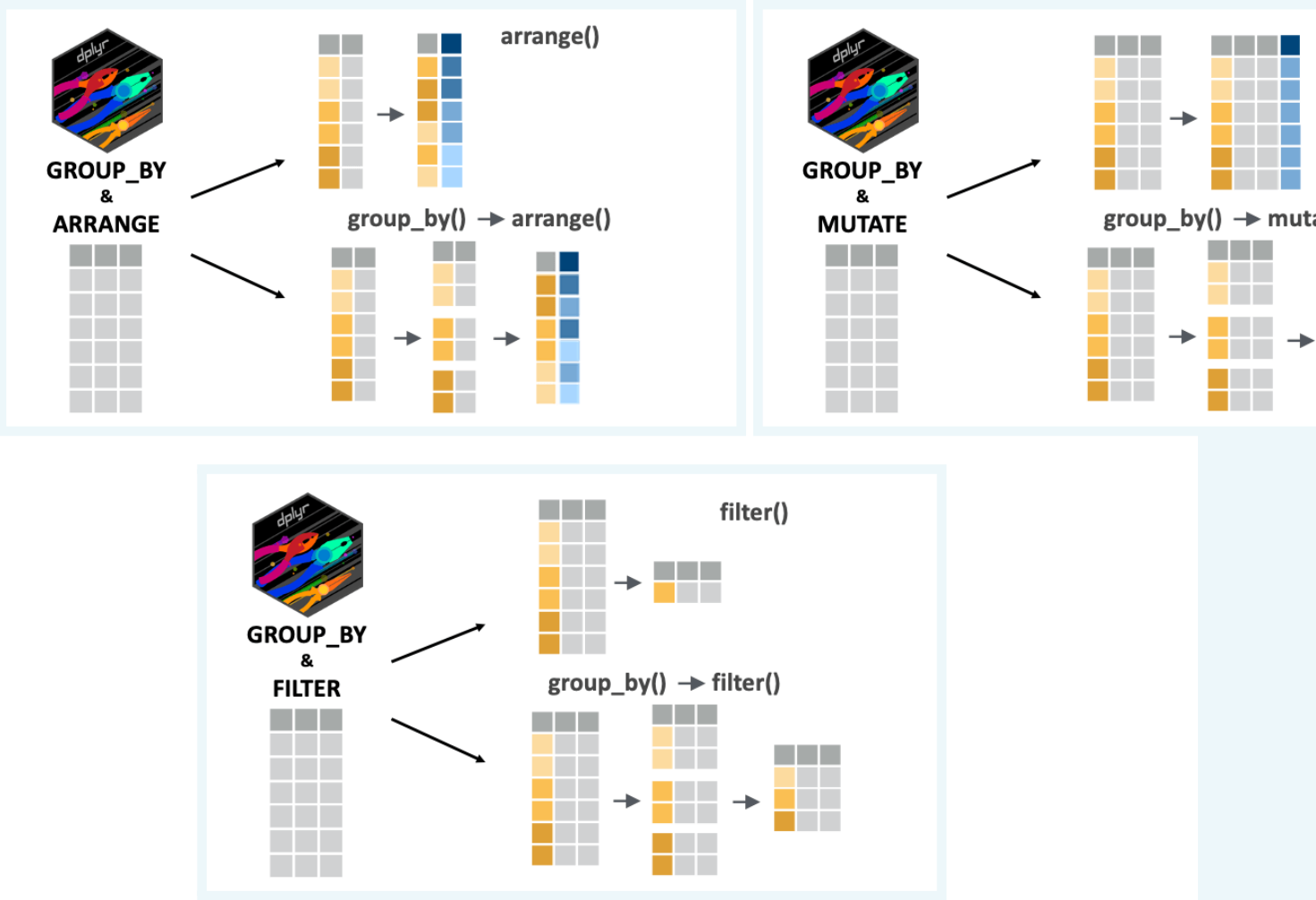


Figure 15.1: Fig: filter() and its use combined with group_by().

There are numerous ways of combining these verbs to manipulate your data. We invite you to take some time and to try these verbs out in different combinations!

See you next time!

References

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

- Horst, A. (2022). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)

- *Group by one or more variables.* (n.d.). Retrieved 21 February 2022, from https://dplyr.tidyverse.org/reference/group_by.html
- *Create, modify, and delete columns — Mutate.* (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/mutate.html>
- *Subset rows using column values — Filter.* (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/filter.html>
- *Arrange rows by column values — Arrange.* (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/arrange.html>

Artwork was adapted from:

- Horst, A. (2022). *R & stats illustrations by Allison Horst.* <https://github.com/allisonhorst/stats-illustrations> (Original work published 2018)

15.9 Solutions

```
.SOLUTION_Q_grip_strength_arranged()
```

```
Q_grip_strength_arranged <-
  sarcopenia %>%
  select(sex_male_1_female_0, grip_strength_kg) %>%
  arrange(sex_male_1_female_0, grip_strength_kg)
```

```
.SOLUTION_Q_age_group_height()
```

```
Q_age_group_height <-
  sarcopenia %>%
  mutate(age_group = factor(age_group, levels = c("Sixties",
                                                  "Seventies",
                                                  "Eighties"))) %>%
  arrange(age_group, height_meters)
```

```
.SOLUTION_Q_max_skeletal_muscle_index()
```

```
Q_max_skeletal_muscle_index <-
  sarcopenia %>%
  group_by(age_group, sex_male_1_female_0) %>%
  filter(skeletal_muscle_index == max(skeletal_muscle_index))
```

```
.SOLUTION_Q_rank_grip_strength()
```

```
Q_rank_grip_strength <-  
  sarcopenia %>%  
  group_by(age_group) %>%  
  mutate(grip_strength_rank = rank(grip_strength_kg))
```

Chapter 16

Pivoting data

16.1 Intro

Pivoting or reshaping is a data manipulation technique that involves re-orienting the rows and columns of a dataset. This is sometimes required to make data easier to analyze, or to make data easier to understand.

In this lesson, we will cover how to effectively pivot data using `pivot_longer()` and `pivot_wider()` from the `tidyr` package.

16.2 Learning Objectives

- You will understand what wide data format is, and what long data format is.
- You will know how to pivot long data to wide data using `pivot_long()`
- You will know how to pivot wide data to long data using `pivot_wider()`
- You will understand why the long data format is easier for plotting and wrangling in R.

16.3 Packages

```
## Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, outbreaks, janitor, rio, here, knitr)
```

16.4 What do wide and long mean?

The terms wide and long are best understood in the context of example datasets. Let's take a look at some now.

Imagine that you have three patients from whom you collect blood pressure data on three days.

You can record the data in a wide format like this:

Or you could record the data in a long format as so :

Take a minute to study the two datasets to make sure you understand the relationship between them.

patient	blood_pressure_day_1	blood_pressure_day_2	blood_pressure_day_3
A	110	112	114
B	120	122	124
C	100	104	105

Figure 16.1: Fig: wide dataset for a timeseries of patients.

patient	day	blood_pressure
A	1	110
A	2	112
A	3	114
B	1	120
B	2	122
B	3	124
C	1	100
C	2	104
C	3	105

Figure 16.2: Fig: long dataset for a timeseries of patients.

In the wide dataset, each observational unit (each patient) occupies only one row. And each measurement. (blood pressure day 1, blood pressure day 2...) is in a separate column.

In the long dataset, on the other hand, each observational unit (each patient) occupies multiple rows, with one row for each measurement.

Here is another example with mock data, in which the observational units are countries:

The examples above are both time-series datasets, because the measurements are repeated across time (day 1, day 2 and so on). But the concepts of long and wide are relevant to other kinds of data too, not just time series data.

Consider the example below, showing the number of patients in different units of three hospitals:

In the wide dataset, again, each observational unit (each hospital) occupies only one row, with the repeated measurements for that unit (number of patients in different rooms) spread across two columns.

In the long dataset, each observational unit is spread over multiple lines.

i Vocab

The “observational units”, sometimes called “statistical units” of a dataset are the primary entities or items described by the columns in that dataset.

In the first example, the observational/statistical units were patients; in the second example, countries, and in the third example, hospitals.

country	year	metric
x	1960	10
x	1970	13
x	2010	15
y	1960	20
y	1970	23
y	2010	25
z	1960	30
z	1970	33
z	2010	35

Figure 16.3: Fig: long dataset where the unique observation unit is a country.

country	yr1960	yr1970	yr2010
x	10	13	15
y	20	23	25
z	30	33	35

Figure 16.4: Fig: the equivalent wide dataset

Hospital	Maternity unit	Intensive care unit
Hospital A	4	2
Hospital B	5	2
Hospital C	6	3

Figure 16.5: Fig: wide dataset, where each hospital is an observational unit

Hospital	Unit	Num. of patients
Hospital A	Maternity	4
Hospital A	Intensive care	2
Hospital B	Maternity	5
Hospital B	Intensive care	2
Hospital C	Maternity	6
Hospital C	Intensive care	3

Figure 16.6: Fig: the equivalent long dataset

Practice

Consider the mock dataset created below:

```
temperatures <-
  data.frame(
    country = c("Sweden", "Denmark", "Norway"),
    avgtemp.1994 = 1:3,
    avgtemp.1995 = 3:5,
    avgtemp.1996 = 5:7)
temperatures
```

```
country avgtemp.1994 avgtemp.1995 avgtemp.1996
1 Sweden           1           3           5
2 Denmark          2           4           6
3 Norway           3           5           7
```

Is this data in a wide or long format?

```
## Enter the string "wide" or the string "long"
## Assign your answer to the object Q_data_type
Q_data_type <- "_____"
## Then run the provided CHECK function
```

16.5 When should you use wide vs long data?

The truth is: it really depends on what you want to do! The wide format is great for *displaying data* because it's easy to visually compare values this way. Long data is best for some data analysis tasks, like grouping and plotting.

It will therefore be essential for you to know how to switch from one format to the other easily. Switching from the wide to the long format, or the other way around, is called **pivoting**.

16.6 Pivoting wide to long

To practice pivoting from a wide to a long format, we'll consider data from [Gapminder](#) on the **number of infant deaths** in specific countries over several years.

i Side Note

[Gapminder](#) is a good source of rich, health-relevant datasets. You are encouraged to peruse their collections.

Below, we read in and view this data on infant deaths:

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

```
# A tibble: 5 x 7
  country      x2010 x2011 x2012 x2013 x2014 x2015
  <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Afghanistan 74600 72000 69500 67100 64800 62700
2 Angola      79100 76400 73700 71200 69000 67200
3 Albania        420   384   354   331   313   301
4 United Arab Emirates 683   687   686   681   672   658
5 Argentina    9550  9230  8860  8480  8100  7720
```

We observe that each observational unit (each country) occupies only one row, with the repeated measurements spread out across multiple columns. Hence this dataset is in a wide format.

To convert to a long format, we can use a convenient function `pivot_longer`. Within `pivot_longer` we define, using the `cols` argument, which columns we want to pivot:

```
infant_deaths_wide %>%
  pivot_longer(cols = x2010:x2015)
```

```
# A tibble: 5 x 3
  country      name value
  <chr>      <chr> <dbl>
1 Afghanistan x2010 74600
2 Afghanistan x2011 72000
3 Afghanistan x2012 69500
4 Afghanistan x2013 67100
5 Afghanistan x2014 64800
```

Very easy!

We can observe that the resulting long format dataset has each country occupying 5 rows (one per year between 2010 and 2015). The years are indicated in the variable `names`, and all the death count values occupy a single variable, `values`.

A useful way to think about this transformation is that the infant deaths values used to be in matrix format (2 dimensions; 2D), but they are now in a vector format (1 dimension; 1D).

This long dataset will be much more handy for many data analysis procedures.

As a good data analyst, you may find the default names of the variables, names and values, to be unsatisfactory; they do not adequately describe what the variables contain. Not to worry; you can give custom column names, using the arguments `names_to` and `values_to`:

```
infant_deaths_wide %>%
  pivot_longer(cols = x2010:x2015,
               names_to = "year",
               values_to = "deaths_count")
```

```
# A tibble: 5 x 3
  country    year deaths_count
  <chr>      <chr>      <dbl>
1 Afghanistan x2010      74600
2 Afghanistan x2011      72000
3 Afghanistan x2012      69500
4 Afghanistan x2013      67100
5 Afghanistan x2014      64800
```

Side Note

Notice that the long format is more informative than the original wide format. Why? Because of the informative column name “deaths_count”. In the wide format, unless the CSV is named something like `count_infant_deaths`, or someone tells you “these are the counts of infant deaths per country and per year”, you have no idea what the numbers in the cells represent.

You may also want to remove the `x` in front of each year. This can be achieved with the convenient `parse_number()` function from the `{readr}` package (part of the tidyverse), which extracts numbers from strings:

```
infant_deaths_wide %>%
  pivot_longer(cols = x2010:x2015,
               names_to = "year",
               values_to = "deaths_count") %>%
  mutate(year = parse_number(year))
```

```
# A tibble: 5 x 3
  country    year deaths_count
  <chr>      <dbl>      <dbl>
1 Afghanistan  2010      74600
2 Afghanistan  2011      72000
3 Afghanistan  2012      69500
4 Afghanistan  2013      67100
5 Afghanistan  2014      64800
```

Great! Now we have a clean, long dataset.

For later use, let’s now store this data:

```
infant_deaths_long <-
  infant_deaths_wide %>%
  pivot_longer(cols = x2010:x2015,
```

```
names_to = "year",
values_to = "deaths_count")
```

Practice

For this practice question, you will use the `euro_births_wide` dataset from [Eurostat](#). It shows the annual number of births in 50 European countries:

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

```
# A tibble: 5 x 8
  country x2015 x2016 x2017 x2018 x2019 x2020 x2021
  <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Belgium 122274 121896 119690 118319 117695 114350 118349
2 Bulgaria 65950 64984 63955 62197 61538 59086 58678
3 Czechia 110764 112663 114405 114036 112231 110200 111793
4 Denmark 58205 61614 61397 61476 61167 60937 63473
5 Germany 737575 792141 784901 787523 778090 773144 795492
```

The data is in a wide format. Convert it to a long format data frame that has the following column names: "country", "year" and "births_count"

```
Q_euro_births_long <-
  euro_births_wide %>% # complete the code with your answer
```

16.7 Pivoting long to wide

Now you know how to pivot from wide to long with `pivot_longer()`. How about going the other way, from long to wide? For this, you can use the fittingly-named `pivot_wider()` function.

But before we consider how to use this function to manipulate long data, let's first consider *where* you're likely to run into long data.

While wide data tends to come from external sources (as we have seen above), long data on the other hand, is likely to be created by *you* while data wrangling, especially in the course of `group_by()`-`summarize()` manipulations.

Let's see an example of this now.

We will use a dataset of patient records from an Ebola outbreak in Sierra Leone in 2014. Below we extract this data from the `{outbreaks}` package and perform some simplifying manipulations on it.

```
ebola <-
  outbreaks::ebola_sierraleone_2014 %>%
  as_tibble() %>%
  mutate(year = lubridate::year(date_of_onset)) %>% # extract the year from the date
  select(patient_id = id, district, year_of_onset = year) # select and rename

ebola
```

```
# A tibble: 5 x 3
  patient_id district year_of_onset
    <int>   <fct>         <dbl>
1         1 Kailahun         2014
2         2 Kailahun         2014
3         3 Kailahun         2014
4         4 Kailahun         2014
5         5 Kailahun         2014
```

Each row corresponds to one patient, and we have each patient's id number, their district and the year in which they contracted Ebola.

Now, consider the following grouped summary of the ebola dataset, which counts the number of patients recorded in each district in each year:

```
cases_per_district_per_year <-
  ebola %>%
    group_by(district) %>%
    count(year_of_onset) %>%
    ungroup()

cases_per_district_per_year
```

```
# A tibble: 5 x 3
  district year_of_onset     n
    <fct>         <dbl> <int>
1 Bo           2014     397
2 Bo           2015     209
3 Bombali      2014    1070
4 Bombali      2015     120
5 Bonthe       2014        7
```

The output of this grouped operation is a quintessentially “long” dataset! Each observational unit (each district) occupies multiple rows (two rows per district, to be exact), with one row for each measurement (each year).

So, as you now see, long data often can arrive as an output of grouped summaries, among other data manipulations.

Now, let's see how to convert such long data into a wide format with `pivot_wider()`.

The code is quite straightforward:

```
cases_per_district_per_year %>%
  pivot_wider(values_from = n,
              names_from = year_of_onset)
```

```
# A tibble: 5 x 3
  district `2014` `2015`
    <fct>    <int> <int>
1 Bo         397   209
2 Bombali   1070   120
```

3 Bonthe	7	77
4 Kailahun	535	35
5 Kambia	127	294

As you can see, `pivot_wider()` has two important arguments: `values_from` and `names_from`. The `values_from` argument defines which values will become the core of the wide data format (in other words: which 1D vector will become a 2D matrix). In our case, these values were in the `n` variable. And `names_from` identifies which variable to use to define column names in the wide format. In our case, this was the `year_of_onset` variable.

Side Note

You might also want to have the *years* be your primary observational/statistical unit, with each year occupying one row. This can be carried out similarly to the above example, but the `district` variable will be provided as an argument to `names_from`, instead of `year_of_onset`.

```
cases_per_district_per_year %>%
  pivot_wider(values_from = n,
              names_from = district)
```

```
# A tibble: 2 x 15
  year_of_onset Bo Bombali Bonthe Kailahun Kambia Kenema Koinadugu Kono
      <dbl> <int>   <int>   <int>   <int>   <int>   <int>   <int> <int>
1      2014   397   1070     7     535   127    641    142   328
2      2015   209    120    77     35   294    139     15   223
# i 6 more variables: Moyamba <int>, `Port Loko` <int>, Pujehun <int>,
#   Tonkolili <int>, `Western Rural` <int>, `Western Urban` <int>
```

Here the unique observation units (our rows) are now the years (2014, 2015).

Practice

The population dataset from the `tidyr` package shows the populations of 219 countries over time. Pivot this data into a wide format. Your answer should have 20 columns and 219 rows.

```
Q_population_widen <-
  tidyr::population
```

16.8 Why is long data better for analysis?

Above we mentioned that long data is best for a majority of data analysis tasks. Now we can justify why. In the sections below, we will go through a few common operations that you will need to do with long data, in each case you will observe that similar manipulations on wide data would be quite tricky.

16.8.1 Filtering grouped data

First, let's talk about filtering grouped data, which is very easy to do on long data, but difficult on wide data.

Here is an example with the infant deaths dataset. Imagine that we want to answer the following question: **For each country, which year had the highest number of child deaths?**

This is how we would do so with the long format of the data :

```
infant_deaths_long %>%
  group_by(country) %>%
  filter(deaths_count == max(deaths_count))
```

```
# A tibble: 5 x 3
# Groups:   country [5]
  country      year deaths_count
  <chr>      <chr>      <dbl>
1 Afghanistan x2010         74600
2 Angola       x2010         79100
3 Albania      x2010           420
4 United Arab Emirates x2011           687
5 Argentina    x2010         9550
```

Easy right? We can easily see, for example, that Afghanistan had its highest infant death count in 2010, and the United Arab Emirates had its highest death count in 2011.

If you wanted to do the same thing with wide data, it would be much more difficult. You could try an approach like this with `rowwise()`:

```
infant_deaths_wide %>%
  rowwise() %>%
  mutate(max_count = max(x2010, x2011, x2012, x2013, x2014, x2015))
```

```
# A tibble: 5 x 8
# Rowwise:
  country      x2010 x2011 x2012 x2013 x2014 x2015 max_count
  <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>
1 Afghanistan 74600 72000 69500 67100 64800 62700    74600
2 Angola      79100 76400 73700 71200 69000 67200    79100
3 Albania       420   384   354   331   313   301       420
4 United Arab Emirates 683   687   686   681   672   658       687
5 Argentina    9550  9230  8860  8480  8100  7720    9550
```

This almost works—we have, for each country, we have the maximum number of child deaths reported—but we still don't know which year is attached to that value in `max_count`. We would have to take that value and index it back to its respective year column somehow... what a hassle! There are solutions to find this but all are very painful. Why make your life complicated when you can just pivot to long format and use the beauty of `group_by()` and `filter()`?

i Side Note

Here we used a special {dplyr} function: `rowwise()`. `rowwise()` allows further operations to be applied *per-row*. It is equivalent to creating one group for each row (`group_by(row_number())`). Without `rowwise()` you would get this:

```
infant_deaths_wide %>%
  mutate(max_count = max(x2010, x2011, x2012, x2013, x2014, x2015))
```

A tibble: 5 x 8

	country	x2010	x2011	x2012	x2013	x2014	x2015	max_count
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Afghanistan	74600	72000	69500	67100	64800	62700	1170000
2	Angola	79100	76400	73700	71200	69000	67200	1170000
3	Albania	420	384	354	331	313	301	1170000
4	United Arab Emirates	683	687	686	681	672	658	1170000
5	Argentina	9550	9230	8860	8480	8100	7720	1170000

...the maximum count over ALL rows in the dataset.

💡 Practice

For this practice question, you will perform a grouped filter on the long format population dataset from the `tidyr` package. Use `group_by()` and `filter()` to obtain a dataset that shows the maximum population recorded for each country, and the year in which that maximum population was recorded.

```
Q_population_max <-
  population
```

16.8.2 Summarizing grouped data

Grouped summaries are also difficult to perform on wide data. For example, considering again the `infant_deaths_long` dataset, if you want to ask: **For each country, what was the mean number of infant deaths and the standard deviation (variation) in deaths?**

With long data it is simple:

```
infant_deaths_long %>%
  group_by(country) %>%
  summarize(mean_deaths = mean(deaths_count),
            sd_deaths = sd(deaths_count))
```

A tibble: 5 x 3

	country	mean_deaths	sd_deaths
	<chr>	<dbl>	<dbl>
1	Afghanistan	68450	4466.
2	Albania	350.	45.2
3	Algeria	21033.	484.
4	Angola	72767.	4513.
5	Antigua and Barbuda	10.7	0.816

With wide data, on the other hand, finding the mean is less intuitive...

```
infant_deaths_wide %>%
  rowwise() %>%
  mutate(mean_deaths = sum(x2010, x2011, x2012,
                           x2013, x2014, x2015, na.rm = T)/6)
```

A tibble: 5 x 8

Rowwise:

	country	x2010	x2011	x2012	x2013	x2014	x2015	mean_deaths
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Afghanistan	74600	72000	69500	67100	64800	62700	68450
2	Angola	79100	76400	73700	71200	69000	67200	72767.
3	Albania	420	384	354	331	313	301	350.
4	United Arab Emirates	683	687	686	681	672	658	678.
5	Argentina	9550	9230	8860	8480	8100	7720	8657.

And finding the standard deviation would be very difficult. (We can't think of any way to achieve this, actually.)

Practice

For this practice question, you will again work with the long format population dataset from the `tidyr` package.

Use `group_by()` and `summarize()` to obtain, for each country, the maximum reported population, the minimum reported population, and the mean reported population across the years available in the data. Your data should have four columns, "country", "max_population", "min_population" and "mean_population".

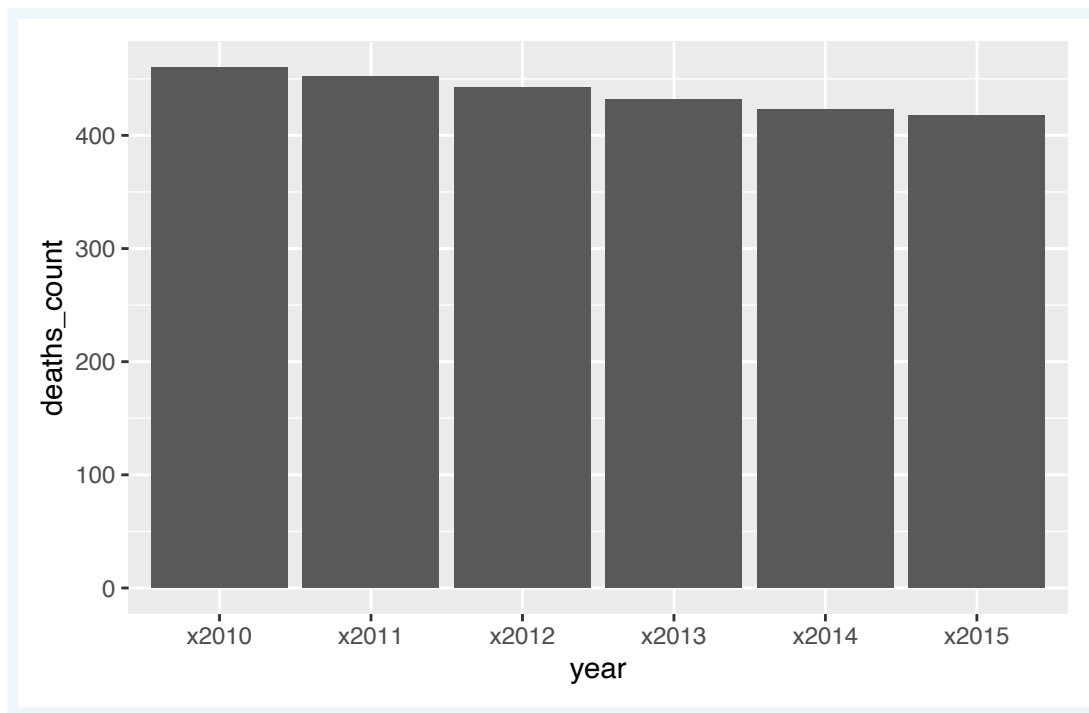
```
Q_population_summaries <-
  population
```

16.8.3 Plotting

Finally, one of the data analysis tasks that is MOST hindered by wide formats is plotting. You may not yet have any prior knowledge of `{ggplot}` and how to plot so we will see the figures without going in depth with the code. What you need to remember is: many plots with `ggplot` are also only possible with long-format data

Consider again the `infant_deaths` data `infant_deaths_long`. We will plot the number of deaths for Belgium per year:

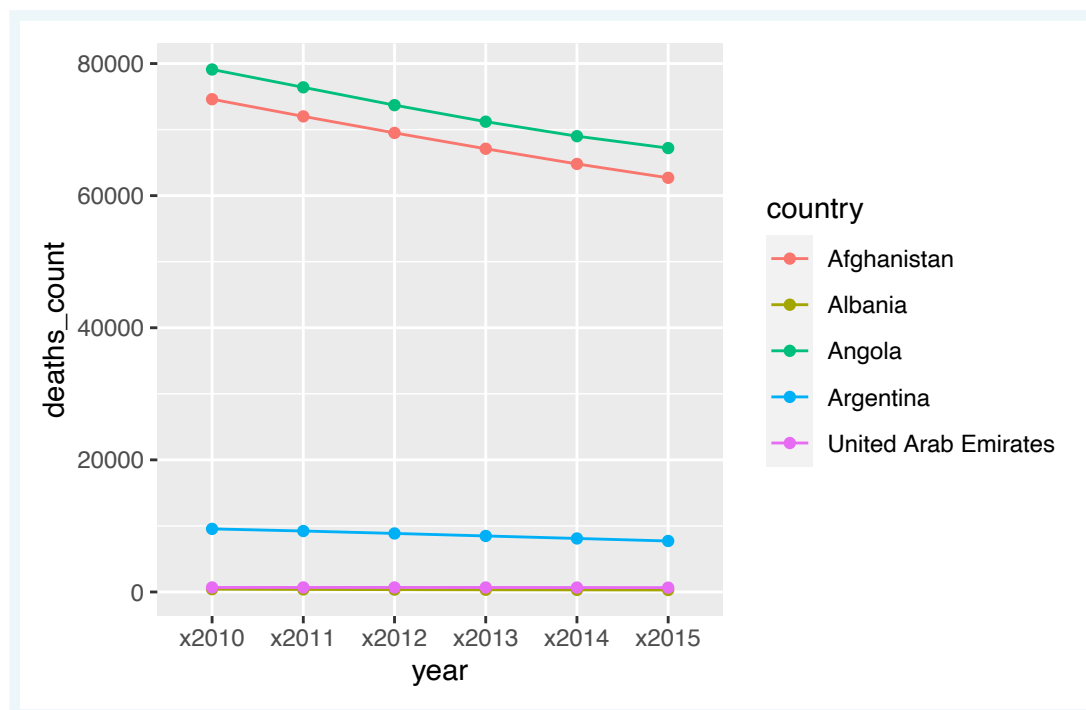
```
infant_deaths_long %>%
  filter(country == "Belgium") %>%
  ggplot() +
  geom_col(aes(x = year, y = deaths_count))
```



The plotting works because we can give the variable `year` for the x-axis. In the long format, `year` is a variable of its own. In the wide format, each there would be no such variable to pass to the x axis.

Another plot that would not be possible without a long format:

```
infant_deaths_long %>%  
  head(30) %>%  
  ggplot(aes(x = year, y = deaths_count, group = country, color = country)) +  
  geom_line() +  
  geom_point()
```



Once again, the reason is the same, we need to tell the plot what to use as an x-axis and a y-axis and it is necessary to have these variables in their own columns (as organized in the long format).

16.9 Pivoting can be hard

We have mostly looked at very simple examples of pivoting here, but in the wild, pivoting can be very difficult to do accurately. This is because the data you are working with may not have all the information necessary for a successful pivot, or the data may contain errors that prevent you from pivoting correctly.

When you run into such cases, we recommend looking at the [official documentation](#) of pivoting from the `tidyr` team, as it is quite rich in examples. You could also post your questions about pivoting on forums like Stack Overflow.

16.10 Wrap up

You have now explored different datasets and how they are either in a long or wide format. In the end, it's just about how you present the information. Sometimes one format will be more convenient, and other times another could be best. Now, you are no longer limited by the format of your data: don't like it? change it !

16.11 Solutions

```
.SOLUTION_Q_data_type()
```

"Wide"

```
.SOLUTION_Q_euro_births_long()
```

```
euro_births_wide %>%  
  pivot_longer(2:8,  
               names_to = "year",  
               values_to = "births_count")
```

```
.SOLUTION_Q_population_widen()
```

```
tidyr::population %>%  
  pivot_wider(names_from = year,  
              values_from = population)
```

```
.SOLUTION_Q_population_max()
```

```
tidyr::population %>%  
  group_by(country) %>%  
  filter(population == max(population)) %>%  
  ungroup()
```

```
.SOLUTION_Q_population_summaries()
```

```
population %>%  
group_by(country) %>%  
summarise(max_population = max(population),  
          min_population = min(population),  
          mean_population = mean(population))
```

Chapter 17

Advanced pivoting

17.1 Intro

You know basic pivoting operations from long format datasets to wide format datasets and vice versa. However, as is often the case, basic manipulations are sometimes not enough for the wrangling you need to do. Let's now see the next level. Let's go !

17.2 Learning Objectives

1. Master complex pivoting from wide to long and long to wide
2. Know how to use separators as a pivoting tool

17.3 Packages

```
## Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, outbreaks, janitor, rio, here, knitr)
```

17.4 Datasets

We will introduce these datasets as we go along but here is an overview:

- Survey data from India on how much money patients spent on tuberculosis treatment
- Biomarker data from an enteropathogen study in Zambia
- A diet survey from Vietnam

17.5 Wide to long

Sometimes you have multiple kinds of wide data in the same table. Consider this artificial example of heights and weights for children over two years:

```
child_stats <-
  tibble::tribble(
    ~child, ~year1_height, ~year2_height, ~year1_weight, ~year2_weight,
    "A",      "80cm",      "85cm",      "5kg",      "10kg",
    "B",      "85cm",      "90cm",      "7kg",      "12kg",
    "C",      "90cm",      "100cm",     "6kg",      "14kg"
  )

child_stats
```

```
# A tibble: 3 x 5
  child year1_height year2_height year1_weight year2_weight
  <chr> <chr>         <chr>         <chr>         <chr>
1 A      80cm         85cm          5kg          10kg
2 B      85cm         90cm          7kg          12kg
3 C      90cm        100cm          6kg          14kg
```

If you pivot all the measurement columns, you'll get overly long data:

```
child_stats %>%
  pivot_longer(2:5)
```

```
# A tibble: 5 x 3
  child name      value
  <chr> <chr>         <chr>
1 A     year1_height 80cm
2 A     year2_height 85cm
3 A     year1_weight 5kg
4 A     year2_weight 10kg
5 B     year1_height 85cm
```

This is not what you (usually) want, because now you have two different kinds of data in the same column—weight and height.

To get the right shape, you'll need to use the `names_sep` argument and the `".value"` identifier:

```
child_stats %>%
  pivot_longer(2:5,
    names_sep = "_",
    names_to = c("period", ".value"))
```

```
# A tibble: 5 x 4
  child period height weight
  <chr> <chr>   <chr>   <chr>
1 A     year1   80cm    5kg
2 A     year2   85cm   10kg
3 B     year1   85cm    7kg
4 B     year2   90cm   12kg
5 C     year1   90cm    6kg
```

Now we have one row for each child-period, an appropriately long format!

What the code above is doing may not be clear, but you should already be able to answer the practice question below by pattern matching with our example. After the practice question, we will explain the `names_sep` argument and the `value` identifier in more depth.

💡 Practice

Consider this other artificial data set:

```
adult_stats <-
  tibble::tribble(
    ~adult, ~year1_BMI, ~year2_BMI, ~year1_HIV, ~year2_HIV,
    "A",      25,      30, "Positive", "Positive",
    "B",      34,      28, "Negative", "Positive",
    "C",      19,      17, "Negative", "Negative"
  )

adult_stats
```

```
# A tibble: 3 x 5
  adult year1_BMI year2_BMI year1_HIV year2_HIV
  <chr>    <dbl>    <dbl> <chr>    <chr>
1 A         25         30 Positive Positive
2 B         34         28 Negative Positive
3 C         19         17 Negative Negative
```

Pivot the data into a long format to get the following structure:

	adult	year	BMI	HIV
1	A	25	30	Positive
2	B	34	28	Negative
3	C	19	17	Negative

```
Q_adult_long <-
  adult_stats %>%
  pivot_longer(_____)
```

i Side Note

The `child_stats` example above has numbers stored as characters [...]

As you saw in the previous lesson, you can easily extract the numbers from the output long data frame in our example using the `parse_number()` function from `readr`:

```
child_stats_long <-
  child_stats %>%
  pivot_longer(2:5,
               names_sep = "_",
               names_to = c("period", ".value"))

child_stats_long
```

```
# A tibble: 5 x 4
  child period height weight
<chr> <chr>   <chr>   <chr>
1 A     year1  80cm    5kg
2 A     year2  85cm   10kg
3 B     year1  85cm    7kg
4 B     year2  90cm   12kg
5 C     year1  90cm    6kg
```

```
child_stats_long %>%
  mutate(height = parse_number(height),
         weight = parse_number(weight))
```

```
# A tibble: 5 x 4
  child period height weight
<chr> <chr>   <dbl>   <dbl>
1 A     year1     80      5
2 A     year2     85     10
3 B     year1     85      7
4 B     year2     90     12
5 C     year1     90      6
```

17.5.1 Understanding names_sep and ".value"

Now let's break down the `pivot_longer()` call we saw above a bit more:

```
child_stats
```

```
# A tibble: 3 x 5
  child year1_height year2_height year1_weight year2_weight
<chr> <chr>         <chr>         <chr>         <chr>
1 A     80cm       85cm         5kg          10kg
2 B     85cm       90cm         7kg          12kg
3 C     90cm      100cm        6kg          14kg
```

```
child_stats %>%
  pivot_longer(2:5,
               names_sep = "_",
               names_to = c("period", ".value"))
```

```
# A tibble: 5 x 4
  child period height weight
  <chr> <chr>  <chr>  <chr>
1 A     year1  80cm   5kg
2 A     year2  85cm  10kg
3 B     year1  85cm   7kg
4 B     year2  90cm  12kg
5 C     year1  90cm   6kg
```

Notice that the column names in the original `child_stats` data frame (`year1_height`, `year2_height` and so on) are made of three parts:

- the period being referenced: e.g. "year1"
- an underscore separator, "_";
- and the type of value recorded "height" or "weight"

We can make a table with these parts:

column_name	period	separator	".value"
year1_height	year1	_	height
year2_height	year2	_	height
year1_weight	year1	_	weight
year2_weight	year2	_	weight

Based on that table, it should now be easier to understand the `names_sep` and `names_to` arguments that we supplied to `pivot_longer()`:

17.5.1.1 `names_sep = "_"`:

This is the separator between the period indicator (year) and the values (year and weight) recorded.

If we have a different separator, this argument would change. For example, if the separator were an empty space, " ", you would have `names_sep = " "`, as seen in the example below:

```
child_stats_space_sep <-
  tibble::tribble(
    ~child, ~`yr1 height`, ~`yr2 height`, ~`yr1 weight`, ~`yr2 weight`,
    "A",      "80cm",      "85cm",      "5kg",      "10kg",
    "B",      "85cm",      "90cm",      "7kg",      "12kg",
    "C",      "90cm",      "100cm",     "6kg",      "14kg"
  )

child_stats_space_sep %>%
  pivot_longer(2:5,
    names_sep = " ",
    names_to = c("period", ".value"))
```

```
# A tibble: 5 x 4
  child period height weight
  <chr> <chr>   <chr>   <chr>
1 A     yr1     80cm    5kg
2 A     yr2     85cm   10kg
3 B     yr1     85cm    7kg
4 B     yr2     90cm   12kg
5 C     yr1     90cm    6kg
```

17.5.1.2 `names_to = c("period", ".value")`

Next, the `names_to` argument indicates how the data should be reshaped. We passed a vector of two character strings, “period” and the “.value” to this argument. Let’s consider each in turn:

The **“period” string** indicated that we want to move the data from each year (or period) into a separate row. Note that there is nothing special about the word “period” used here; we could change this to any other string. So instead of “period”, you could have written “time” or “year_of_measurement” or anything else:

```
child_stats %>%
  pivot_longer(2:5,
               names_sep = "_",
               names_to = c("year_of_measurement", ".value"))
```

```
# A tibble: 5 x 4
  child year_of_measurement height weight
  <chr> <chr>                 <chr>   <chr>
1 A     year1                80cm    5kg
2 A     year2                85cm   10kg
3 B     year1                85cm    7kg
4 B     year2                90cm   12kg
5 C     year1                90cm    6kg
```

Now, the **“.value” placeholder** is a special indicator, that tells `pivot_longer()` to make a separate column for every distinct value that appears after the separator. In our example, these distinct values are “height” and “weight”.

The “.value” string cannot be arbitrarily replaced. For example, this won’t work:

```
child_stats %>%
  pivot_longer(2:5,
               names_sep = "_",
               names_to = c("period", "values"))
```

```
# A tibble: 5 x 4
  child period values value
  <chr> <chr>   <chr>   <chr>
1 A     year1 height 80cm
2 A     year2 height 85cm
3 A     year1 weight 5kg
4 A     year2 weight 10kg
5 B     year1 height 85cm
```

To restate the point, the “.value” placeholder tells `pivot_longer()` that we want to separate out the “height” and “weight” values into separate columns, because there are the two value types that occur after the “_” separator in the column names.

This means that if you had a wide dataset with three types of values, you would get separated-out columns, one for each value type. For example, consider the mock dataset below which shows children’s records, at two time points, for the following variables:

- age in months,
- body fat %
- bmi

```
child_stats_three_values <-
  tibble::tribble(
    ~child, ~t1_age, ~t2_age, ~t1_fat, ~t2_fat, ~t1_bmi, ~t2_bmi,
    "a", "5mths", "8mths", "13%", "15%", 14, 15,
    "b", "7mths", "9mths", "15%", "17%", 16, 18
  )
child_stats_three_values
```

```
# A tibble: 2 x 7
  child t1_age t2_age t1_fat t2_fat t1_bmi t2_bmi
<chr> <chr> <chr> <chr> <chr> <dbl> <dbl>
1 a      5mths 8mths 13%   15%    14    15
2 b      7mths 9mths 15%   17%    16    18
```

Here, in the column names there are three value types occurring after the “_” separator: age, fat and bmi; the “.value” string tells `pivot_longer()` to make a new column for each value type:

```
child_stats_three_values %>%
  pivot_longer(2:7,
    names_sep = "_",
    names_to = c("time", ".value")
  )
```

```
# A tibble: 4 x 5
  child time age fat bmi
<chr> <chr> <chr> <chr> <dbl>
1 a      t1 5mths 13% 14
2 a      t2 8mths 15% 15
3 b      t1 7mths 15% 16
4 b      t2 9mths 17% 18
```

Practice

A pediatrician records the following information for a set of children over two years:

- head circumference;

- neck circumference; and
- hip circumference

all in centimeters.

The output table resembles the below:

```
growth_stats <-
  tibble::tribble(
    ~child,~yr1_head,~yr2_head,~yr1_neck,~yr2_neck,~yr1_hip,~yr2_hip,
    "a",      45,      48,      23,      24,      51,      52,
    "b",      48,      50,      24,      26,      52,      52,
    "c",      50,      52,      24,      27,      53,      54
  )

growth_stats
```

```
# A tibble: 3 x 7
  child yr1_head yr2_head yr1_neck yr2_neck yr1_hip yr2_hip
  <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 a         45     48     23     24     51     52
2 b         48     50     24     26     52     52
3 c         50     52     24     27     53     54
```

Pivot the data into a long format to get the following structure:

child	year	head	neck	hip
-------	------	------	------	-----

```
Q_growth_stats_long <-
  growth_stats %>%
  pivot_longer(_____)
```

17.5.2 Value type before the separator

In all the example we have used so far, the column names were constructed such that value type came after the separator (Recall our table:

column_name	period	separator	".value"
year1_height	year1	_	height
year2_height	year2	_	height
year1_weight	year1	_	weight
year2_weight	year2	_	weight

)

But of course, the column names could be constructed differently, with the value types coming before the separator, as in this example:

```
child_stats2 <-
  tibble::tribble(
    ~child, ~height_year1, ~height_year2, ~weight_year1, ~weight_year2,
    "A",      "80cm",      "85cm",      "5kg",      "10kg",
    "B",      "85cm",      "90cm",      "7kg",      "12kg",
    "C",      "90cm",      "100cm",     "6kg",      "14kg"
  )

child_stats2
```

```
# A tibble: 3 x 5
  child height_year1 height_year2 weight_year1 weight_year2
  <chr> <chr>         <chr>         <chr>         <chr>
1 A      80cm         85cm          5kg          10kg
2 B      85cm         90cm          7kg          12kg
3 C      90cm        100cm          6kg          14kg
```

Here, the value types (height and weight) come before the “_” separator.

How can our `pivot_longer()` command accommodate this? Simple! Just swap the order of the vector given to the `names_to` argument:

So instead of `names_to = c("time", ".value")`, you would have `names_to = c(".value", "time")`:

```
child_stats2 %>%
  pivot_longer(2:5,
    names_sep = "_",
    names_to = c(".value", "time"))
```

```
# A tibble: 5 x 4
  child time height weight
  <chr> <chr> <chr> <chr>
1 A     year1 80cm  5kg
2 A     year2 85cm 10kg
3 B     year1 85cm  7kg
4 B     year2 90cm 12kg
5 C     year1 90cm  6kg
```

And that’s it!

Practice

Consider the following [data set from Zambia](#) about enteropathogens and their biomarkers.

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

enteropathogens_zambia_wide
```

```
# A tibble: 5 x 7
```

```

      ID LPS_1 LPS_2 LBP_1 LBP_2 IFABP_1 IFABP_2
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  1002  222.  390. 38414. 6840.  1294.   610.
2  1003  181.   NA 26888.   NA   22.5    NA
3  1004  257.  221. 49183. 5426.    0      0
4  1005   NA  369.   NA 1938.    0  1010.
5  1006  275.   NA 61758.   NA    0     NA

```

This data frame has the following columns:

- LPS_1 and LPS_2: lipopolysaccharide levels, measured by Pyrochrome LAL, in EU/mL
- LBP_1 and LBP_2: LPS binding protein levels, in pg/mL
- IFABP_1 and IFAPB_2: intestinal-type fatty acid binding protein levels, in pg/mL

Pivot the dataset so that it resembles the following structure

ID	sample_count	LPS	LBP	IFABP
----	--------------	-----	-----	-------

```

enteropathogens_zambia_wide %>%
  pivot_longer(_____)

```

17.5.3 A non-time-series example

So far we have been using person-period (time series) datasets to illustrate the idea of complex pivots with multiple value types.

But as we have mentioned, not all reshape-requiring datasets are time series data. Let's see a quick non-time-series example [...]

You might measure the height (cm) and weight (kg) of a series of parental couples in a table like this:

```

family_stats <-
  tibble::tribble(
    ~couple, ~father_height, ~father_weight, ~mother_height, ~mother_weight,
    "a",      180,           80,           160,           70,
    "b",      185,           90,           150,           76,
    "c",      182,           93,           143,           78
  )
family_stats

```

```
# A tibble: 3 x 5
```

```

  couple father_height father_weight mother_height mother_weight
  <chr>      <dbl>         <dbl>         <dbl>         <dbl>
1 a          180            80            160            70
2 b          185            90            150            76
3 c          182            93            143            78

```

Here we have two different types of values (weight and height) for each person in the couple.

To pivot this to one-row per person, we'll again need the `names_sep` and `names_to` arguments:

```
family_stats %>%
  pivot_longer(2:5,
    names_sep = "_",
    names_to = c("person", ".value"))
```

```
# A tibble: 5 x 4
  couple person height weight
  <chr>   <chr>   <dbl>   <dbl>
1 a     father    180     80
2 a     mother    160     70
3 b     father    185     90
4 b     mother    150     76
5 c     father    182     93
```

The separator is an underscore, “_”, so we used `names_sep = “_”` and because the value types come after the separator, the “`.value`” identifier was placed second in the `names_to` argument.

17.5.4 Escaping the dot separator

A special example may crop up when you try to pivot a dataset where the separator is a period.

```
child_stats_dot_sep <-
  tibble::tribble(
    ~child, ~year1.height, ~year2.height, ~year1.weight, ~year2.weight,
    "A",      "80cm",      "85cm",      "5kg",      "10kg",
    "B",      "85cm",      "90cm",      "7kg",      "12kg",
    "C",      "90cm",      "100cm",     "6kg",      "14kg"
  )

child_stats_dot_sep %>%
  pivot_longer(2:5,
    names_to = c("period", ".value"),
    names_sep = "\\.")
```

```
# A tibble: 5 x 4
  child period height weight
  <chr> <chr>   <chr>   <chr>
1 A    year1  80cm    5kg
2 A    year2  85cm   10kg
3 B    year1  85cm    7kg
4 B    year2  90cm   12kg
5 C    year1  90cm    6kg
```

There we used the string “`\\.`” to indicate a dot “`.`” because the “`.`” is a special character in R, and sometimes needs to be [escaped](#)

Practice

Consider again the `adult_stats` data you saw above. Now the column names have been changed slightly.

```
adult_stats_dot_sep <-
  tibble::tribble(
    ~adult, ~`BMI.year1`, ~`BMI.year2`, ~`HIV.year1`, ~`HIV.year2`,
    "A",      25,      30,    "Positive", "Positive",
    "B",      34,      28,    "Negative", "Positive",
    "C",      19,      17,    "Negative", "Negative"
  )

adult_stats_dot_sep
```

```
# A tibble: 3 x 5
  adult BMI.year1 BMI.year2 HIV.year1 HIV.year2
  <chr>   <dbl>   <dbl> <chr>   <chr>
1 A         25       30 Positive Positive
2 B         34       28 Negative Positive
3 C         19       17 Negative Negative
```

Again, pivot the data into a long format to get the following structure:

adult	year	BMI	HIV
-------	------	-----	-----

```
Q_adult2_long <-
  adult_stats_dot_sep %>%
  pivot_longer(_____)
```

17.5.5 What to do when you don't have a neat separator ?

Sometimes you do not have a neat separator.

Consider this [survey data from India](#) that looked at how much money patients spent on tuberculosis treatment:

```
tb_visits <- read_csv(here("data/india_tb_pathways_and_costs_data.csv")) %>%
  clean_names() %>%
  select(id, first_visit_location, first_visit_cost, second_visit_location, second_visit_cost,
  tb_visits
```

```
# A tibble: 5 x 7
  id first_visit_location first_visit_cost second_visit_location
  <dbl> <chr>                <dbl> <chr>
```

```

1 100202 GH                                0 <NA>
2 100396 Pvt. docto                        1500 Pvt. clini
3 100590 Pvt. docto                        2000 Pvt. docto
4 100687 Pvt. hospi                        20000 Pvt. hospi
5 100784 Pvt. docto                        1000 GH
# i 3 more variables: second_visit_cost <dbl>, third_visit_location <chr>,
#   third_visit_cost <dbl>

```

It does not have a neat separator between the time indicators (first, second, third) and the value type (cost, location). That is, rather than something like “firstvisit_location”, we have instead “first_visit_location”, so the underscore is used for two purposes. For this reason, if you try our usual pivot strategy, you will get an error:

```

tb_visits %>%
  pivot_longer(2:7,
    names_to = c("visit_count", ".value"),
    names_sep = "_")

```

```

Error in `pivot_longer_spec()` :
! Can't combine `first_visit_location` <character> and `first_visit_cost` <double>.
Run `rlang::last_error()` to see where the error occurred.

```

The most direct way to reshape this dataset successfully would be to use special “regex” (string manipulation), but you likely have not learned this yet!

So for now, the solution we recommend is to manually rename your columns to insert a clear separator, “__”:

```

tb_visits_renamed <-
  tb_visits %>%
  rename(first__visit_location = first_visit_location,
    first__visit_cost = first_visit_cost,
    second__visit_location = second_visit_location,
    second__visit_cost = second_visit_cost,
    third__visit_location = third_visit_location,
    third__visit_cost = third_visit_cost)

tb_visits_renamed

```

```

# A tibble: 5 x 7
  id first__visit_location first__visit_cost second__visit_location
  <dbl> <chr>                <dbl> <chr>
1 100202 GH                0 <NA>
2 100396 Pvt. docto        1500 Pvt. clini
3 100590 Pvt. docto        2000 Pvt. docto
4 100687 Pvt. hospi        20000 Pvt. hospi
5 100784 Pvt. docto        1000 GH
# i 3 more variables: second__visit_cost <dbl>, third__visit_location <chr>,
#   third__visit_cost <dbl>

```

Now we can try the pivot:

```
tb_visits_long <-
  tb_visits_renamed %>%
  pivot_longer(2:7,
               names_to = c("visit_count", ".value"),
               names_sep = "__")
tb_visits_long
```

```
# A tibble: 5 x 4
      id visit_count visit_location visit_cost
  <dbl> <chr>      <chr>          <dbl>
1 100202 first      GH              0
2 100202 second    <NA>           0
3 100202 third     <NA>           0
4 100396 first     Pvt. docto     1500
5 100396 second    Pvt. clini     1000
```

Now let's polish the data frame:

```
tb_visits_long %>%
  # remove nonexistent entries
  filter(!visit_location == "") %>%
  # give significant naming to the visit_count values
  mutate(visit_count = case_when(visit_count == "first" ~ 1,
                                visit_count == "second" ~ 2,
                                visit_count == "third" ~ 3)) %>%
  # ensure visit_cost is numerical
  mutate(visit_cost = as.numeric(visit_cost))
```

```
# A tibble: 5 x 4
      id visit_count visit_location visit_cost
  <dbl>   <dbl> <chr>          <dbl>
1 100202       1 GH              0
2 100396       1 Pvt. docto     1500
3 100396       2 Pvt. clini     1000
4 100396       3 Pvt. hospi     2500
5 100590       1 Pvt. docto     2000
```

Above, we first remove the entries where we do not have the visit location information (i.e. we filter out the rows where the visit location variable is set to ""). We then convert to numeric values the visit count variable, where the strings "first" to "third" are converted to numerical entries 1 to 3. Finally, we ensure the variable of visit cost is numeric using `mutate()` and the helper function `as.numeric()`.

Practice

We will use [a survey data about diet from Vietnam](#). Women in Hanoi were interviewed about their food shopping, and this was used to create nutrition profiles for each women. Here we will use a subset of this data for 61 households who came for 2 visits, recording:

- `enerc_kcal_w_1`: the consumed energy from ingredient/food (Kcal) during the first visit (with `_2` for the second visit)

- `dry_w_1`: the consumed dry from ingredient/food (g) during the first visit (with `_2` for the second visit)
- `water_w_1`: the consumed water from ingredient/food (g) during the first visit (with `_2` for the second visit)
- `fat_w_1`: the consumed Lipid from ingredient/food (g) during the first visit (with `_2` for the second visit)

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

diet_diversity_vietnam_wide
```

```
# A tibble: 5 x 9
  household_id enerc_kcal_w_1 enerc_kcal_w_2 dry_w_1 dry_w_2 water_w_1 water_w_2
      <dbl>         <dbl>         <dbl>   <dbl>   <dbl>     <dbl>     <dbl>
1         348         2268.         1386.    548.    281.     4219.     1997.
2         354         2775.         1240.    600.    284.     2376.     3145.
3          53         3104.         2075.    646.    451.     2808.     2305.
4          18         2802.         2146.    620.    807.     3457.     1903.
5         211         1298.         1191.    269.    288.     2584.     2269.
# i 2 more variables: fat_w_1 <dbl>, fat_w_2 <dbl>
```

You should first distinguish if we have a neat operator or not. Based on this, rename your columns if necessary. Then bring the different visit records (1 and 2) into a sole column for energy, fat weight, water weight and dry weight. In other words, pivot the dataset into long format of this form:

household_id	visit	enerc_kcal_w	dry_w	water_w	fat_w
--------------	-------	--------------	-------	---------	-------

```
Q_diet_diversity_vietnam_long <-
  diet_diversity_vietnam_wide %>%
  pivot_longer(_____)
```

17.6 Long to wide

We just saw how to do some complex operations wide to long, which we saw in the previous lesson is essential for plotting and wrangling. Let's see the opposite transformation.

It could be useful to put long to wide to do different transformations, filters, and processing NAs. In this format, your measurements / collected data become the columns of the data set.

Let's take the Zambia enteropathogen data, and this time, let's take the original ! Indeed, what you were handling before was a dataset **prepared for you**, in a wide format. **The original dataset is long** and we will now see the data preparation I did beforehand, behind the scenes. You're almost becoming the teacher of this lesson ;)

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

```
# A tibble: 5 x 5
  ID group  LPS    LBP  IFABP
<dbl> <dbl> <dbl> <dbl> <dbl>
1 1002     1 222. 38414. 1294.
2 1002     2 390.  6840.  610.
3 1003     1 181. 26888.  22.5
4 1004     2 221.  5426.    0
5 1004     1 257. 49183.    0
```

This is how we convert it from long to wide:

```
enteropathogens_zambia_wide <-
  enteropathogens_zambia_long %>%
  pivot_wider(
    names_from = group,
    values_from = c(LPS, LBP, IFABP)
  )

enteropathogens_zambia_wide
```

```
# A tibble: 5 x 7
  ID LPS_1 LPS_2  LBP_1 LBP_2 IFABP_1 IFABP_2
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 1002 222.  390. 38414. 6840. 1294.  610.
2 1003 181.   NA 26888.   NA  22.5   NA
3 1004 257. 221. 49183. 5426.    0     0
4 1005  NA 369.   NA 1938.    0 1010.
5 1006 275.   NA 61758.   NA    0     NA
```

You can see that the values of the variable `group` (1 or 2) are added to the values' names (LPS, LBP, IFABP) to create the new columns representing different group data: for example, `LPS_1` and `LPS_2`.

We are considering this “advanced” pivoting because we are pivoting wider several variables at the same time, but as you can see, the syntax is quite simple—the same arguments are used as we did with the simpler pivots in the previous lesson—`names_from` and `values_from`.

Let's see another example, using the diet survey data from Vietnam that you manipulated previously:

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

```
# A tibble: 5 x 6
  visit_number household_id enerc_kcal_w dry_w water_w fat_w
      <dbl>         <dbl>         <dbl> <dbl>   <dbl> <dbl>
```

1	1	348	2268.	548.	4219.	78.4
2	1	354	2775.	600.	2376.	115.
3	1	53	3104.	646.	2808.	127.
4	1	18	2802.	620.	3457.	87.4
5	1	211	1298.	269.	2584.	47.8

Here we will use the `visit_number` variable to create new variable for energy, water, fat and dry content of foods recorded at different visits:

```
diet_diversity_vietnam_wide <-
  diet_diversity_vietnam_long %>%
  pivot_wider(
    names_from = visit_number,
    values_from = c(enerc_kcal_w, dry_w, water_w, fat_w)
  )

diet_diversity_vietnam_wide
```

```
# A tibble: 5 x 9
  household_id enerc_kcal_w_1 enerc_kcal_w_2 dry_w_1 dry_w_2 water_w_1 water_w_2
      <dbl>         <dbl>         <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1         1         348         2268.    1386.    548.    281.    4219.
2         2         354         2775.    1240.    600.    284.    2376.
3         3          53         3104.    2075.    646.    451.    2808.
4         4          18         2802.    2146.    620.    807.    3457.
5         5         211         1298.    1191.    269.    288.    2584.
# i 2 more variables: fat_w_1 <dbl>, fat_w_2 <dbl>
```

You can see that the values of the variable `visit_number` (1 or 2) are added to the values' names (energy_kcal_w, dry_w, fat_w, water_w) to create the new columns representing different group data: for example, `water_w_1` and `water_w_2`. We have pivoted to wide format all of these variables at the same time. Now each weight measure per visit is represented as a single variable (i.e. column) in the dataset.

With this format, it is easy to sum together the energy intake per household for example:

```
diet_diversity_vietnam_wide %>%
  select(household_id, enerc_kcal_w_1, enerc_kcal_w_2) %>%
  mutate(total_energy_kcal = enerc_kcal_w_1 + enerc_kcal_w_2) %>%
  arrange(household_id)
```

```
# A tibble: 5 x 4
  household_id enerc_kcal_w_1 enerc_kcal_w_2 total_energy_kcal
      <dbl>         <dbl>         <dbl>         <dbl>
1         1         14         1040.         1663.         2704.
2         2         17         2100.         1286.         3386.
3         3         18         2802.         2146.         4948.
4         4         22         3187.         1582.         4769.
5         5         24         2359.         2026.         4385.
```

However, you could get something similar in the long format:

```
diet_diversity_vietnam_long %>%
  group_by(household_id) %>%
  summarize(total_energy = sum(enerc_kcal_w))
```

```
# A tibble: 5 x 2
  household_id total_energy
      <dbl>         <dbl>
1          14         2704.
2          17         3386.
3          18         4948.
4          22         4769.
5          24         4385.
```

Practice

Take `tb_visits_long` dataset that we manipulated above and pivot it back to a wide format.

```
Q_tb_visit_wide <-
  tb_visits_long %>%
  pivot_wider(_____)
```

17.7 Wrap up

Your data wrangling skills have just been enhanced with advanced pivoting. This skill will often prove essential when handling real world data. I have no doubt you will soon put it into practice. It is also essential, as we have seen, for plotting. So I hope pivoting will be of use not only for your wrangling, but also for your plotting tasks.

17.8 Solutions

```
.SOLUTION_Q_adult_long()
```

```
adult_stats %>%
  pivot_longer(cols = 2:5,
               names_sep = "_",
               names_to = c("year", ".value"))
```

```
.SOLUTION_Q_growth_stats_long()
```

```
growth_stats %>%
  pivot_longer(cols = 2:7,
               names_to = c("year", ".value"),
               names_sep = "_")
```

```
.SOLUTION_Q_adult2_long()
```

```
adult_stats_dot_sep %>%
  pivot_longer(cols = 2:5,
               names_sep = "\\.",
               names_to = c(".value", "year"))
```

```
.SOLUTION_Q_diet_diversity_vietnam_long()
```

```
diet_diversity_vietnam_wide%>%
  rename(
    enerc_kcal_w__1 = enerc_kcal_w_1,
    enerc_kcal_w__2 = enerc_kcal_w_2,
    dry_w__1 = dry_w_1,
    dry_w__2 = dry_w_2,
    water_w__1 = water_w_1,
    water_w__2 = water_w_2,
    fat_w__1 = fat_w_1,
    fat_w__2 = fat_w_2
  ) %>%
  pivot_longer(2:9, names_sep = "__", names_to = c(".value", "visit"))
```

```
.SOLUTION_Q_tb_visit_wide()
```

```
tb_visits_long %>%
  pivot_wider(names_from = visit_count,
              values_from = c(visit_location, visit_cost))
)
```

```
.SOLUTION_Q_adult_long()
```

```
adult_stats %>%
  pivot_longer(cols = 2:5,
               names_sep = "_",
               names_to = c("year", ".value"))
```

```
.SOLUTION_Q_growth_stats_long()
```

```
growth_stats %>%
  pivot_longer(cols = 2:7,
               names_to = c("year", ".value"),
               names_sep = "_")
```

```
.SOLUTION_Q_adult2_long()
```

```
adult_stats_dot_sep %>%
  pivot_longer(cols = 2:5,
               names_sep = "\\.",
               names_to = c(".value", "year"))
```

```
.SOLUTION_Q_diet_diversity_vietnam_long()
```

```
diet_diversity_vietnam_wide%>%
  rename(
    enerc_kcal_w__1 = enerc_kcal_w_1,
    enerc_kcal_w__2 = enerc_kcal_w_2,
    dry_w__1 = dry_w_1,
    dry_w__2 = dry_w_2,
    water_w__1 = water_w_1,
    water_w__2 = water_w_2,
    fat_w__1 = fat_w_1,
    fat_w__2 = fat_w_2
  ) %>%
  pivot_longer(2:9, names_sep = "__", names_to = c(".value", "visit"))
```

```
.SOLUTION_Q_tb_visit_wide()
```

```
tb_visits_long %>%  
  pivot_wider(names_from = visit_count,  
              values_from = c(visit_location, visit_cost))  
)
```

Chapter 18

Intro to ggplot2

18.1 Introduction

Welcome to The GRAPH Courses' Data Visualization course!

We will focus on learning how to use the **{ggplot2} package** to produce high quality visualizations in R.

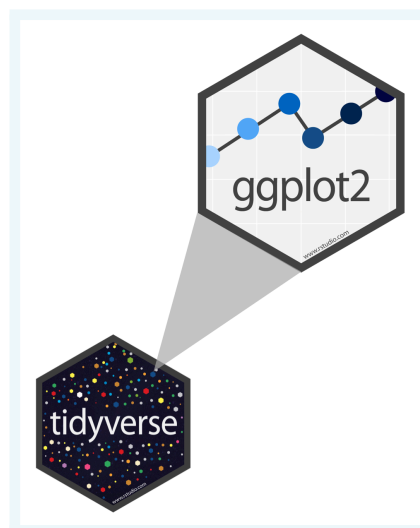


Figure 18.1: {ggplot2} is one of the core packages of the {tidyverse} metapackage. It is the most popular R package for data visualization.

Let's dive in!

18.2 Learning objectives

By the end of this lesson you should be able to:

1. Recall and explain how the **{ggplot2}** package for data visualization is based on a theoretical framework called the **grammar of graphics**.
2. Name and describe the 3 essential components required for building a graph: **data**, **aesthetics**, and **geometries**.

3. Write code to **build a complete ggplot graphic** by correctly supplying the 3 essential layers to the `ggplot()` **function**.
4. Create different types of plots such as **scatter plots**, **line graphs**, and **bar graphs**.
5. Add or modify visual elements of a plot such as **color** and **size**.
6. Distinguish between **aesthetic mappings** and **fixed aesthetics**, and how to apply them.



Figure 18.2: Illustration by Allison Horst

18.3 Packages

The `{tidyverse}` meta package includes `{ggplot2}`, so we don't need to add it separately. The `{here}` package will help us correctly reference file paths.

```
## Load packages
pacman::p_load(tidyverse,
               here)
```

18.4 Measles outbreaks in Niger

In this lesson, we will explore patterns of measles outbreaks in Niger.

Measles is a **highly infectious virus** spread by airborne respiratory droplets.

[Slide presentation about geography]

Since it is transmitted through direct contact, **population density** is an important driver of measles dynamics.

18.4.1 The `nigerm` dataset

We will be creating plots with a dataset of weekly reported measles cases at the region level in Niger.

These data were collected by the Ministry of Health of Niger, from 1 Jan 1995 to 31 Dec 2005.

To get started, let's first load the (preprocessed) data set:

```
## Import data frame to RStudio Environment  
load(here("data/clean/nigerm_cases_rgn.RData"))
```

Take a moment to browse through the data:

```
## Print Niger measles (nigerm) data frame  
nigerm
```

year	week	region	cases
1995	1	Agadez	0
1995	1	Diffa	0
1995	1	Dosso	4
1995	1	Maradi	64
1995	1	Niamey	22
1995	1	Tahoua	16
1995	1	Tillaberi	1
1995	1	Zinder	3
1995	2	Agadez	0
1995	2	Diffa	0

1–10 of 4576 rows

Previous **1** 2 3 4 5 ... 458 Next

The `nigerm` data frame has 4 variables (or columns):

1. **year**: Calendar year (ranges from 1995 to 2005)
2. **week**: Week of the year (ranges from 1 to 52)

3. region: Region in which the cases were recorded (see figure below)
4. cases: Number of measles cases reported

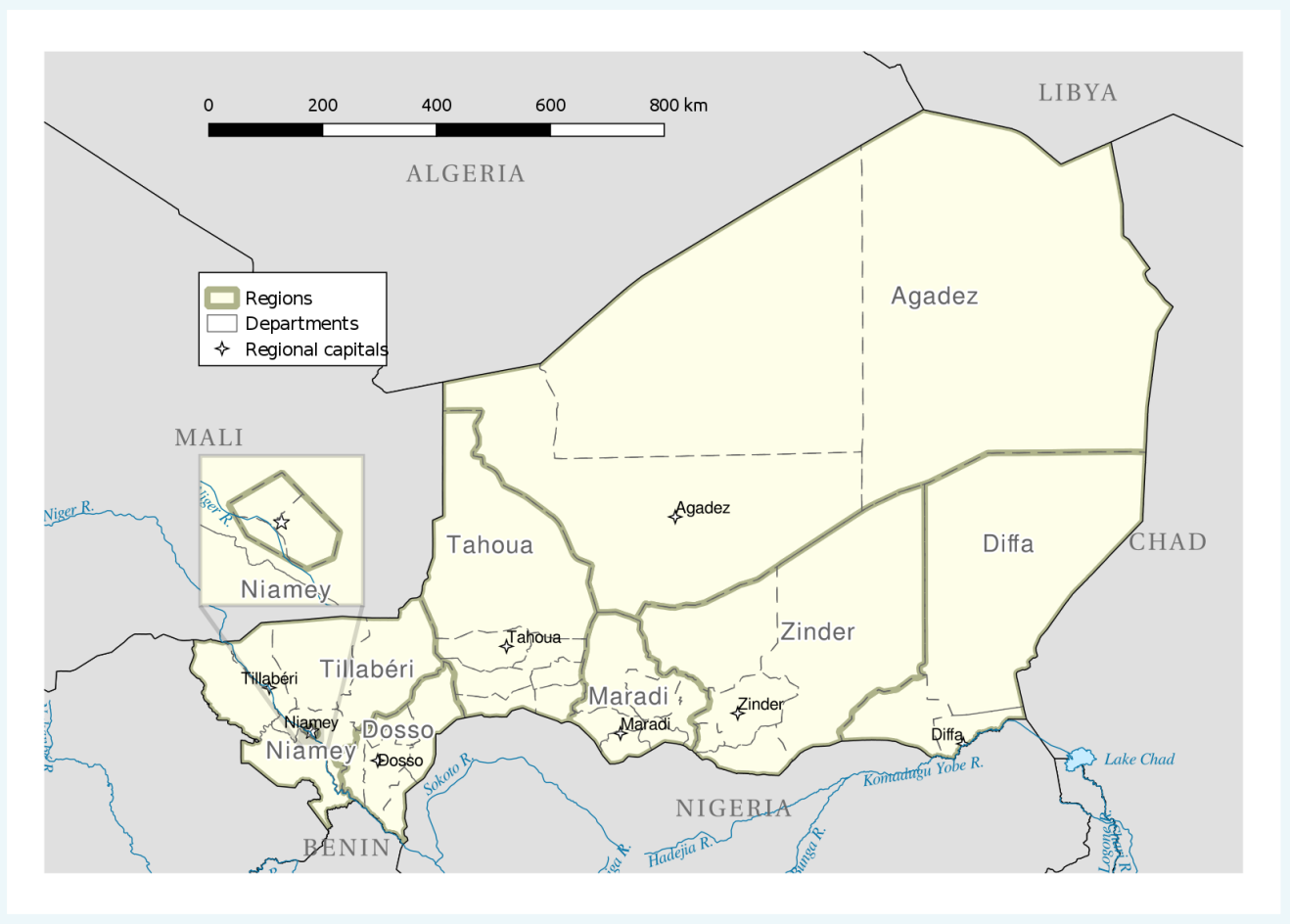


Figure 18.3: Administrative divisions of Niger: Districts and Regions

Several papers have investigated these trends, linking measles to human activity, migration, and seasonality. These studies are much more complex than what we will do there, but let’s see if we can find any patterns even with basic **exploratory data visualization**.

We can get some information about patterns in this data by inspecting summary statistics given by the `summary()` function:

```
summary(nigerm)
```

year	week	region	cases
Min. :1995	Min. : 1.00	Agadez : 572	Min. : 0.0
1st Qu.:1997	1st Qu.:13.75	Diffa : 572	1st Qu.: 1.0
Median :2000	Median :26.50	Dosso : 572	Median : 16.0
Mean :2000	Mean :26.50	Maradi : 572	Mean : 100.3
3rd Qu.:2003	3rd Qu.:39.25	Niamey : 572	3rd Qu.: 86.0
Max. :2005	Max. :52.00	Tahoua : 572	Max. :1887.0
		(Other):1144	

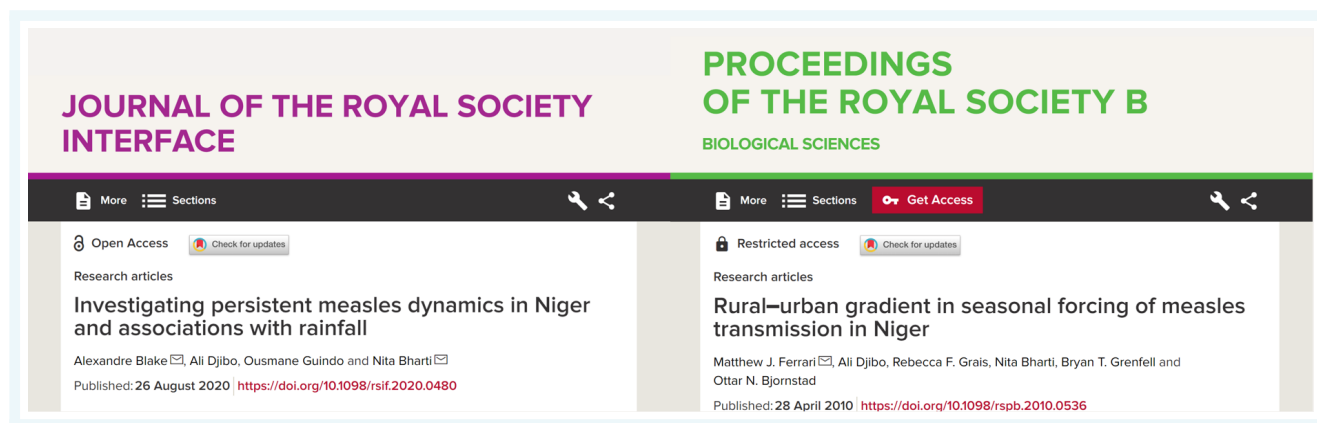


Figure 18.4: Research articles that have used this dataset, and analyzed it in R!

This gives us values for the maximum, minimum, and quartiles of each numeric variable, and the number of observations (rows) for each region. This is summary useful, but it omits a large amount information contained in the dataset.

Keep in mind that summary statistics can be highly misleading, and a simple plot can reveal a lot more.

The easiest and clearest way to analyze patterns from this dataset is to visualize it!

The best way to do this in R is with `{ggplot2}`. So let's see how that works.

18.4.2 The layered Grammar of Graphics

The `gg` in `ggplot` is short for “grammar of graphics”, which is the data visualization philosophy that `{ggplot2}` is based on.

The **grammar of graphics** is a theoretical framework which deconstructs the process of producing a graph.

Think of how we construct and form sentences in written and spoken languages by combining different elements, like nouns, verbs, articles, subjects, objects, etc. We can't just combine these elements in any arbitrary order; we must do so following a set of rules known as a linguistic grammar.

Similarly, the grammar of graphics (GG) defines a set of rules for constructing *graphics* by combining different types of elements, known as *layers*.

The three layers at the bottom of this figure - **data**, **aesthetics**, and **geometries** - are required for building any plot.

Let's define what they mean:

1. **data**: the dataset containing the variables of interest.

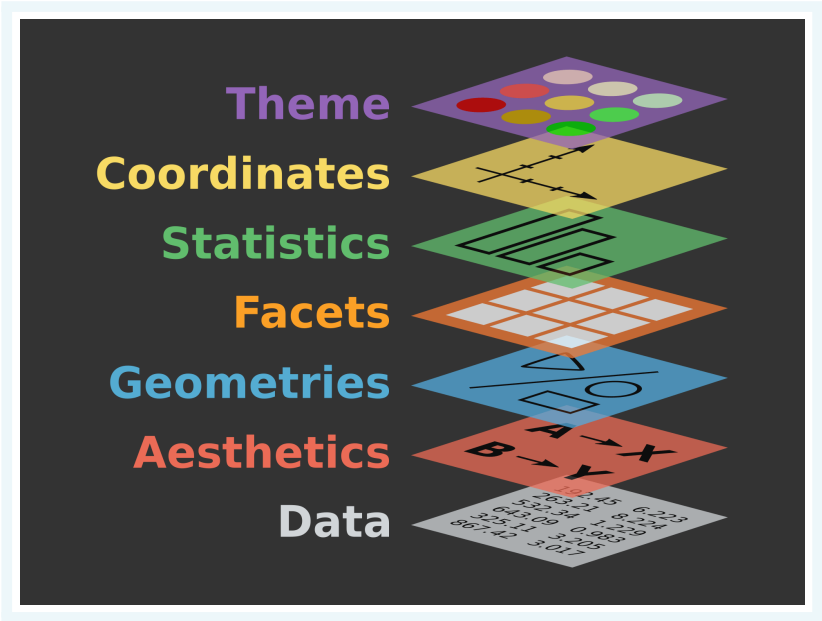
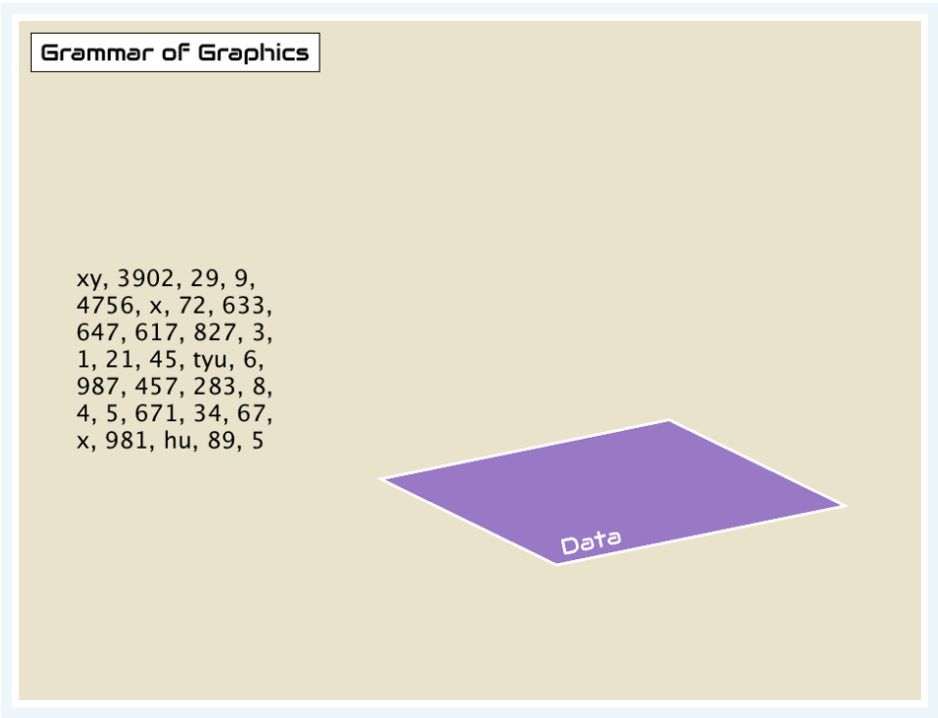
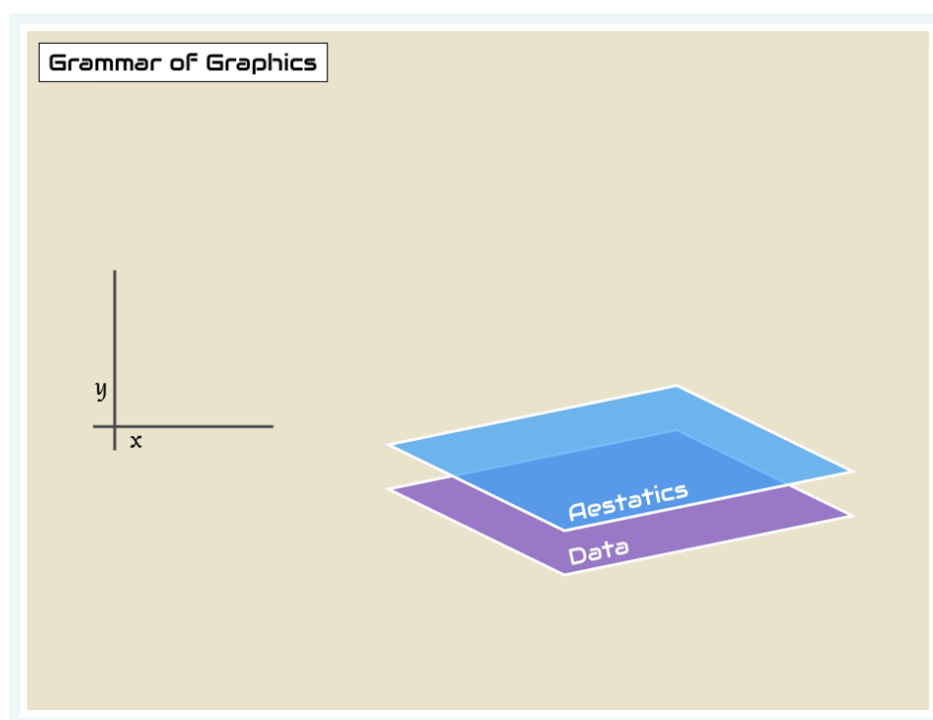


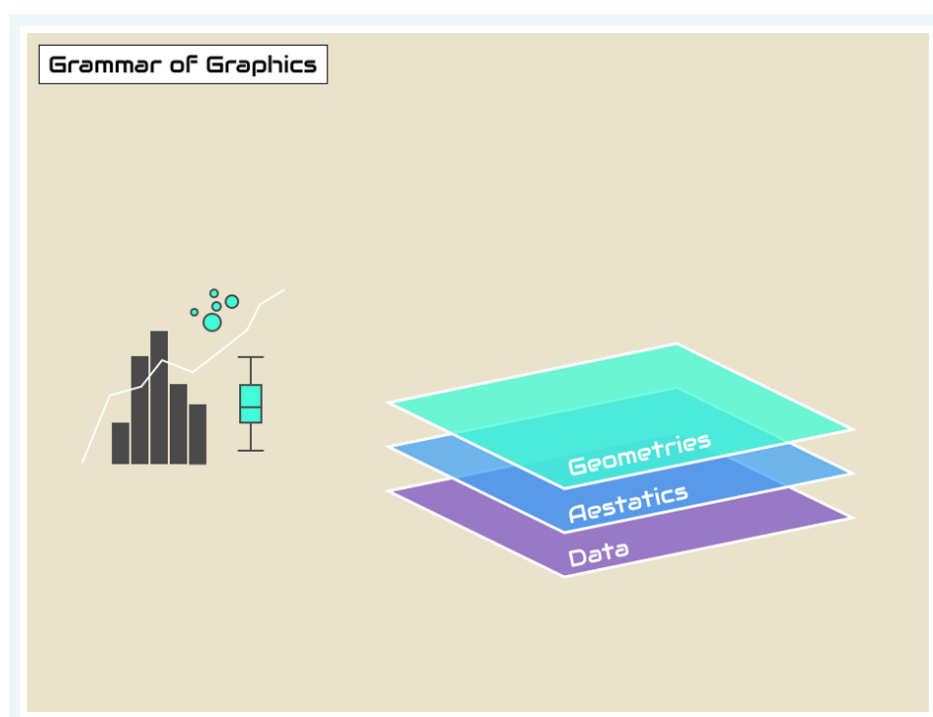
Figure 18.5: The grammar of graphics framework dissects a graph into individual components, which belong to these seven distinct layers. We take these different layers and combine them together to build a plot.



2. aesthetics: things we can see that visually communicate information in our data.



3. **geometry**: the geometric shape used to represent data in a plot: points, lines, bars, etc.



You might be wondering why we wrote `data`, `geom`, and `aes` in a computer code type font. You'll see very shortly that we use these terms in R code to represent GG layers.

i Challenge

The terms and syntax used for `ggplot` functions, arguments, and layers can be hard to keep up with at first, but as you gain experience using these terms to make plots in R, you will become fluent in no time.

18.5 Working through the essential layers

In this section, we will work towards a first plot with `{ggplot2}`. It will be a scatter plot using data from `nigerm`.

For easier plotting in this lesson, we will use a smaller subsets of the `nigerm` data frame at a time.

First let's create one called `nigerm96`, which only contains measles case data for the year 1996. Running the code below will create `nigerm96` and add it to your RStudio Environment:

```
## Create nigerm96 data frame
nigerm96 <- nigerm %>%
  filter(year == 1996) %>% # filter to only include rows from 1996
  select(-year) # remove the year column
```

Reminder

The `select()` and `filter()` functions are part of the `{dplyr}` package for data manipulation, which is a core package of the `{tidyverse}`. These topics are covered in the Data Wrangling course. See The GRAPH Courses [website](#) for more.

Let's look at our new dataframe, `nigerm96`:

```
## Print nigerm96
nigerm96
```

week	region	cases
1	Agadez	120
1	Diffa	75
1	Dosso	98
1	Maradi	123
1	Niamey	0

1–5 of 416 rows

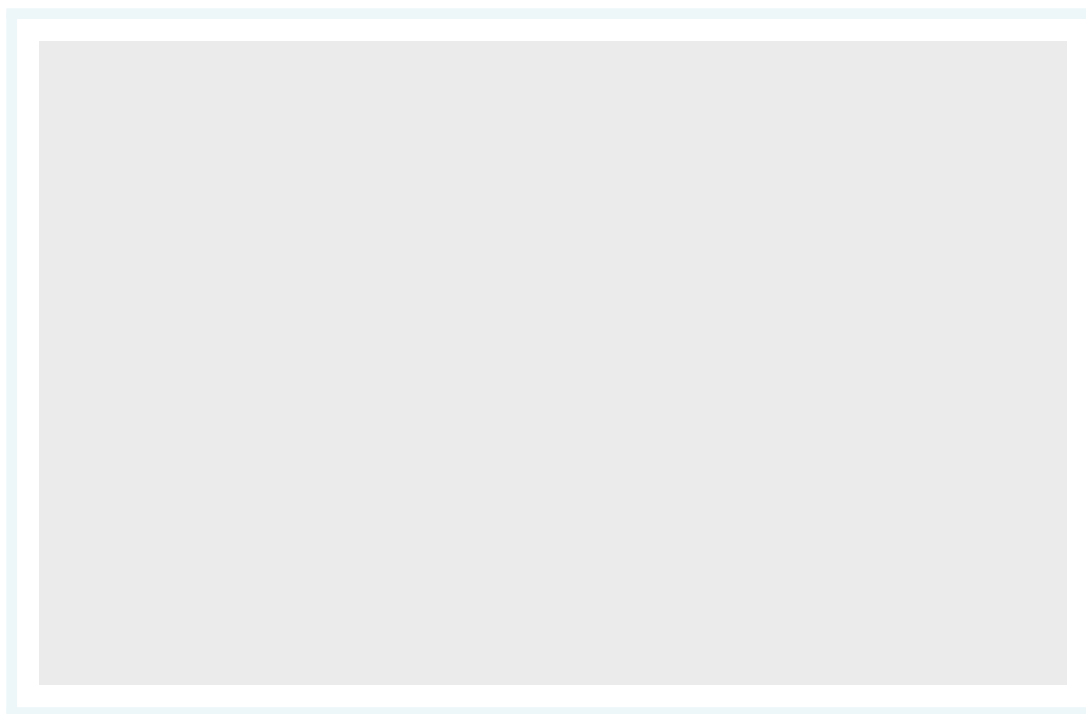
Previous **1** 2 3 4 5 ... 84 Next

18.5.1 Building a ggplot() in steps

Time to start building a ggplot in increments! We'll do this by starting with a blank canvas and then adding one layer at a time.

Step 0: Call the `ggplot()` function

```
## Call the `ggplot()` function  
ggplot()
```



As you can see, this gives us nothing but a blank canvas. But not to worry, we're about to add some more elements.

Step 1: Provide data

The first input we need to supply the `ggplot()` function is the data layer (i.e., a data frame), by filling in the data argument (`data = DF_NAME`):

```
## Data layer  
ggplot(data = nigerm96) # what data to use
```



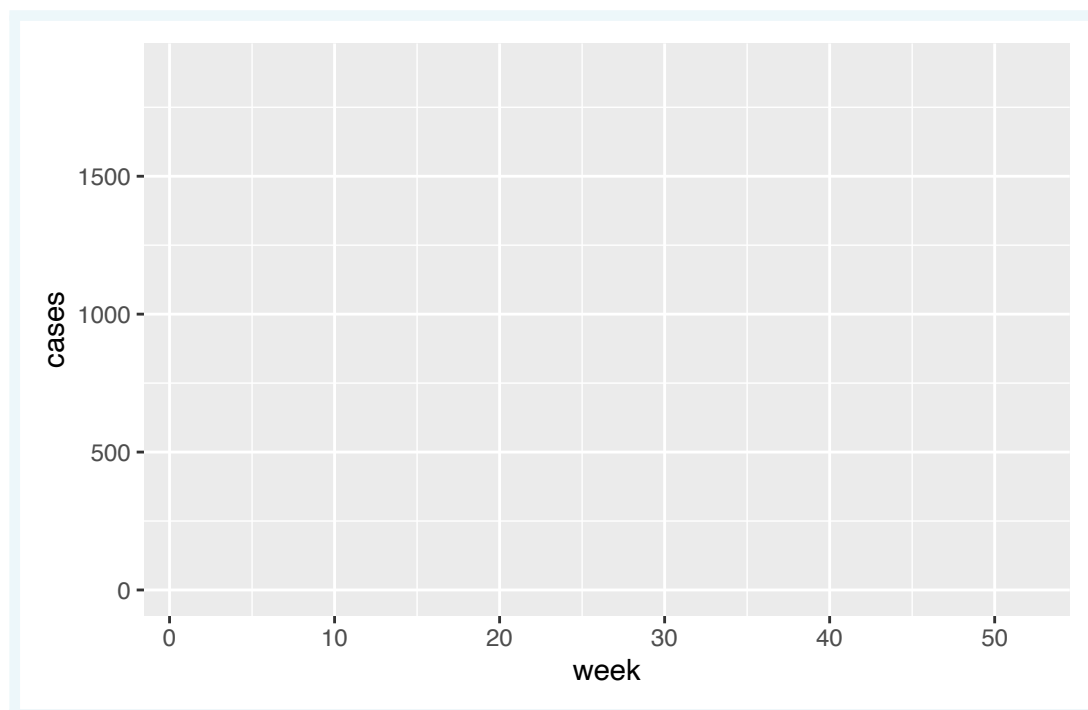
This gives us blank plot again, since we've only supplied one out of the three inputs required for a complete graphic. Next we need to assign variables to aesthetic mappings.

Step 2: Define the variables

What should we plot on our axes? Let's say we want to make an epidemic time series plot. To do that, we plot time (in weeks) on the x-axis, and disease incidence (number of reported cases) on the y-axis. In ggplot-speak, we are mapping the variable cases to the x aesthetic, and week to the y aesthetic.

Let's tell `ggplot()` which variables to plot on the aesthetics layer with a mapping argument, using this syntax: `mapping = aes(x = VAR1, y = VAR2)`.

```
## Aesthetics layer: x and y position
ggplot(data = nigerm96, # what data to use
        mapping = aes(  # supply a mapping in the form of an 'aesthetic'
            x = week,    # which variable to map onto the x-axis
            y = cases))  # which variable to map onto the y-axis
```



There's still no data plotted, but the axis scales, titles, and labels are present. The x-axis marks weeks of the year from 1 to 52, and the y-axis shows that the number of weekly reported cases per region ranges from 0 to around 2000.

The plot is still lacking the required geometry layer.

i Key Point

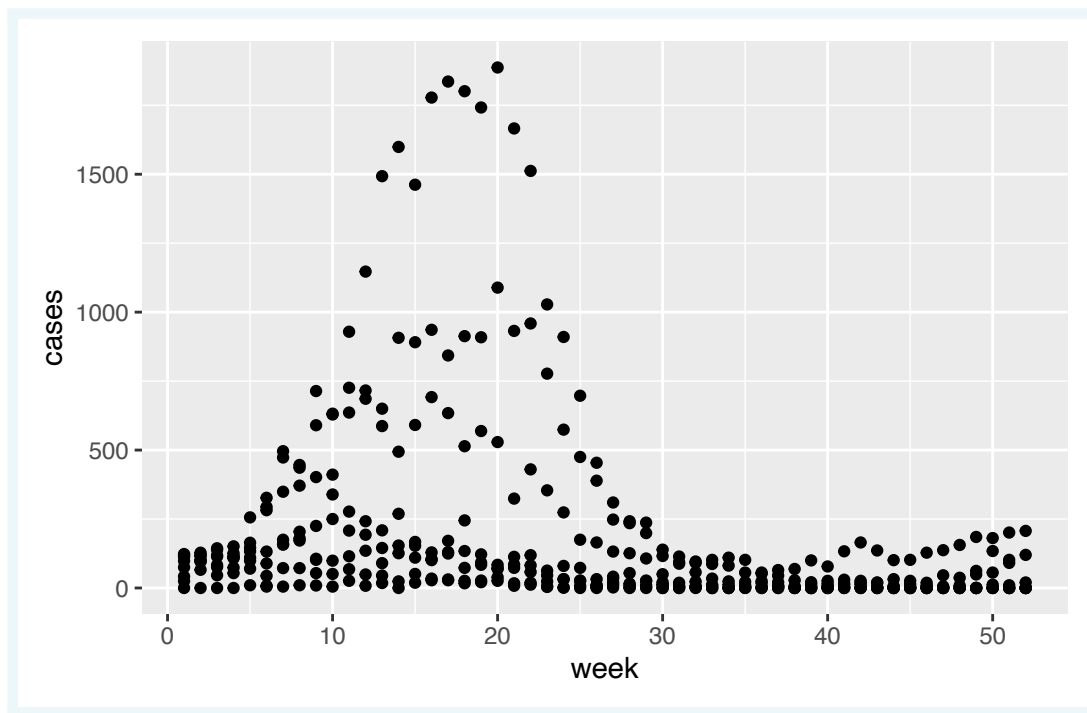
`aes()` stands for aesthetics - things we can see. Variables are always inside the `aes()` function, which in return is inside a `ggplot()`. Take a moment to observe the double closing brackets `)` - the first one belongs to `aes()`, the second one to `ggplot()`.

Step 3: Specify which type of plot to create

Finally, we add a geometry layer using a `geom_*` function. This determines which geometric objects - or visual markers - should be used to map the data.

Since we are looking at the relationship of two numerical variables, it makes sense to use a **scatter plot**. The geometric objects used to represent data on scatter plots are **points**, and the `geom_*` function for scatter plots is conveniently named `geom_point()`. We'll add this function as new layer using a `+` sign:

```
## Geometries layer: points
ggplot(data = nigerm96, # what data to use
       mapping = aes(   # define mapping
         x = week,      # which variable to map onto the x-axis
         y = cases)) +   # which variable to map onto the y-axis
  geom_point()           # add a geom of type `point` (for scatter plot)
```



Points have been added, and this is now a complete scatter plot! There are 8 points per week, representing each of the 8 regions (but at this point we cannot tell which point is from which region).

i Reminder

The `aesthetic` function is nested inside the `ggplot()` function, so be sure to close the brackets for both functions before adding the `+` sign for the `geom_*` function, or your code will not run correctly.

It's your turn to practice plotting with `ggplot()`! For practice exercises in this lesson, you will be using a different subset of `nigerm` called `nigerm04`, which contains only data from the year 2004:

week	region	cases
1	Agadez	11
1	Diffa	0
1	Dosso	7
1	Maradi	134
1	Niamey	60

1–5 of 416 rows

Previous12345...84Next

Plotting with a different set of data will also allow you to explore if the patterns we see for 1996 is also true for 2004.

Practice

Using the `nigerm04` data frame, write `ggplot` code that will create a scatter plot displaying the relationship between cases on the y-axis and week on the x-axis.

18.6 Modifying the layers

Generally speaking, the grammar of graphics allows for a high degree of customization of plots and also a consistent framework for easily updating and modifying them.

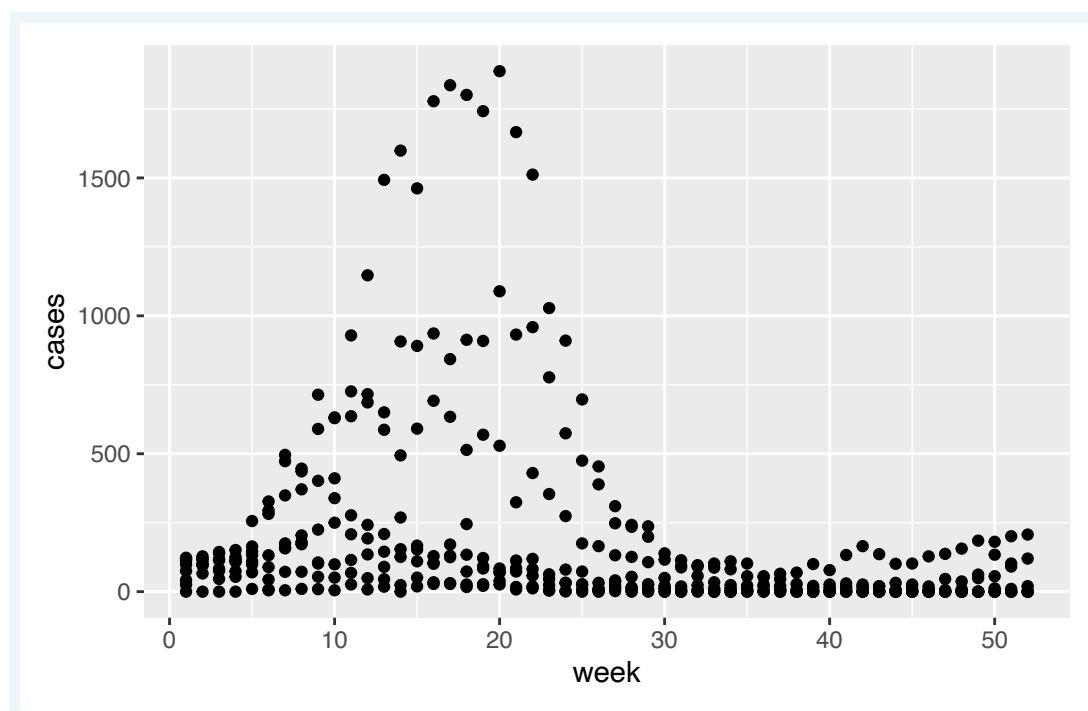
We can tinker with our existing code to switch up the data, aesthetics, and geometry inputs supplied to `ggplot()`, and create variations of the original plot. In fact, you've already done this by changing the dataset from `nigerm96` to `nigerm04` in the practice question.

Similarly, the aesthetics and geometry inputs can also be changed to create different visualizations. In the next few sections we will take the scatter plot we built in the previous section, and make incremental changes to modify different elements of the original code.

18.6.1 Changing aesthetic mappings

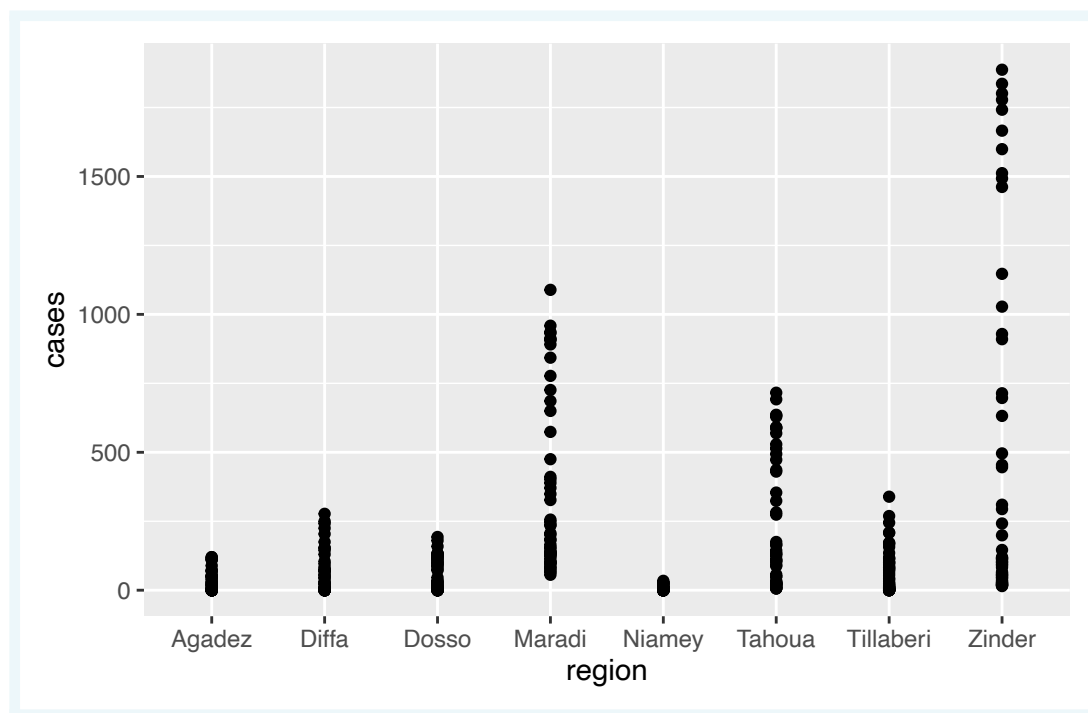
We created a scatter plot of cases vs week for `nigerm96` with this code:

```
ggplot(data = nigerm96,  
       mapping = aes(x = week,  
                     y = cases)) +  
  geom_point()
```



If we copy the same code and change just one thing - by replacing the x variable `week` (numerical) with `region` (categorical) - we get what's called a **strip plot**:

```
ggplot(data = nigerm96,
       mapping = aes(x = region, # change which variable to map on the x-axis
                     y = cases)) +
  geom_point()
```



While the y-axis values of the points are the same as before, their x-axis mappings have changed significantly. They are now mapped to 8 separate positions along the x-axis, each corresponding to a discrete category of the region variable.

18.6.2 Changing geom_* functions

Similarly, we can modify the geometry layer to create a different type of plot, while still using the same aesthetic mappings.

Let's copy and paste the original scatter plot code once again, but this time we will replace the geom_* function instead of the x aesthetic. If we change geom_point() to geom_col(), we get a **bar plot** (sometimes called a column chart):

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases)) +
  geom_col() # declare that we want a bar plot
```

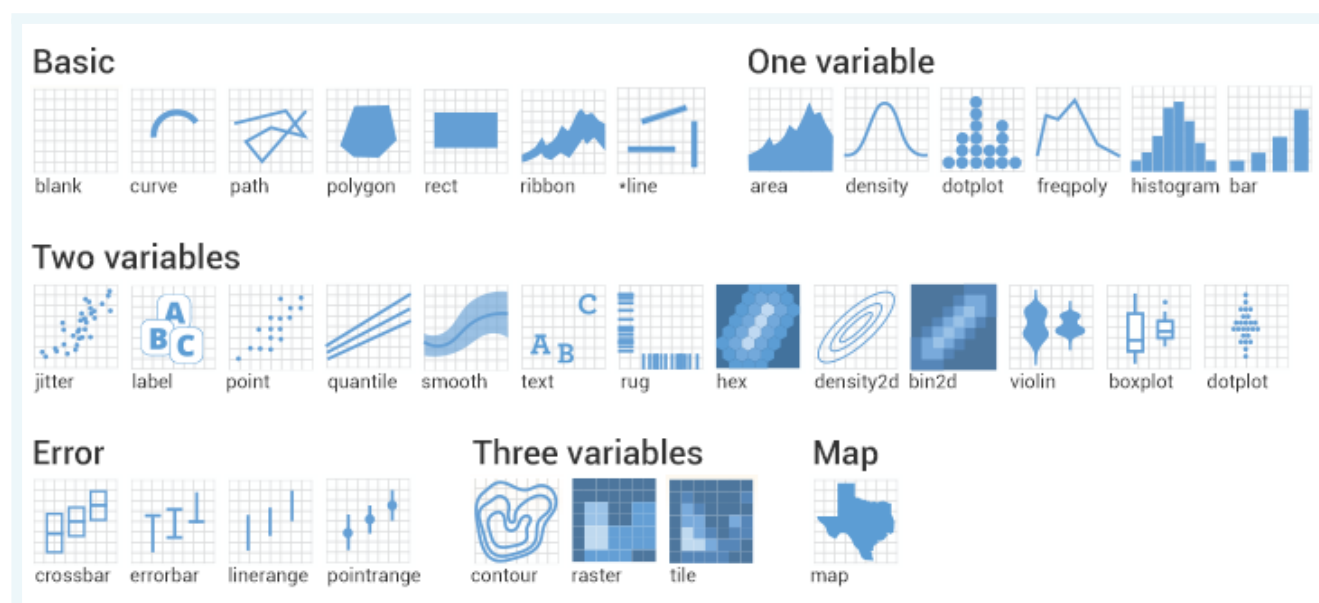
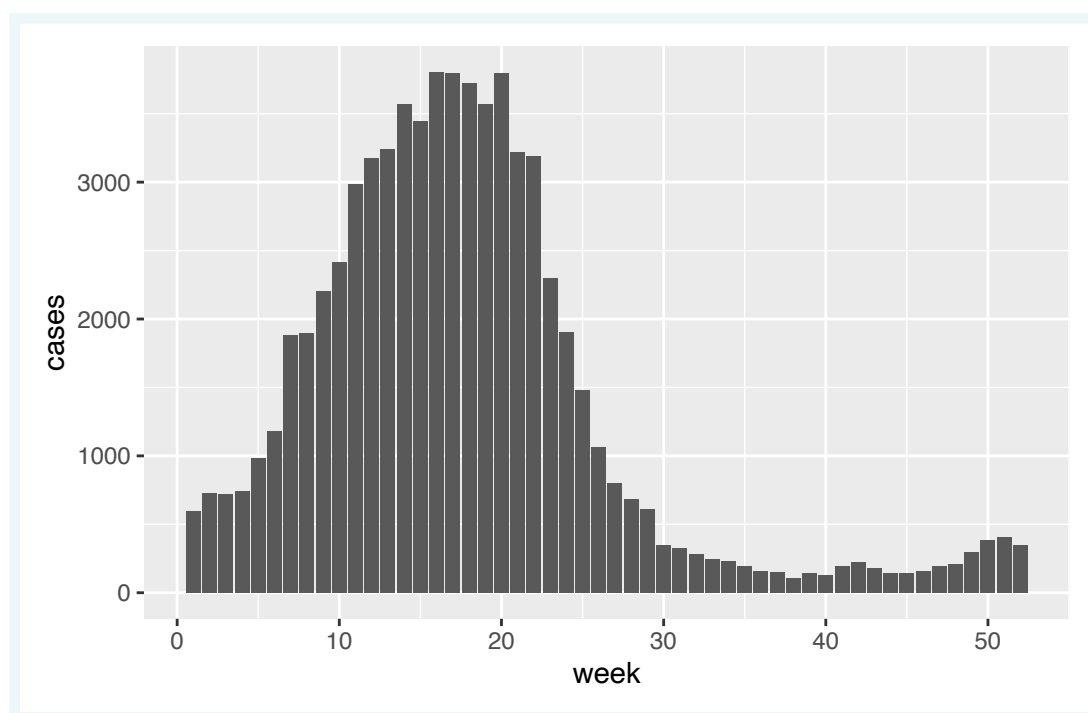


Figure 18.6: `{ggplot2}` has a variety of different `geom_*` functions and geometric objects which you can use to visualize your data. Here are some examples of different types of geoms that can be used with `ggplot()`.



Again, the rest of the code is still the same - we just changed the key word of the `geom_*` function. However, the plot is significantly different that either the scatter plot or the strip plot.

Notice that the y-axis has been rescaled. The height of each bar represents the cumulative number of weekly cases, i.e, the total number of cases reported from all eight regions that week, rather than showing 8 separate data points for each region.

Caution

Not all plot types are interchangeable. Using a `geom_*` function that is not compatible with the variables you defined in `aes()` will give you an error. For example, let's replace `geom_point()` with `geom_histogram()` instead:

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases)) +
  geom_histogram()
```

This is because a histogram shows the distribution of one numerical variable. `ggplot()` can't map two variables to both the x and y-axis positions with a histogram, so it throws an error.

Practice

Use the `nigerm04` data frame to create a bar plot of weekly cases with the `geom_col()` function. Map cases on the y-axis and week on the x-axis.

18.6.3 Additional aesthetic mappings inside `aes()`

So far, we have only mapped variables to the x and y aesthetic attributes. We can also map variables to other aesthetics like color, size, or shape.

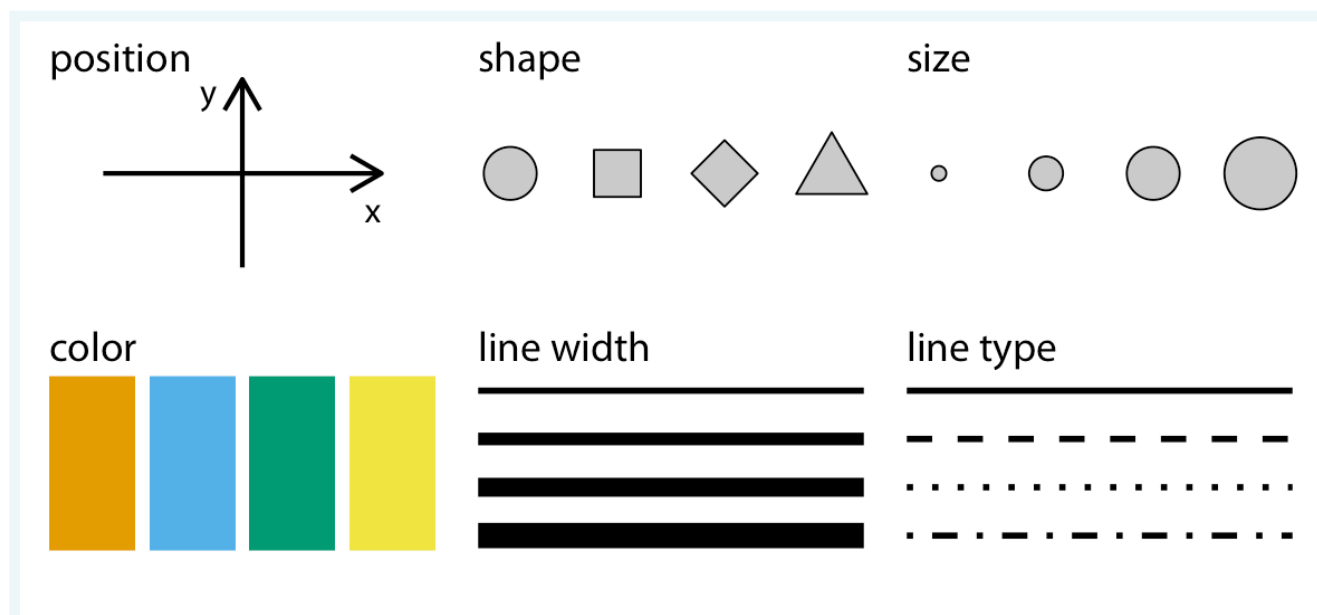
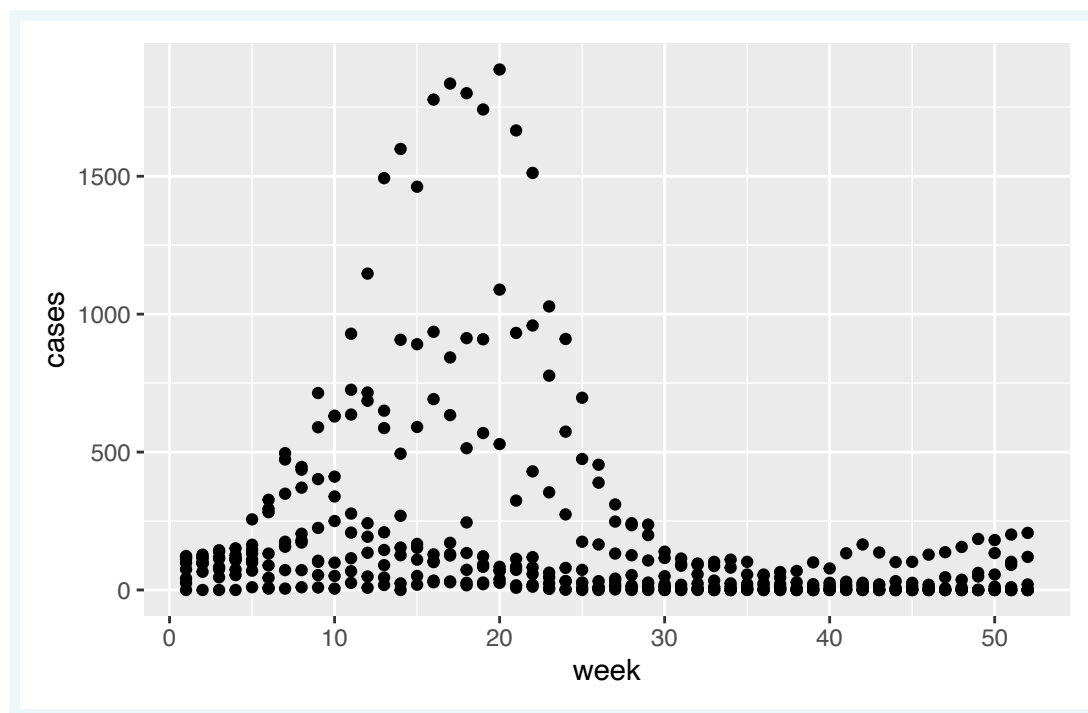


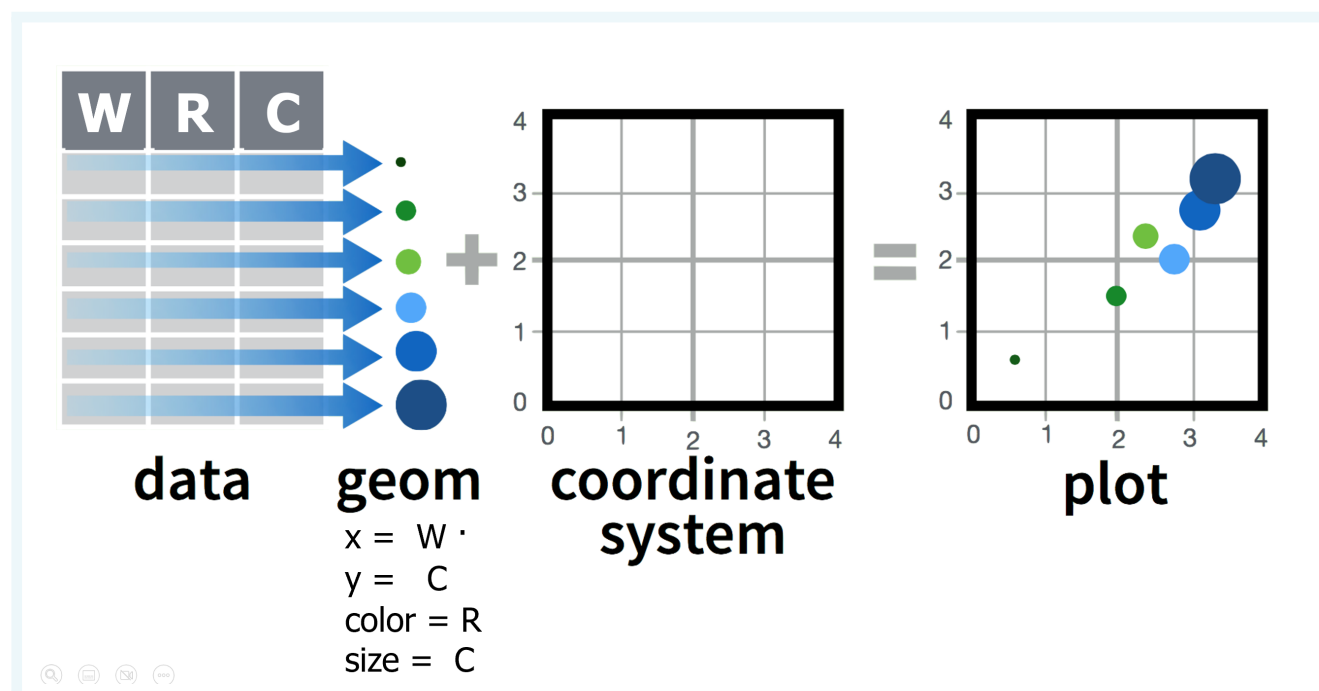
Figure 18.7: Common aesthetic attributes used in `ggplot` graphics.

Let's return to our original scatter plot (cases vs week):

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases)) +
  geom_point()
```



There are other aesthetics we can add, like color or size.

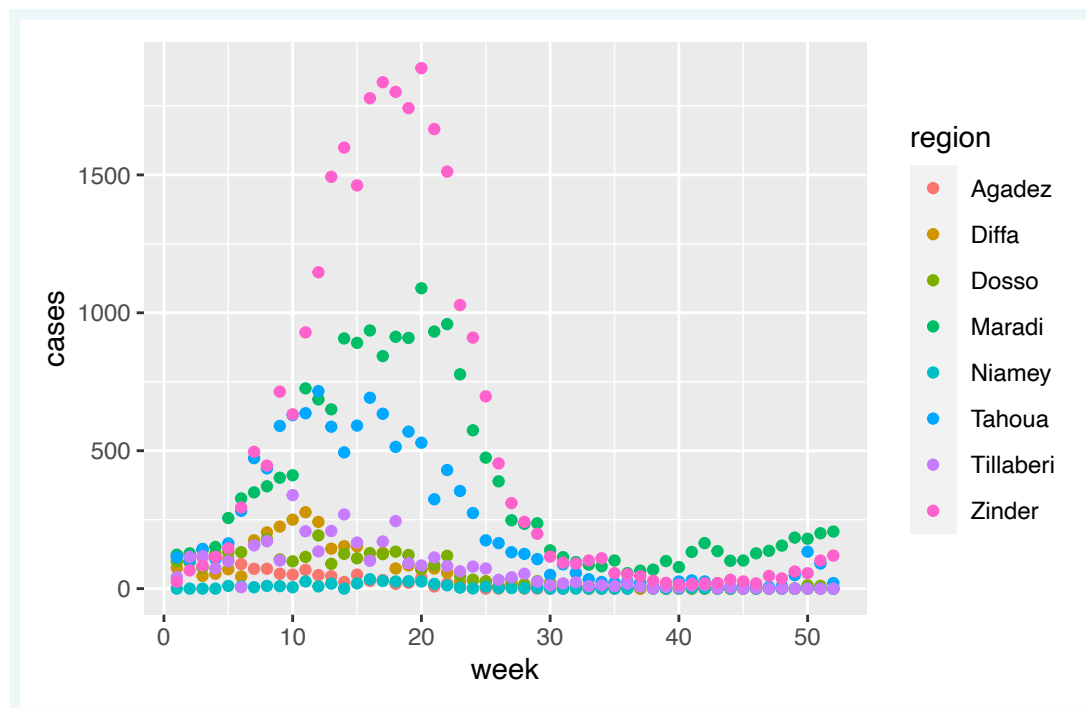


i Pro Tip

To see the full list of aesthetics that can be used with a particular `geom_*` function look it up the function documentation. You can do this by pressing F1 on a function, e.g., `geom_point()` to open the Help tab, and scroll down to the “Aesthetics” section. If F1 is hard to summon on your keyboard, type and run `?geom_point` in your Console tab.

Let’s add color to our scatter plot. We can map the categorical variable `region` to the `color` aesthetic. We can do this by modifying the original code to add a new argument inside `mapping = aes()`. Let’s see what happens when we add `color = region` inside `aes()`:

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases,
                     color = region)) + # use a different color for each region
geom_point()
```



Now we have a colorful scatter plot! Each point is colored according to the region it belongs to. This allows us to better distinguish between regions.

Note that `ggplot()` automatically provides a color legend on the left.

Side Note

The colors are from `{ggplot2}`'s default rainbow color palette. In later lessons we will learn how to customize color scales and palettes, including making figures colorblind-friendly.

By examining the color patterns in the plot, you can make out the classic bell-shaped epidemic curves showing a rise and fall in measles incidence in each region.

Zinder had the largest number of cases and the steepest epidemic curve, followed by Maradi and Niamey.

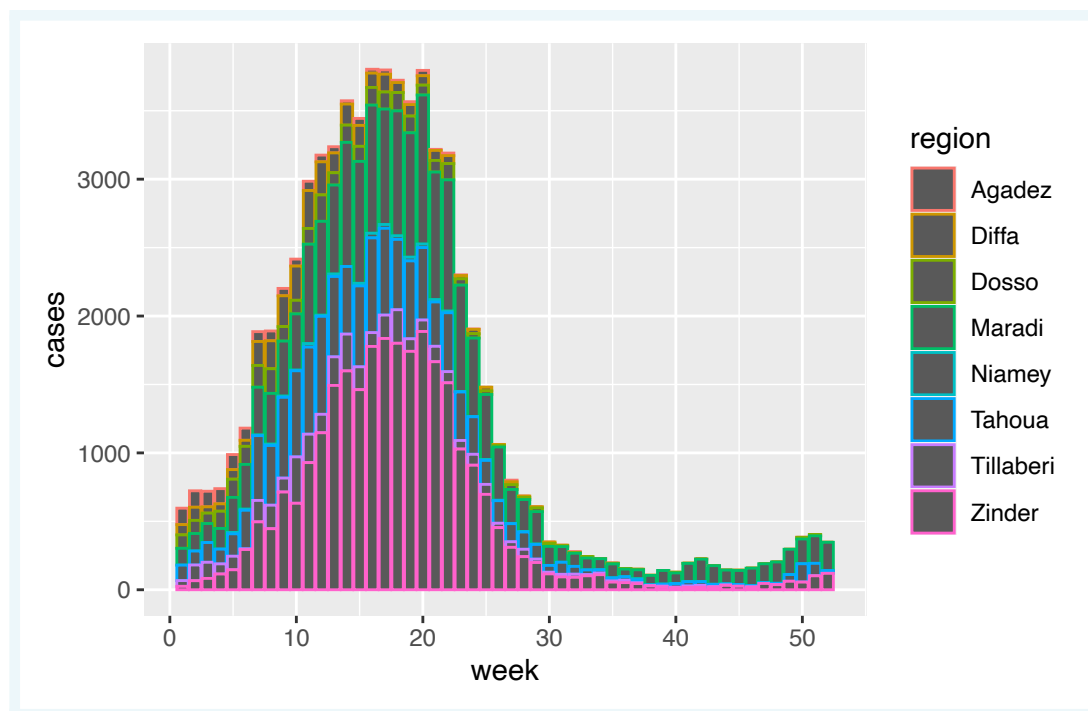
While the colorful plot provides more insight into measles patterns at the regional level than the scatter plot with no color mapping, this graph still looks busy and is not the most intuitive to read. A different plot type could help with this.

Next we will try a bar plot, then a line graph.

Let's try the same `color = region` aesthetic mapping with `geom_col()` instead:

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases,
                     color = region)) + # use a different outline color for each region
```

```
geom_col()
```

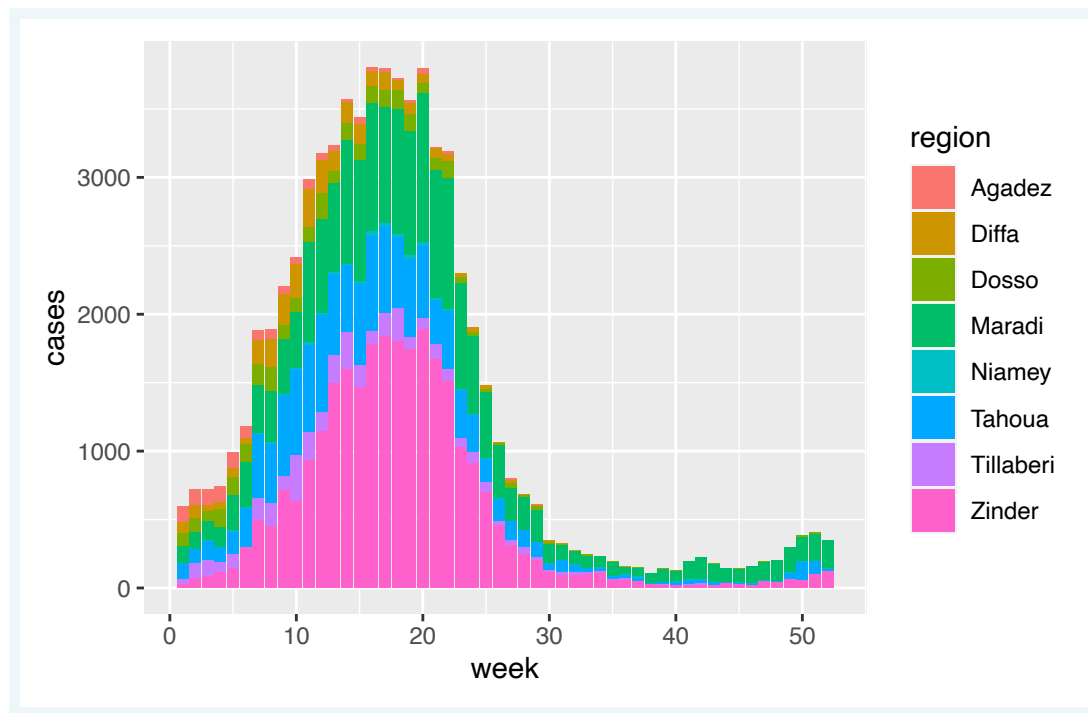


This gives us a stacked bar plot, where the bars are divided into smaller sections. This shows us the proportional contribution of individual regions (i.e., the height or length of each subsection represents how much each region contributes to the total number of cases that week).

The stacked bar plot here is outlined by color. This is because the `color` aesthetic in `{ggplot2}` generally refers to the border around a shape. This did not apply to the default shapes in our scatter plot created with `geom_point()` because they are solid dots (not hollow), but you can see that it does apply to the bars in a bar chart created `geom_col()`. However, the grey filling is not very pretty.

We might want to color the inside of the bars instead. This is done by mapping our variable to the `fill` aesthetic. We can copy the code above and simply change `color` to `fill` inside `aes()`:

```
ggplot(data = niger96,
       mapping = aes(x = week,
                     y = cases,
                     fill = region)) + # use a different fill color for each region
  geom_col()
```



Voila! The inside of the bars are now filled with colors.

Now practice using the `color` aesthetic mapping with a new plot type: line graphs. Line graphs are generally considered one of the best plot types for time series data.

💡 Practice

Use the `nigerm04` data frame to create a line graph of weekly cases, colored by region. Map cases on the y-axis, week on the x-axis, and region to color. The `geom_*` function for a line graph is called `geom_line()`.

18.6.4 Fixed aesthetics outside `aes()`

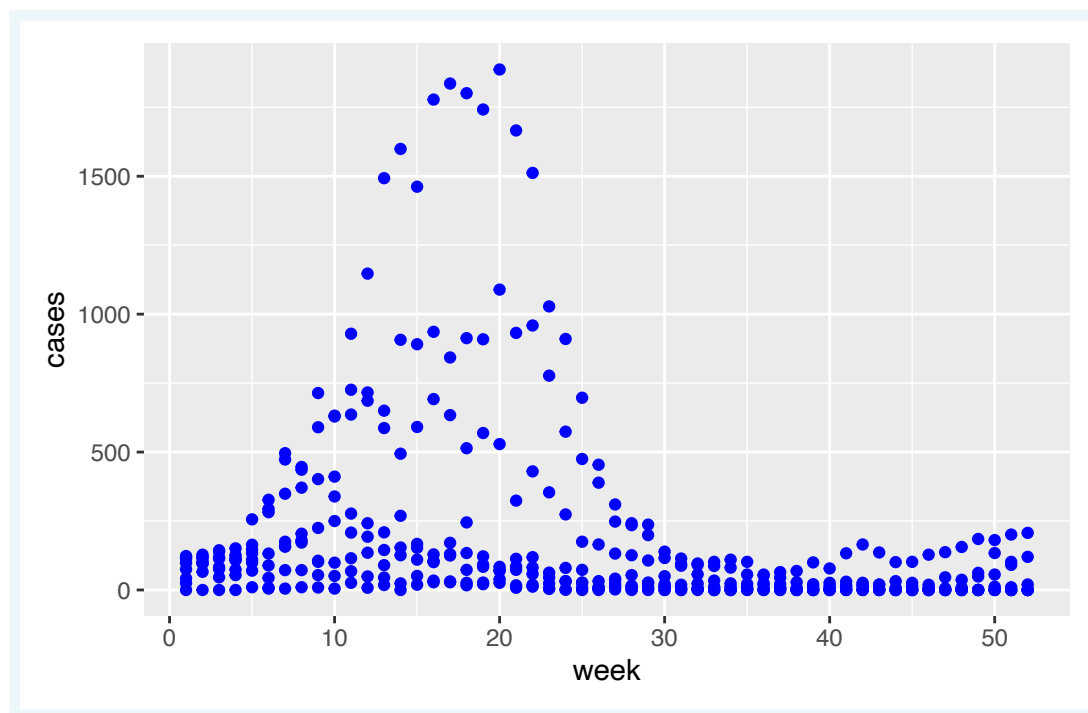
It is very important to understand the difference between **aesthetic mappings** and **fixed aesthetics**. The main aesthetics in `ggplot` are: `x`, `y`, `color`, `fill`, and `size`, and any of these could be either a mapping or a fixed value. This depends on whether they appear inside or outside the `aes()` function.

When we apply an aesthetic to modify the geometric objects according to a variable (e.g., the color of points changes according to the region variable), that's an aesthetic mapping. This must always be defined **inside** `mapping = aes()`, like we just did in previous examples.

But if you want to apply a visual modification to *all* the geometric objects evenly (e.g., manually change the color of all points to be one color), that's a fixed aesthetic. We must set fixed aesthetics to a constant value **outside** `mapping = aes()` and directly inside the `geom_*` function - e.g., `geom_point(color = "COLOR_NAME")`.

Here let's change the color of all the points in our scatter plot to blue:

```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases)) +
  geom_point(color = "blue") # use the same color for all points
```



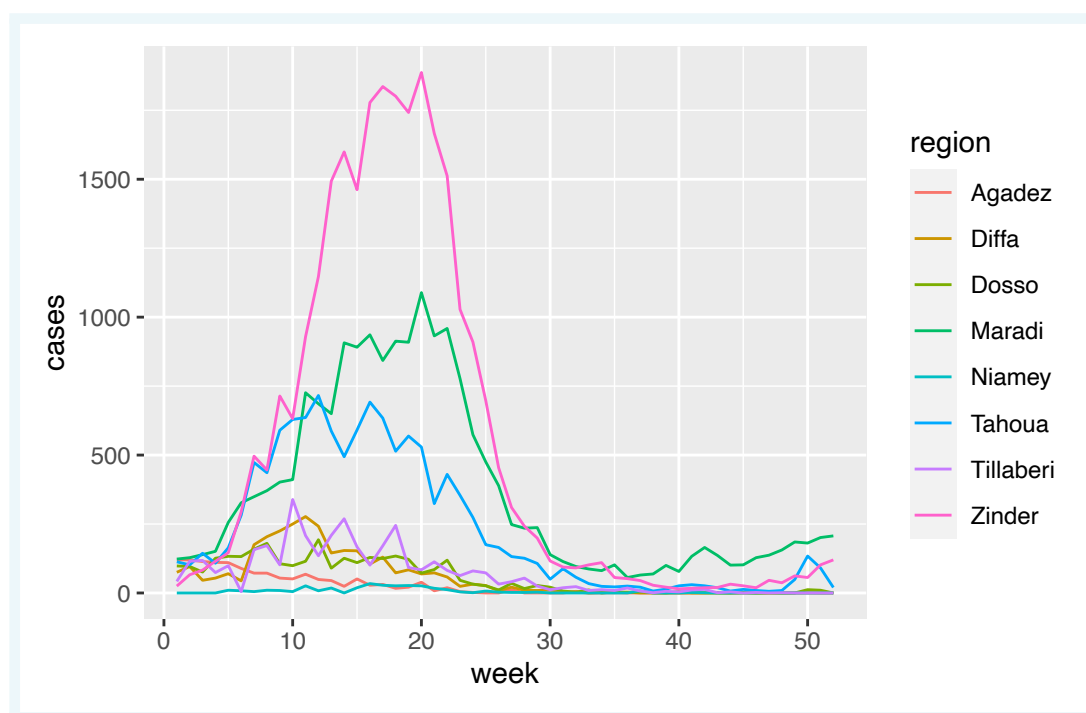
This colors each point with the same R color ("blue"). In this plot, the color aesthetic does not represent any values from the data frame. Note that the color names in R are character strings, so it needs to go inside quotation marks.

i Side Note

If you're curious, run `colors()` in your console to see all possible choice of colors in R! To find out exactly how many options that is, try running `colors() %>% length()`.

Now let's add a fixed aesthetic called `size`. The default line width used by `geom_line()` is 0.5 mm, which looks like this:

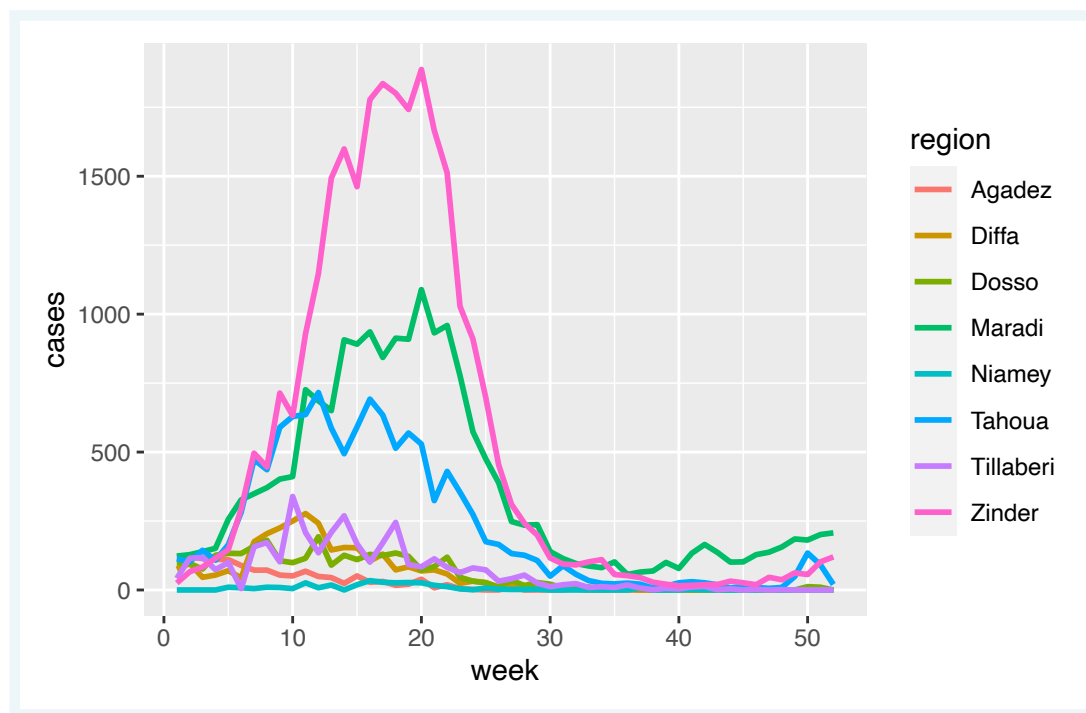
```
ggplot(data = nigerm96,
       mapping = aes(x = week,
                     y = cases,
                     color = region)) +
  geom_line()
```



To make all of the lines in our figure a little thicker, let's fix this aesthetic at 1 mm. We do this by adding `size = 1` inside the `geom_line()` function:

```
ggplot(data = niger96,
       mapping = aes(x = week,
                     y = cases,
                     color = region)) +
  geom_line(size = 1)
```

Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
Please use `linewidth` instead.

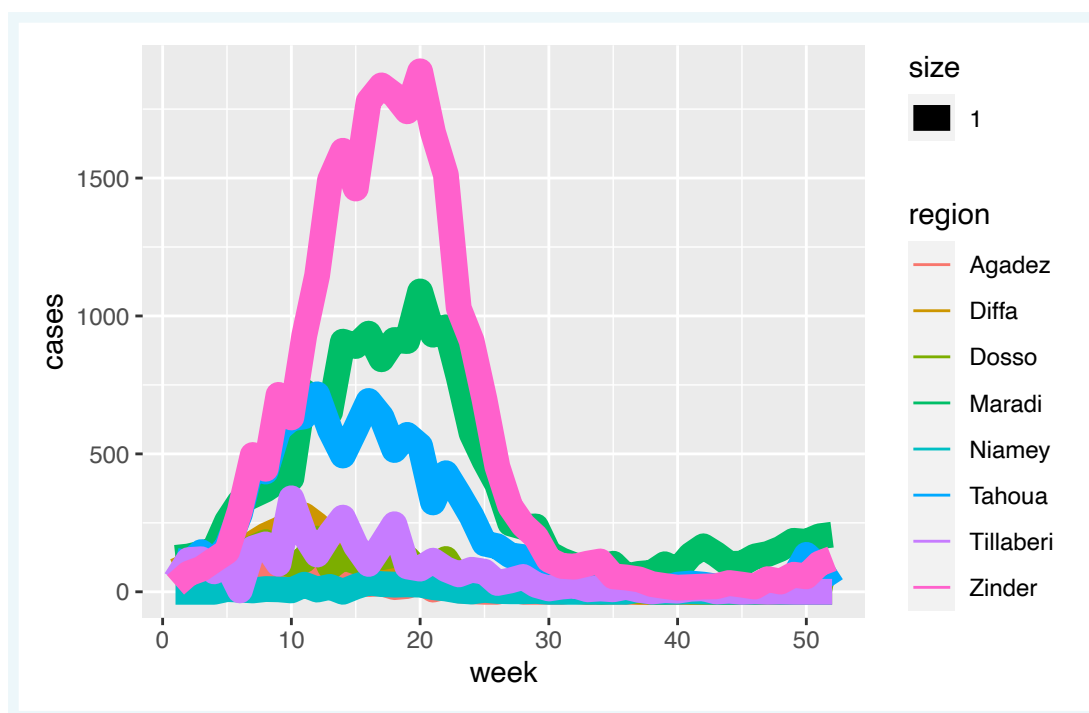


All the lines in the plot have been made thicker, and the line width is set to a constant value of 1 mm. Note that here the value of `size` is numeric, so it should not be in quotation marks.

🔥 Watch Out

Remember that fixed aesthetics are manually set to constant value (as opposed to a variable from the data), and goes directly in the `geom_*` function, **not** inside `aes()`. If you try to put a fixed aesthetic in `aes()`, you might get a weird result. For example, let's try moving the `size = 1` aesthetic from `geom_line()` to `aes()` to see how it can go wrong:

```
ggplot(data = niger96,
       mapping = aes(x = week,
                     y = cases,
                     color = region,
                     size = 1)) +      # INCORRECT placement
  geom_line()
```



`aes()` is a mapping function that modifies plots based on variables from the data. Since there is no variable called "1" in the `nigerm96` data frame, `aes()` cannot process or map this aesthetic correctly.

Practice using `fill` as a fixed aesthetic for a bar plot.

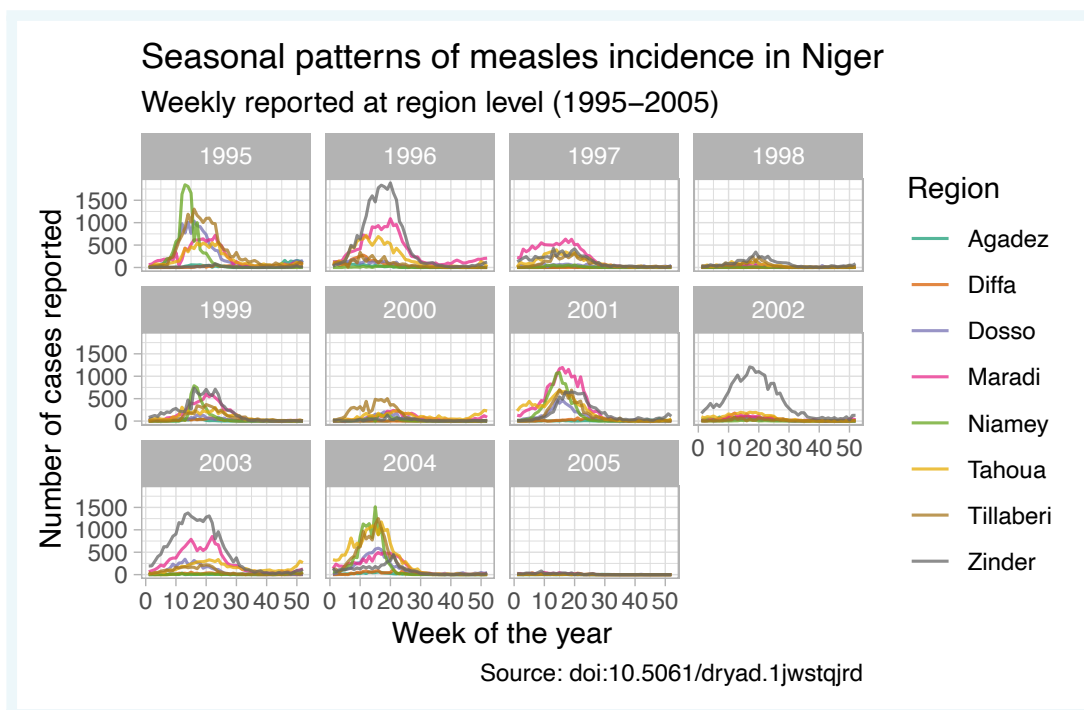
💡 Practice

Use the `nigerm04` data frame to create a bar graph of weekly cases, and fill all bars with the same color. Map cases on the y-axis, week on the x-axis, and fix the `color` aesthetic of the bars to the R color "hotpink".

18.7 Additional GG layers

In this lesson, we kept things simple and only worked with the three required layers. As you start to delve deeper into plotting with `{ggplot2}`, you'll start to encounter the other layers more frequently.

Soon you'll be able to create more complex plots, like this one:



i Recap

To build a complete `ggplot`, you must first supply a data frame using the `data` argument of `ggplot()`, and define variables and map them to aesthetics inside `aes()` using the `mapping` argument of `ggplot()`. Then start a new layer with a `+` sign and specify the type of plot you want using an appropriate `geom_*` function. You can copy this code template and adapt it to create different `ggplot` graphics:

```
ggplot(data = DF_NAME,
       mapping = aes(AES1 = VAR1,
                     AES2 = VAR2,
                     AES3 = VAR3,
                     ...)) +
  geom_FUCNTION()
```

18.8 Learning outcomes

1. You can recall and explain how the **{ggplot2}** package for data visualization is based on a theoretical framework called the **grammar of graphics**.
2. You can name and describe the 3 essential layers for building a graph: **data**, **aesthetics**, and **geometries**.
3. You can write code to **build a complete ggplot graphic** by correctly supplying the 3 essential layers to the `ggplot()` **function**.
4. You can create different types of plots such as **scatter plots**, **line graphs**, and **bar graphs**.
5. You can add or modify aesthetics of a plot such as the **color**, and **size**.

References

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

- Blake, Alexandre, Ali Djibo, Ousmane Guindo, and Nita Bharti. 2020. "Investigating Persistent Measles Dynamics in Niger and Associations with Rainfall." *Journal of The Royal Society Interface* 17 (169): 20200480. <https://doi.org/10.1098/rsif.2020.0480>.
- Cmprince. *Administrative divisions of Niger: Departments and Regions*. 29 October 2017. Wikimedia Commons. Accessed October 14, 2022. https://commons.wikimedia.org/wiki/File:Niger_administrative_divisions.svg
- DeBruine, Lisa, and Dale Barr. 2022. *Chapter 3 Data Visualisation | Data Skills for Reproducible Research*. <https://psyteachr.github.io/reprores-v3/ggplot.html>.
- Franke, Michael. n.d. *6 Data Visualization | An Introduction to Data Analysis*. Accessed October 12, 2022. <https://michael-franke.github.io/intro-data-analysis/Chap-02-02-visualization.html>.
- Geography Now, dir. 2019. *Geography Now! NIGER*. <https://www.youtube.com/watch?v=AHeq99pojLo>.
- Giroux-Bougard, Xavier, Maxwell Farrell, Amanda Winegardner, Étienne Low-Decarie and Monica Grana-dos. 2020. *Workshop 3: Introduction to Data Visualisation with Ggplot2*. <http://r.qcbs.ca/workshop03/book-en/>.
- Ismay, Chester, and Albert Y. Kim. 2022. *A ModernDive into R and the Tidyverse*. <https://moderndive.com/>.
- Kabacoff, Rob. 2020. *Data Visualization with R*. <https://rkabacoff.github.io/datavis/>.
- Lisa DeBruine. 2020. *Basic Plots*. <https://www.youtube.com/watch?v=tOFQFPRgZ3M>.
- Pius, Ewen Harrison and Riinu. n.d. *R for Health Data Science*. Accessed October 11, 2022. https://argoshare.is.ed.ac.uk/healthyr_book/.
- Prabhakaran, Selva. 2016. "How to Make Any Plot in Ggplot2? | Ggplot2 Tutorial." 2016. <http://r-statistics.co/ggplot2-Tutorial-With-R.html>.

18.9 Solutions

```
.SOLUTION_nigerm04_scatter()
```

```
ggplot(data = nigerm04,
       mapping = aes(x = week,
                     y = cases)) +
  geom_point()
```

```
.SOLUTION_nigerm04_bar()
```

```
ggplot(data = nigerm04,
       mapping = aes(x = week,
                     y = cases)) +
  geom_col()
```

```
.SOLUTION_nigerm04_line()
```

```
ggplot(data = niger04,  
       mapping = aes(x = week,  
                     y = cases,  
                     color = region)) +  
  geom_line()
```

```
.SOLUTION_niger04_pinkbar()
```

```
ggplot(data = niger04,  
       mapping = aes(x = week,  
                     y = cases)) +  
  geom_col(fill = "hotpink")
```

Chapter 19

Scatter plots and smoothing lines

19.1 Introduction

Scatter plots - which are sometimes called **bivariate plots** - allow you to visualize the **relationship** between two numerical variables.

They are among the most commonly used plots because they can provide an immediate way to see how one numerical variable varies against another.

Scatter plots can also display multiple relationships by mapping additional variable to aesthetic properties, such as color of the points.

Trends and relationships in a scatter plot can be made clearer by adding a smoothing line over the points.

We will use ggplot to do all that and more. Let's get started!

19.2 Learning Objectives

1. You can visualize relationships between numerical variables using **scatter plots** with `geom_point()`.
2. You can use `color` as an aesthetic argument to map variables from the dataset onto individual points.
3. You can change the size, shape, color, fill, and opacity of geometric objects by setting **fixed aesthetics**.
4. You can add a **trend line** to a scatter plot with `geom_smooth()`.

19.3 Childhood diarrheal diseases in Mali

We will be using data collected for a prospective observational study of acute **diarrhea in children** aged 0-59 months. The study was conducted in Mali and in early 2020.

The full dataset can be obtained from [Dryad](#), and the paper can be viewed [here](#).

Vocab

A prospective study watches for outcomes, such as the development of a disease, during the study period and relates this to other factors such as suspected risk or protection factors.

Spend some time browsing through this dataset. Each row corresponds to one patient surveyed. There are demographic, physiological, clinical, socioeconomic, and geographic variables.

n	admit_date	sex	age_months	height_cm	muac_cm	breastfeed...	vomit	fever	bloody_stool
1	2020-01-16	M	5	61.2	11.3	1	0	0	0
2	2020-01-17	F	12	70.6	13.2	1	1	1	0
3	2020-01-17	M	11	71.1	13.5	1	1	0	0
4	2020-01-17	M	9	68.5	12.6	1	0	0	0
5	2020-01-21	F	16	78.7	14.2	1	1	0	0
6	2020-01-21	M	6	67.7	14.5	1	0	0	0
7	2020-01-22	M	5	64.7	14.1	1	0	0	0
8	2020-01-22	M	46	98.1	14.4	0	1	0	0
9	2020-01-17	M	4	61.2	13.1	1	0	1	0
10	2020-01-21	M	25	85.7	15	0	1	1	0

1–10 of 150 rows

Previous 1 2 3 4 5 ... 15 Next

We will begin by visualizing the relationship between the following two numerical variables:

1. `age_months`: the patient's **age** in months on the horizontal **x**-axis and
2. `viral_load`: the patient's **viral load** on the vertical **y**-axis

19.4 Scatter plots via `geom_point()`

We will explore relationships between some numerical variables in the `malidd` data frame.

We will now examine at and run the code that will create the desired scatter plot, while keeping in mind the GG framework. Let's take a look at the code and break it down piece-by-piece.

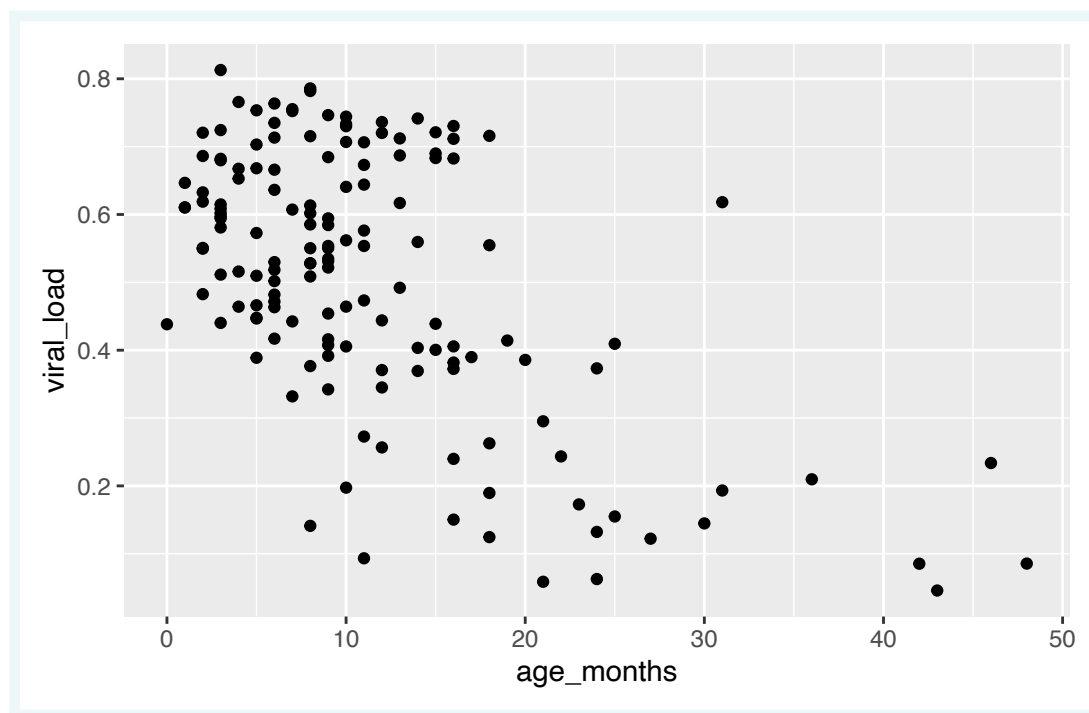
Remember that we specify the first two GG layers as arguments (i.e., inputs) within the `ggplot()` function:

1. We provide the `malidd` data frame with the `data` argument, by inputting `data = malidd`.
2. We define the variables to be plotted in the aesthetics function of the `mapping` argument, by inputting `mapping = aes(x = age_months, y = viral_load)`. Specifically, the variable `age_months` is mapped to the `x` aesthetic, while the variable `viral_load` is mapped to the `y` aesthetic.

We then add **the `geom_*()` function** on a new layer with a `+` sign. The geometric objects (i.e., shapes) needed for a scatter plot are points, so we add `geom_point()`.

After running the following lines of code, you'll produce the scatter plot below:

```
## Simple scatter plot of viral load vs age
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point()
```



This suggests that viral load generally **decreases** with age.

Practice

- Using the `malidd` data frame, create a scatter plot showing the relationship between age and height (`height_cm`).

19.5 Aesthetic modifications

An aesthetic is a visual property of the geometric objects (*geoms*) in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point in different ways by changing the values of its aesthetic properties.

Remember, there are two methods for changing the aesthetic properties of your *geoms* (in this case, points).

1. You can convey information about your data by *mapping* the variables in your dataset to aesthetics in your plot. For this method, you use `aes()` in the mapping argument to associate the name of the aesthetic with a variable to display.
2. You can also *set* the aesthetic properties of your *geoms* *manually*. Here the aesthetic doesn't convey information about a variable, but only changes the appearance of the plot. To change an aesthetic manually, you set the aesthetic by name as an argument of your `geom_*()` function; i.e. it goes *outside* of `aes()`.

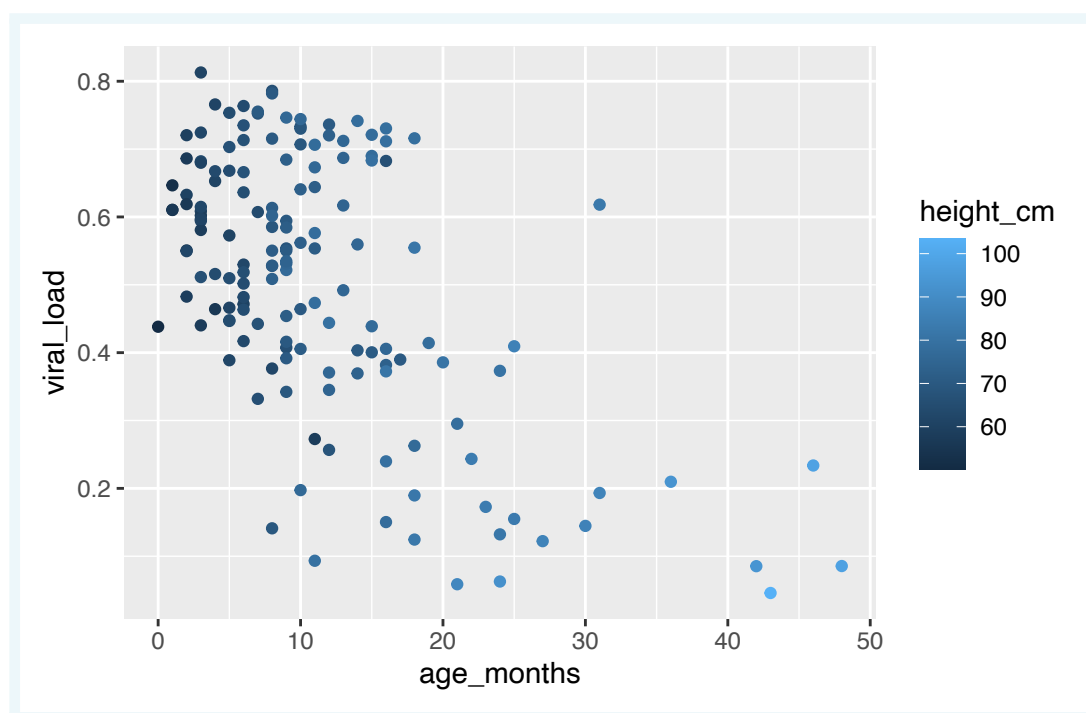
19.5.1 Mapping data to aesthetics

In addition to mapping variables to the **x** and **y** axes like with did above, variables can be mapped to the color, shape, size, opacity, and other visual characteristics of *geoms*. This allows groups of observations to be superimposed in a single graph.

To map a variable to an aesthetic, associate the name of the aesthetic to the name of the variable inside `aes()`. This way, we can visualize a third variable to our simple two dimensional scatter plot by mapping it to a new aesthetic.

For example, let's map `height_cm` to the colors of our points, to show us how height varies with age and viral load:

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = height_cm))
```



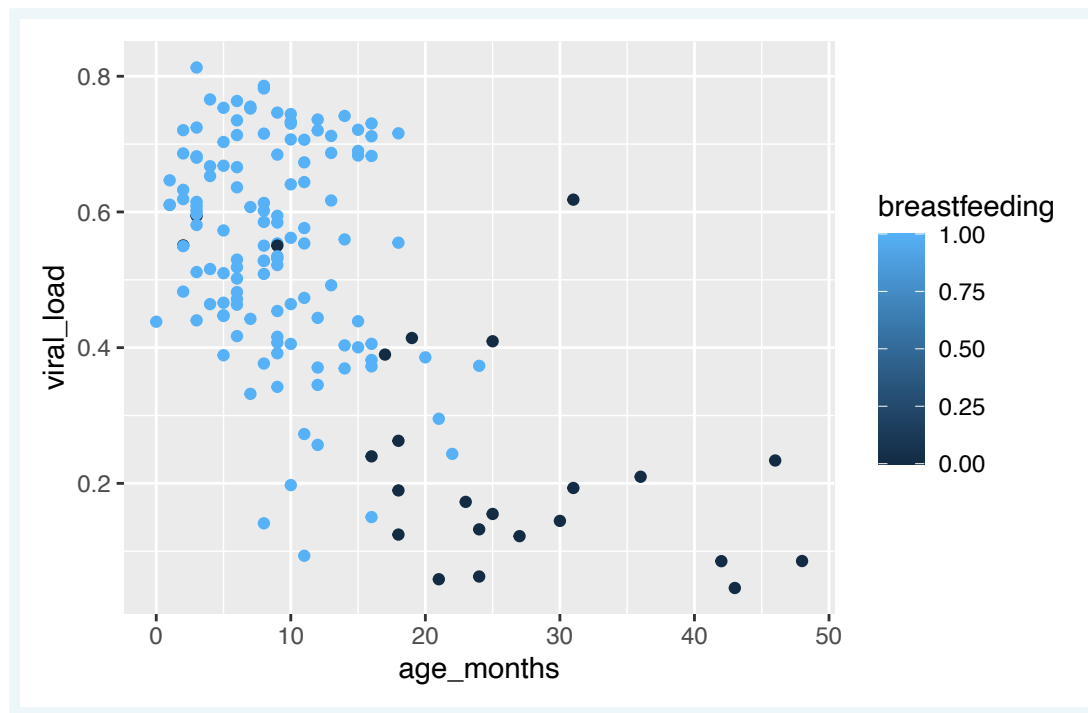
We see that `{ggplot2}` has automatically assigned the values of our variable to an aesthetic, a process known as **scaling**. `{ggplot2}` will also add a legend that explains which levels correspond to which values.

Here the points are colored by different shades of the same blue hue, with darker colors representing lower values.

This shows us that height increases with age, as expected.

Instead of a continuous variable like `height_cm`, we can also map a binary variable like `breastfeeding`, to show us the which children are breastfed and which ones are not:

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = breastfeeding))
```



We get the same gradual color scaling like with did with height. This communicates a continuum of values, rather than the two distinct values in our variable - 0 or 1.

This is because of the data class of the `breastfeeding` variable in `malidd`:

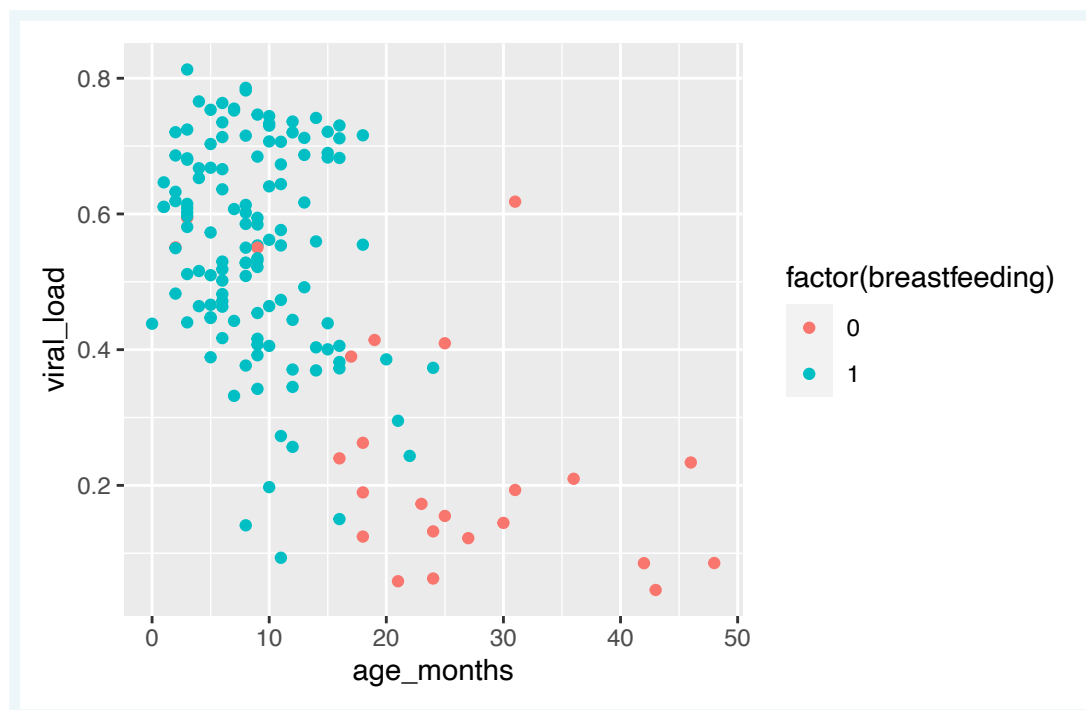
```
class(malidd$breastfeeding)
```

```
[1] "numeric"
```

But even though binary variables are numerical, they represent two *discrete* possibilities. So the continuous color scaling in the plot above is not ideal.

In cases like this, we add the function `factor()` around the `breastfeeding` variable to tell `ggplot()` to treat the variable as a factor. Let's see what happens when we do that:

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = factor(breastfeeding)))
```



When the variable is treated like a factor, the colors chosen are clearly distinguishable. With factors, {ggplot2} will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable. (this is what happened with the `region` variable of the `nigerm` dataframe that we use in the last lesson)

This plot reveals a clear relationship between age and breastfeeding, as we might expect. Children are likely to stop breastfeeding around 20 months of age. In this study, no child at or above 25 months was being breastfed.

Adding colors to the scatter plot allowed us to visualize a **third variable** in addition to the relationship between age and viral load. The third variable could be either discrete or continuous.

💡 Practice

- Using the `malidd` data frame, create a scatter plot showing the relationship between age and viral load, and map a third variable, `freqrespi`, to color:
- Create the same age vs. height scatterplot again, but this time, map the binary variable `fever` to the color of the points. Keep in mind that `fever` should be treated as a factor.

```
## Type and view your answer:
age_height_fever <- "YOUR ANSWER HERE"
age_height_fever
```

19.5.2 Setting fixed aesthetics

Aesthetic arguments set to a fixed value will be static, and the visual effect is not data-dependent. To add a fixed aesthetic, we add as a direct argument of the `geom_*()` function; i.e., it goes *outside* of `mapping = aes()`.

Let's look at some of the aesthetic arguments we can place directly within `geom_point()` to make visual changes to the points in our scatter plot:

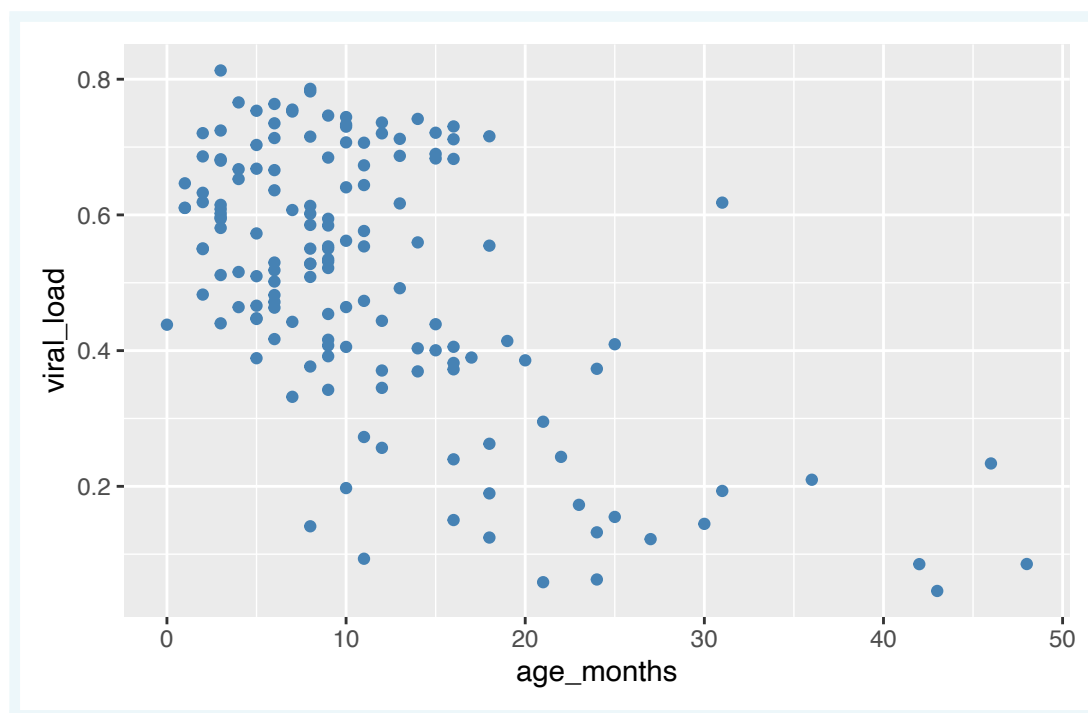
- `color` - point color or point outline color
- `size` - point size
- `alpha` - point opacity
- `shape` - point shape
- `fill` - point fill color (only applies if the point has an outline)

To use these options to create a more attractive scatter plot, you'll need to pick a value for each argument that makes sense for that aesthetic, as shown in the examples below.

19.5.2.1 Changing color, size and alpha

Let's change the color of the points to a fixed value by setting the `color` argument directly within `geom_point()`. The color we choose must be a character string that R recognizes as a color. Here we will set the point colors to steel blue:

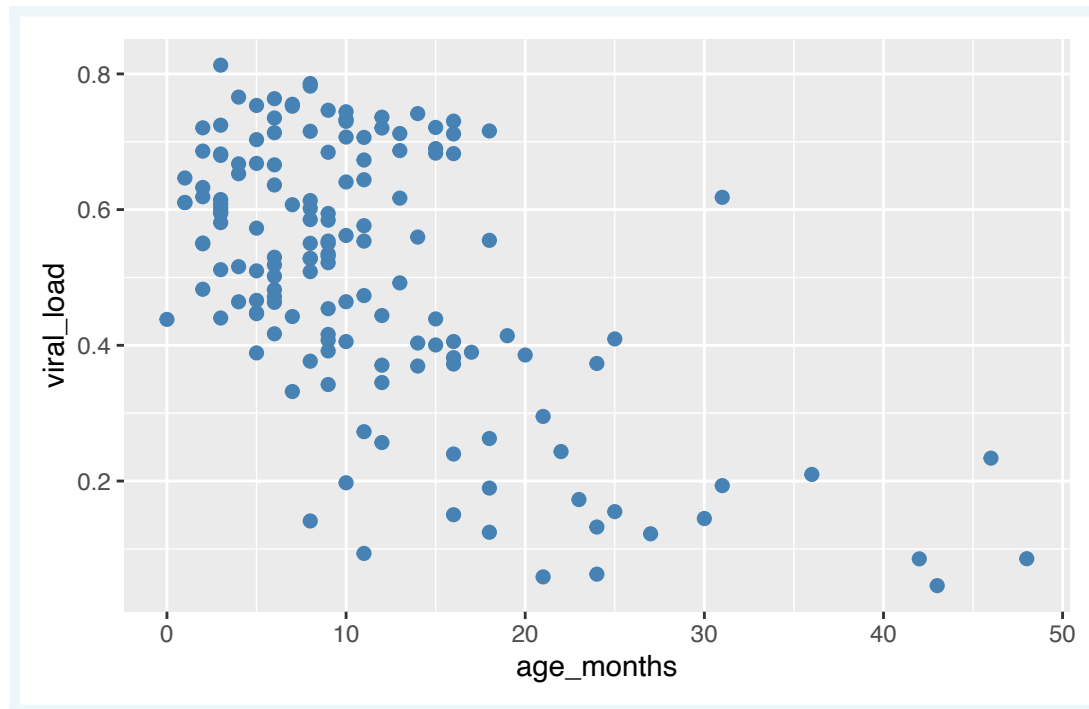
```
## Modify original scatter plot by setting `color = "steelblue"`
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(color = "steelblue")      # set color
```



In addition to changing the default color, now we will modify the size aesthetic of the points by assigning it to a fixed number (in millimeters). The default size is 1 mm, so let's choose a larger value:

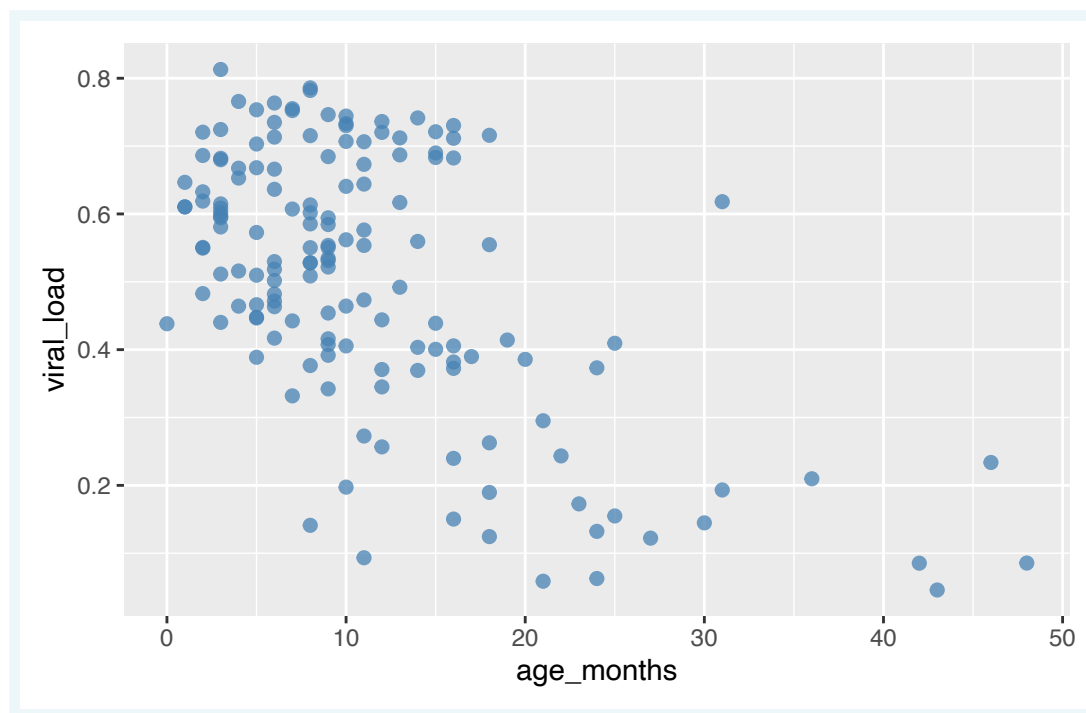
```
## Set size to 2 mm by adding `size = 2`
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
```

```
geom_point(color = "steelblue",      # set color
           size = 2)                 # set size (mm)
```



The `alpha` aesthetic controls the level of opacity of geoms. `alpha` is also numerical, and ranges from 0 (completely transparent) to the default of 1 (completely opaque). Let's make our points more transparent by reducing the opacity:

```
## Set opacity to 75% by adding `alpha = 0.75`
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(color = "steelblue",      # set color
            size = 2,                 # set size (mm)
            alpha = 0.75)             # set level of opacity
```



Now we can see where multiple points overlap. This is a useful parameter for scatter plots where there is **overplotting**.

Remember, changing the color, size, or opacity of our points here is not conveying any information in the data - they are design choices we make to create prettier plots.

Practice

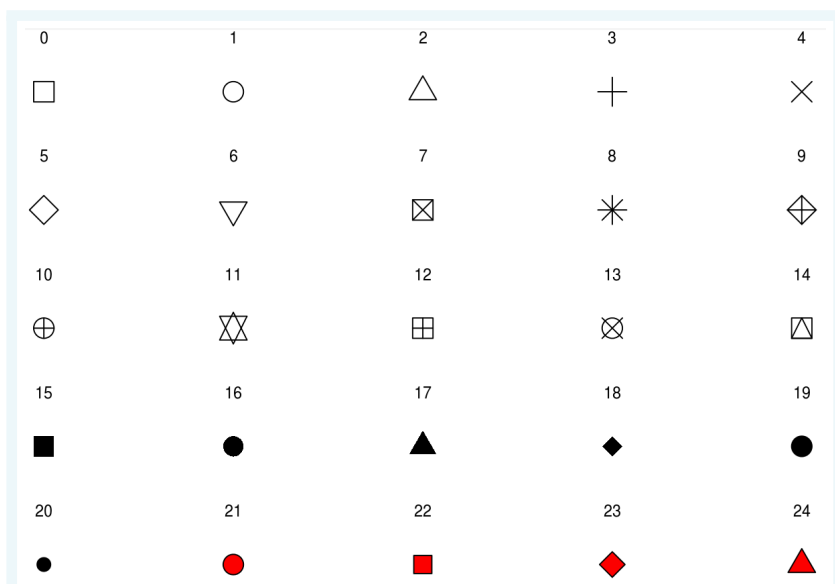
- Create a scatter plot with the same variables as the previous example, but change the color of the points to `cornflowerblue`, increase the size of points to 3 mm and set the opacity to 60%.

19.5.2.2 Changing shape and fill

We can change the appearance of points in a scatter plot with the `shape` aesthetic.

To change the shape of your geoms to a fixed value, set `shape` equal to a number corresponding to your desired shape.

`{ggplot2}` will accept the following numbers:



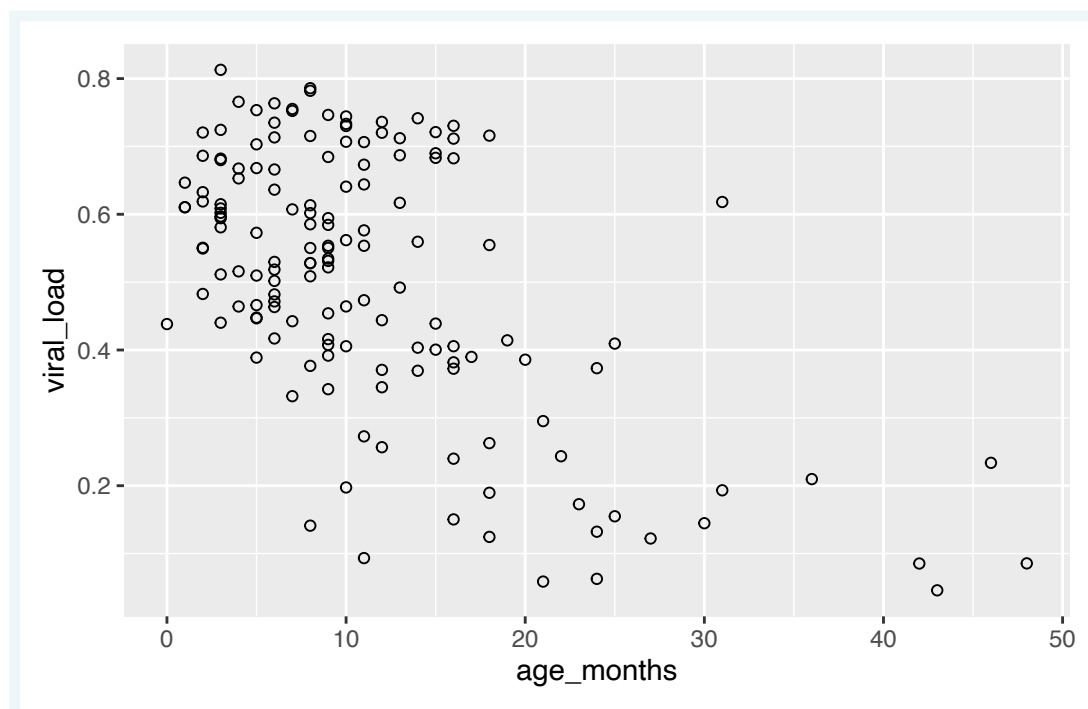
Notice that some of the shapes are

filled in with red. This indicates that objects 21-24 are sensitive to both `color` and `fill`, but the others are only sensitive to `color`.

First let's modify our original scatterplot by changing the shapes to a something that can be filled in:

```
## Set shape to fillable circles by adding `shape = 21`

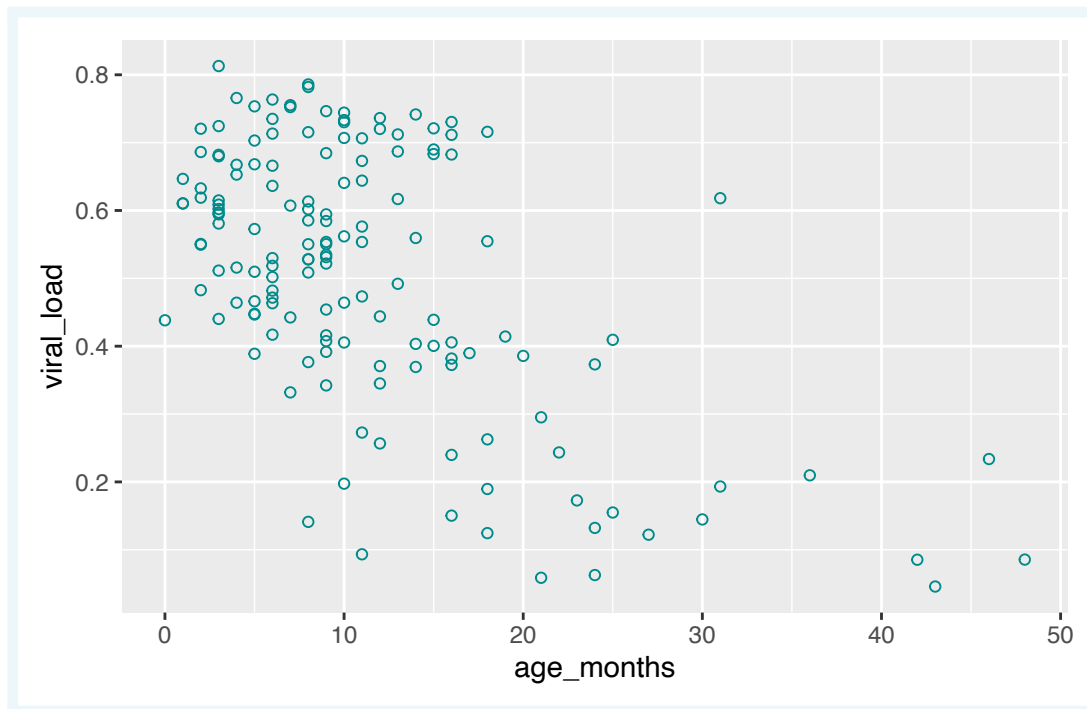
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(shape = 21) # set shapes to display
```



Fillable shapes can have different colors for the outline and interior. Changing the `color` aesthetic will only change the outline of our points:

```
## Set outline color of the shapes by adding `color = cyan4`

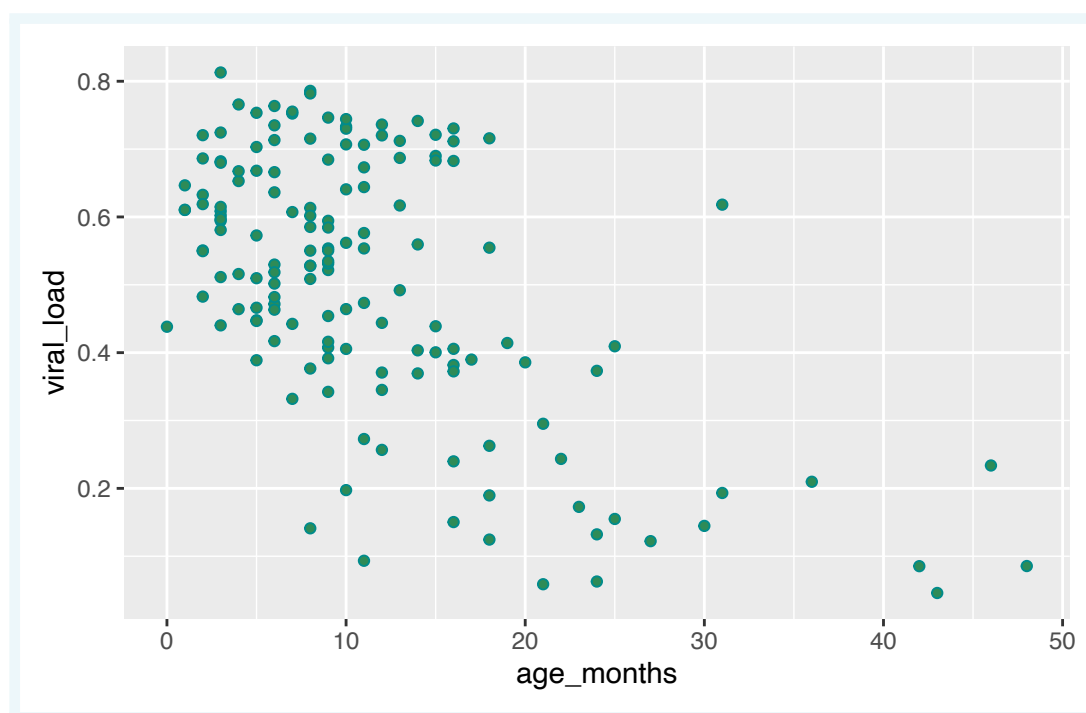
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(shape = 21,                # set shapes to display
            color = "cyan4")           # set outline color
```



Now let's fill in the points:

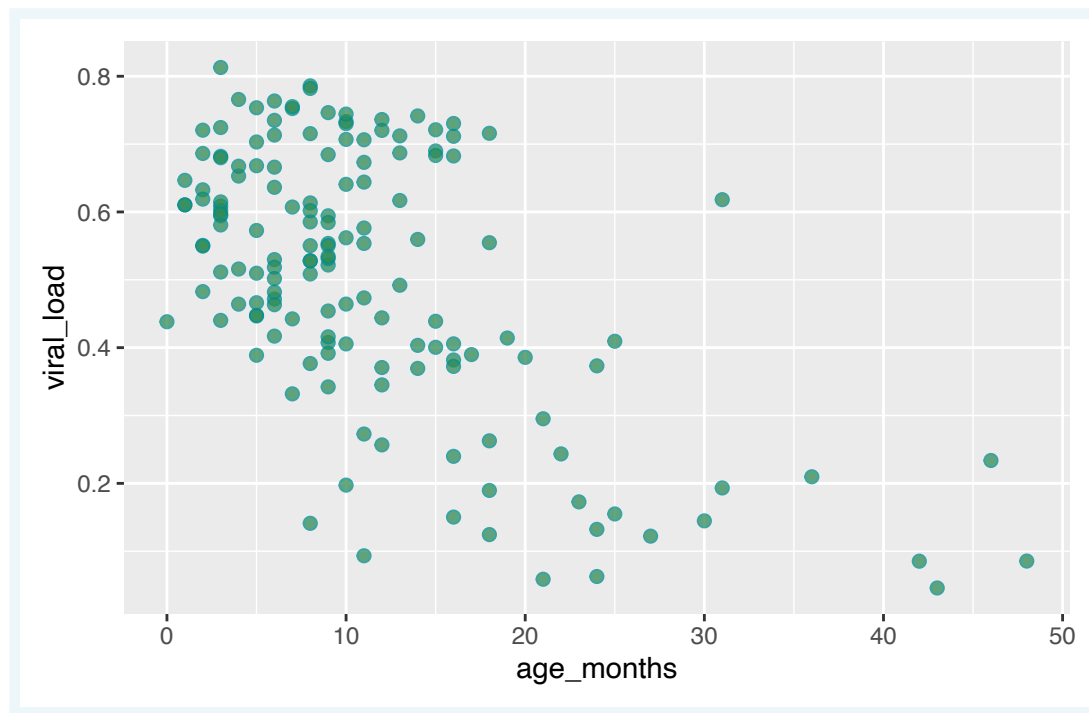
```
## Set interior color of the shapes by adding `fill = "seagreen"`

ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(shape = 21,                # set shapes to display
            color = "cyan4",           # set outline color
            fill = "seagreen")         # set fill color
```



We can improve the readability by increasing size and reducing opacity with size and alpha, like we did before:

```
ggplot(data = malidd,  
       mapping = aes(x = age_months,  
                     y = viral_load)) +  
  geom_point(shape = 21,           # set shapes to display  
            color = "cyan4",      # set outline color  
            fill = "seagreen",    # set fill color  
            size = 2,             # set size (mm)  
            alpha = 0.75)         # set level of opacity
```



19.6 Adding a trend line

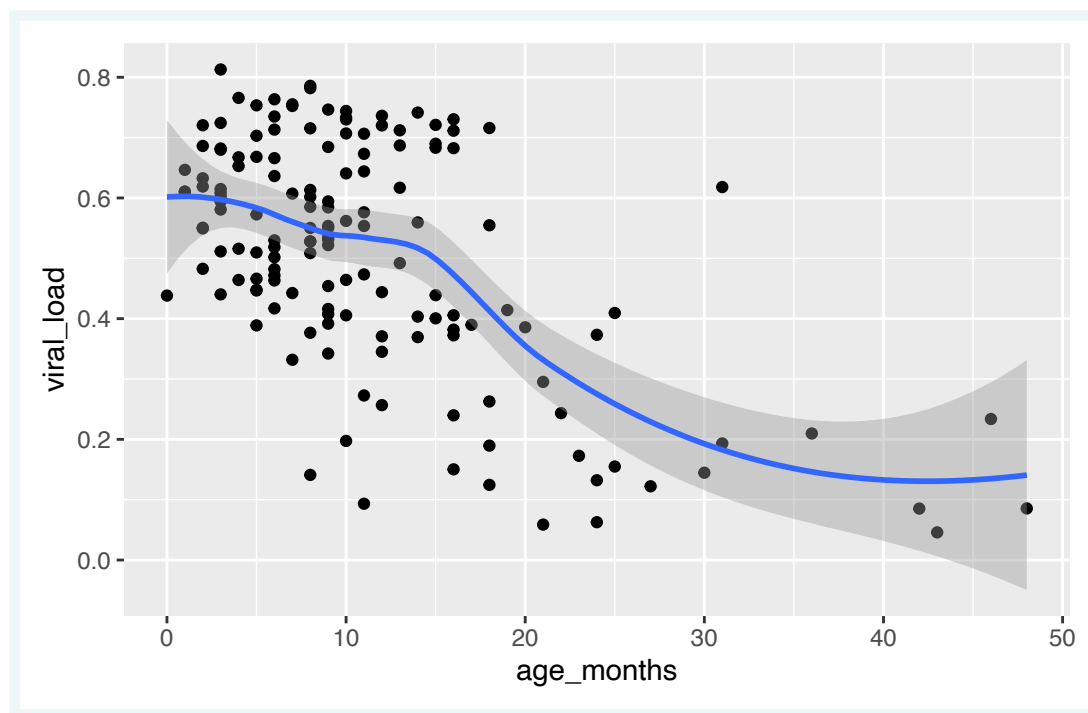
It can be hard to view relationships or trends with just points alone. Often we want to add a smoothing line in order to see what the trends look like. This can be especially helpful when trying to understand regressions.

To get a better idea of the relationship between these two variables, we can add a trend line (also known as a best fit line or a smoothing line).

To do this, we add the function `geom_smooth()` to our scatter plot:

```
ggplot(data = malidd,  
       mapping = aes(x = age_months,  
                     y = viral_load)) +  
  geom_point() +  
  geom_smooth()
```

`geom_smooth()` using `method = 'loess'` and `formula = 'y ~ x'`



The smoothing line comes after our points as another geometric layer added onto our plot.

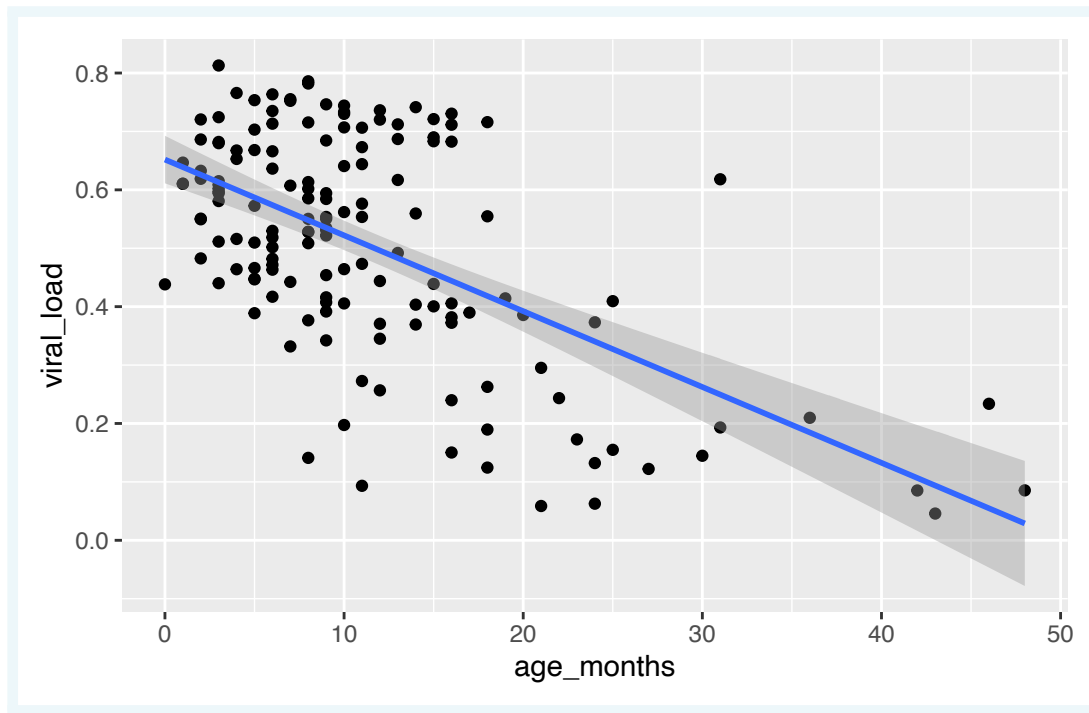
The default smoothing function used in this scatter plot is “loess” which stands for **l**ocally **w**eighted **s**catter **p**lot **s**moothing. Loess smoothing is a process used by many statistical softwares. In {ggplot2} this generally should be done when you have less than 1000 points, otherwise it can be time consuming.

Many other smoothing functions can also be used in `geom_smooth()`.

Let’s request a linear regression method. This time we will use a generalized linear model by setting the `method` argument inside `geom_smooth()`:

```
## Change to a linear smoothing function with `method = "glm"`
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point() +
  geom_smooth(method = "glm")
```

``geom_smooth()`` using `formula = 'y ~ x'`

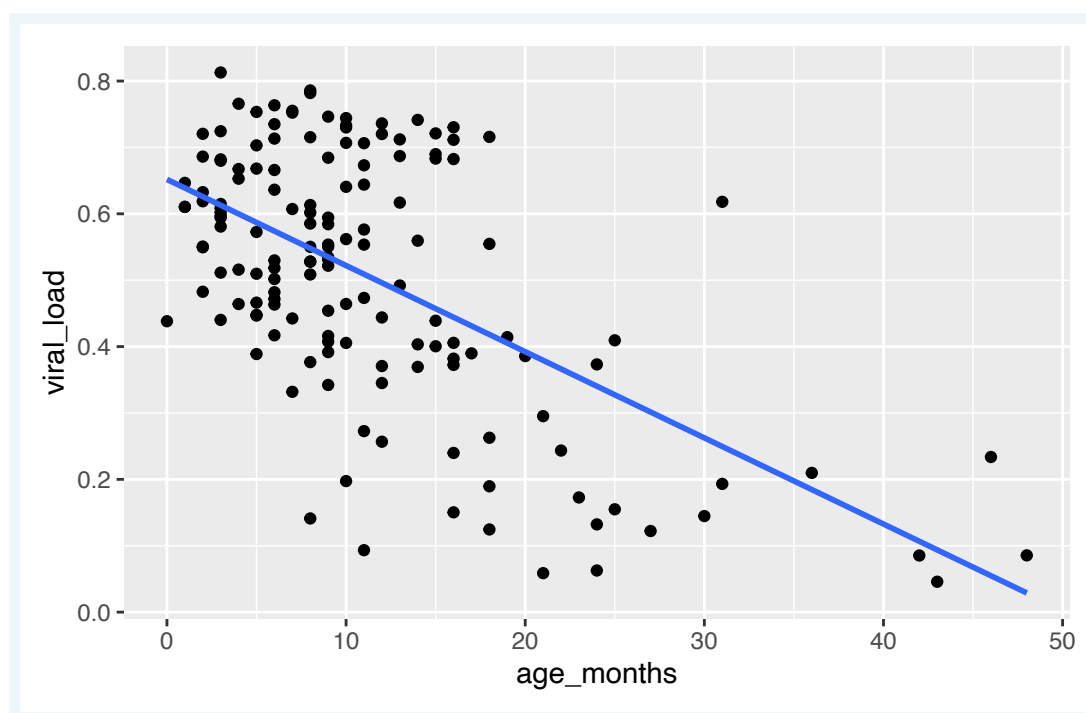


By default, 95% confidence limits for these lines are displayed.

You can suppress the confidence bands by including the argument `se = FALSE` inside `geom_smooth()`:

```
## Remove confidence interval bands by adding `se = FALSE`
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point() +
  geom_smooth(method = "glm",
             se = FALSE)
```

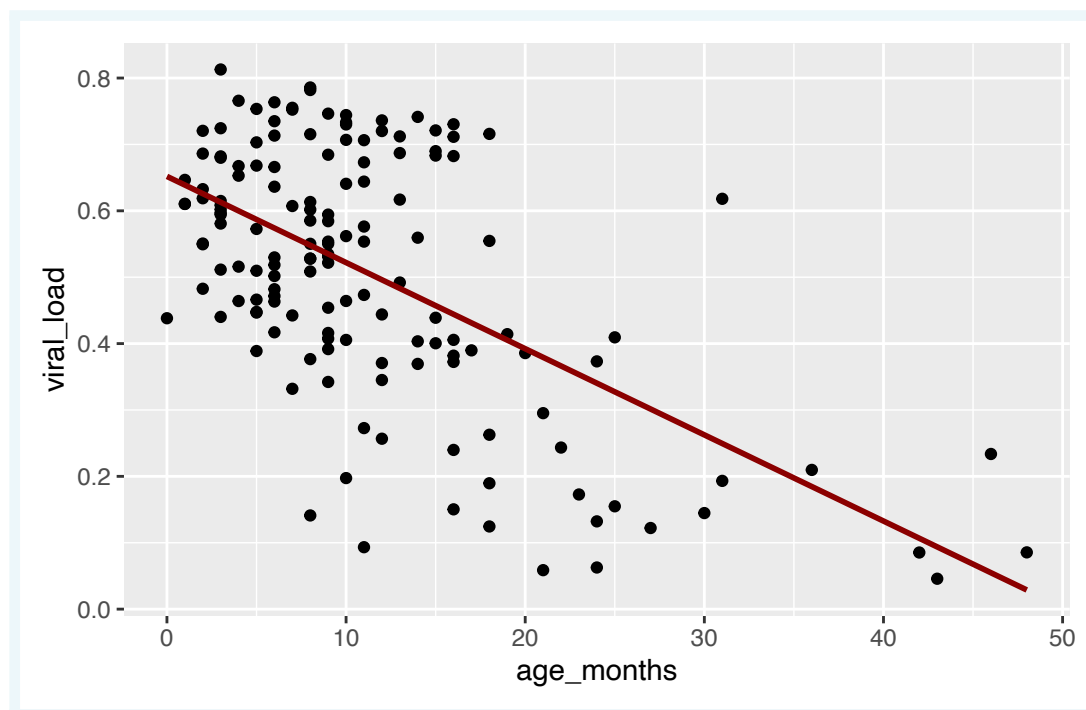
``geom_smooth()`` using `formula = 'y ~ x'`



In addition to changing the method, let's add the `color` argument inside `geom_smooth()` to change the color of the line.

```
## Change the color of the trend line by adding `color = "darkred"`  
ggplot(data = malidd,  
       mapping = aes(x = age_months,  
                     y = viral_load)) +  
  geom_point() +  
  geom_smooth(method = "glm",  
             se = FALSE,  
             color = "darkred")
```

``geom_smooth()`` using `formula = 'y ~ x'`



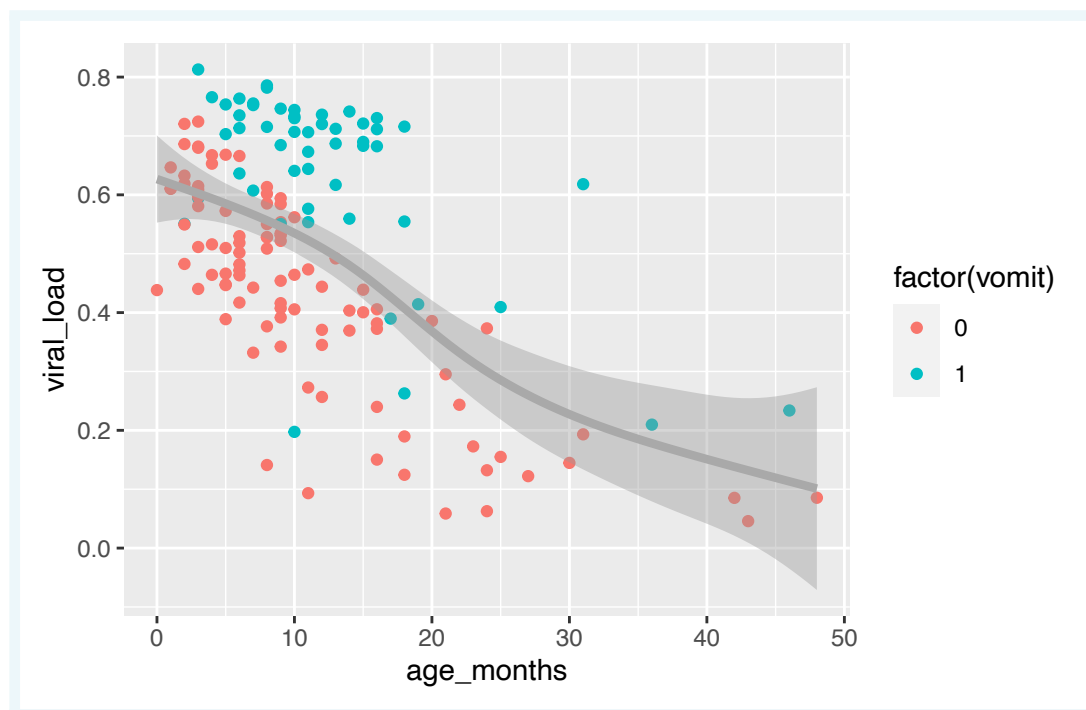
This linear regression concurs with what we initially observed in the first scatter plot. A *negative relationship* exists between `age_months` and `viral_load`: as age increases, viral load tends to decrease.

Let's add a third variable from the `malidd` dataset called `vomit`. This which is a binary variable that records whether or not the patient vomited. We will add the `vomit` variable to the plot by mapping it to the color aesthetic. We will again change the smoothing method to generalized additive model ("gam") and make some aesthetic modifications to the line in the `geom_smooth()` layer.

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = factor(vomit))) +
  geom_smooth(method = "gam",
             size = 1.5,
             color = "darkgray")
```

Warning: Using ``size`` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use ``linewidth`` instead.

``geom_smooth()`` using formula = `'y ~ s(x, bs = "cs")'`



Observe the distribution of blue points (children who vomited) compared to red points (children who did not vomit). The blue points mostly occur above the trend line. This shows that higher viral loads were not only associated with younger children, but that children with higher viral loads were more likely to exhibit symptoms of vomiting.

💡 Practice

- Create a scatter plot with the `age_months` and `height_cm` variables. Set the color of the points to "steelblue", the size to 2.5mm, the opacity to 80%. Then add trend line with the smoothing method "lm" (linear model). To make the trend line stand out, set its color to "indianred3".
- Recreate the plot you made in the previous question, but this time adapt the code to change the shape of the points to tilted rectangles (number 23), and add the body temperature variable (`temp`) by **mapping** it to fill color of the points.

```
## Type and view your answer:
age_height_3 <- "YOUR ANSWER HERE"
age_height_3
```

19.7 Summary

Scatter plots display the relationship between two numerical variables.

With medium to large datasets, you may need to play around with the different modifications to scatter plots we saw such as adding trend lines, changing the color, size, shape, fill, or opacity of the points. This tweaking is often a fun part of data visualization, since you'll have the chance to see different relationships emerge as you tinker with your plots.

References

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

- Ismay, Chester, and Albert Y. Kim. 2022. *A ModernDive into R and the Tidyverse*. <https://moderndive.com/>.
- Kabacoff, Rob. 2020. *Data Visualization with R*. <https://rkabacoff.github.io/datavis/>.
- Giroux-Bougard, Xavier, Maxwell Farrell, Amanda Winegardner, Étienne Low-Decarie and Monica Grados. 2020. *Workshop 3: Introduction to Data Visualisation with {ggplot2}*. <http://r.qcbs.ca/workshop03/book-en/>.

19.8 Solutions

```
.SOLUTION_age_height()
```

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = height_cm)) +
  geom_point()
```

```
.SOLUTION_age_height_respi()
```

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = freqrespi))
```

```
.SOLUTION_age_viral_respi()
```

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = viral_load)) +
  geom_point(mapping = aes(color = freqrespi))
```

```
.SOLUTION_age_height_fever()
```

```
ggplot(data = malidd,
       mapping = aes(x = age_months,
                     y = height_cm)) +
  geom_point(mapping = aes(color = factor(fever)))
```

```
.SOLUTION_age_viral_blue()
```

```
ggplot(data = malidd,  
       mapping = aes(x = age_months,  
                     y = viral_load)) +  
  geom_point(color = "cornflowerblue",  
            size = 3,  
            alpha = 0.6)
```

```
.SOLUTION_age_height_2()
```

```
ggplot(data = malidd,  
       mapping = aes(x = age_months,  
                     y = height_cm)) +  
  geom_point(color = "steelblue",  
            size = 2.5,  
            alpha = 0.8) +  
  geom_smooth(method = "lm", color = "indianred3")
```

```
.SOLUTION_age_height_3()
```

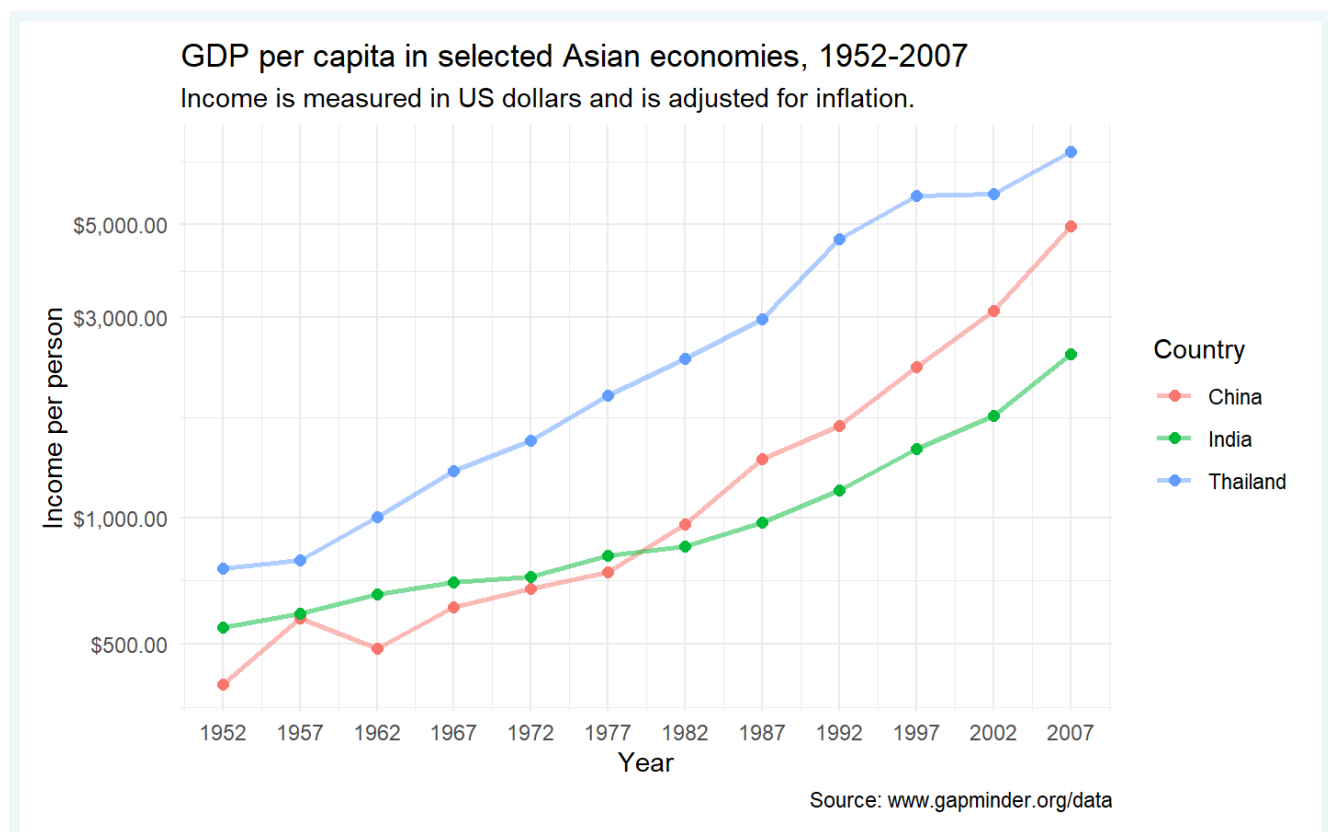
```
ggplot(data = malidd,  
       mapping = aes(x = age_months, y = height_cm)) +  
  geom_point(color = "steelblue",  
            size = 2.5,  
            alpha = 0.8,  
            shape = 23,  
            mapping = aes(fill = temp)) +  
  geom_smooth(method = "lm", color = "indianred3")
```

Chapter 20

Lines, scales, and labels

20.1 Learning Objectives

1. You can create **line graphs** to visualize relationships between two numerical variables with `geom_line()`.
2. You can **add points** to a line graph with `geom_point()`.
3. You can use aesthetics like `color`, `size`, `color`, and `linetype` to modify line graphs.
4. You can **manipulate axis scales** for continuous data with `scale_*_continuous()` and `scale_*_log10()`.
5. You can **add labels** to a plot such as a title, subtitle, or caption with the `labs()` function.



20.2 Introduction

Line graphs are used to show **relationships** between two **numerical variables**, just like scatterplots. They are especially useful when the variable on the x-axis, also called the *explanatory* variable, is of a **sequential** nature. In other words, there is an inherent ordering to the variable.

The most common examples of line graphs have some notion of **time on the x-axis**: hours, days, weeks, years, etc. Since time is sequential, we connect consecutive observations of the variable on the y-axis with a line. Line graphs that have some notion of time on the x-axis are also called **time series plots**.

20.3 Packages

```
## Load packages
pacman::p_load(tidyverse,
               gapminder,
               here)
```

20.4 The gapminder data frame

In February 2006, a Swedish physician and data advocate named Hans Rosling gave a famous TED talk titled “The best stats you’ve ever seen” where he presented global economic, health, and development data compiled by the Gapminder Foundation.

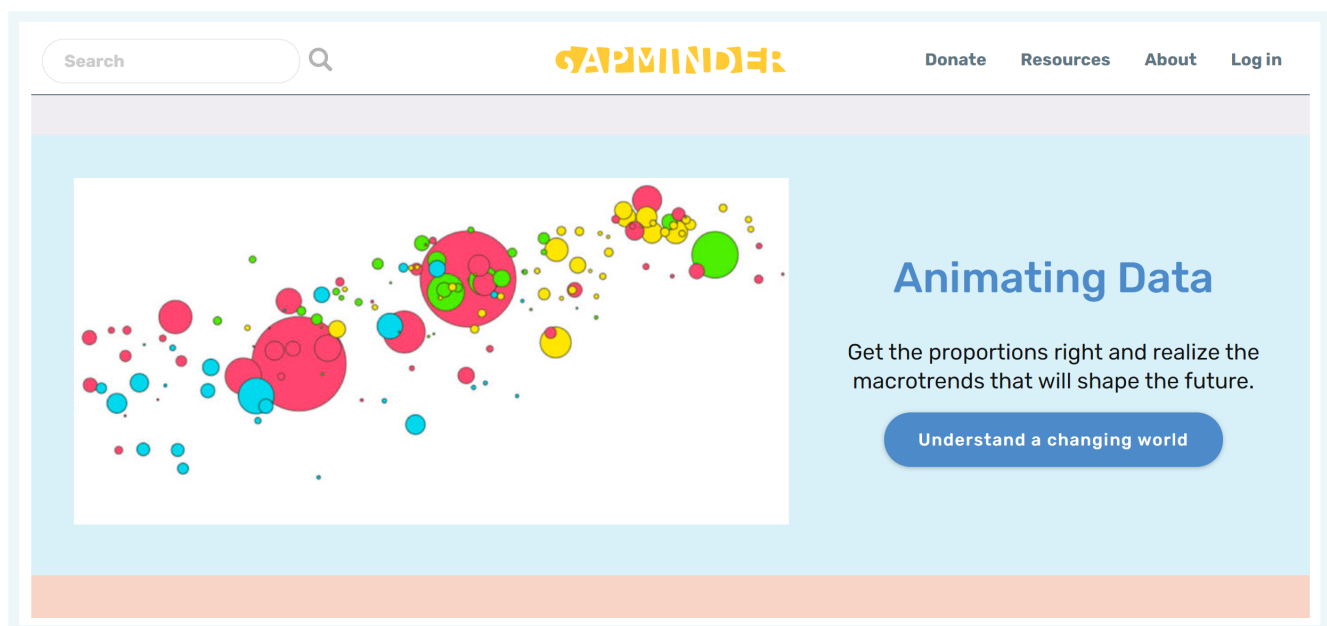


Figure 20.1: Interactive data visualization tools with up-to-date data are available on the Gapminder’s website.

We can access a clean subset of this data with the R package **{gapminder}**, which we just loaded.

```
## Load gapminder data frame from the gapminder package
data(gapminder, package="gapminder")

## Print dataframe
```

```
gapminder
```

```
# A tibble: 10 x 6
  country      continent  year lifeExp      pop gdpPercap
  <fct>        <fct>    <int>  <dbl>    <int>    <dbl>
1 Afghanistan Asia      1952   28.8  8425333    779.
2 Afghanistan Asia      1957   30.3  9240934    821.
3 Afghanistan Asia      1962   32.0 10267083    853.
4 Afghanistan Asia      1967   34.0 11537966    836.
5 Afghanistan Asia      1972   36.1 13079460    740.
6 Afghanistan Asia      1977   38.4 14880372    786.
7 Afghanistan Asia      1982   39.9 12881816    978.
8 Afghanistan Asia      1987   40.8 13867957    852.
9 Afghanistan Asia      1992   41.7 16317921    649.
10 Afghanistan Asia      1997   41.8 22227415    635.
```

Each row in this table corresponds to a country-year combination. For each row, we have 6 columns:

- 1) **country**: Country name
- 2) **continent**: Geographic region of the world
- 3) **year**: Calendar year
- 4) **lifeExp**: Average number of years a newborn child would live if current mortality patterns were to stay the same
- 5) **pop**: Total population
- 6) **gdpPercap**: Gross domestic product per person (inflation-adjusted US dollars)

The `str()` function can tell us more about these variables.

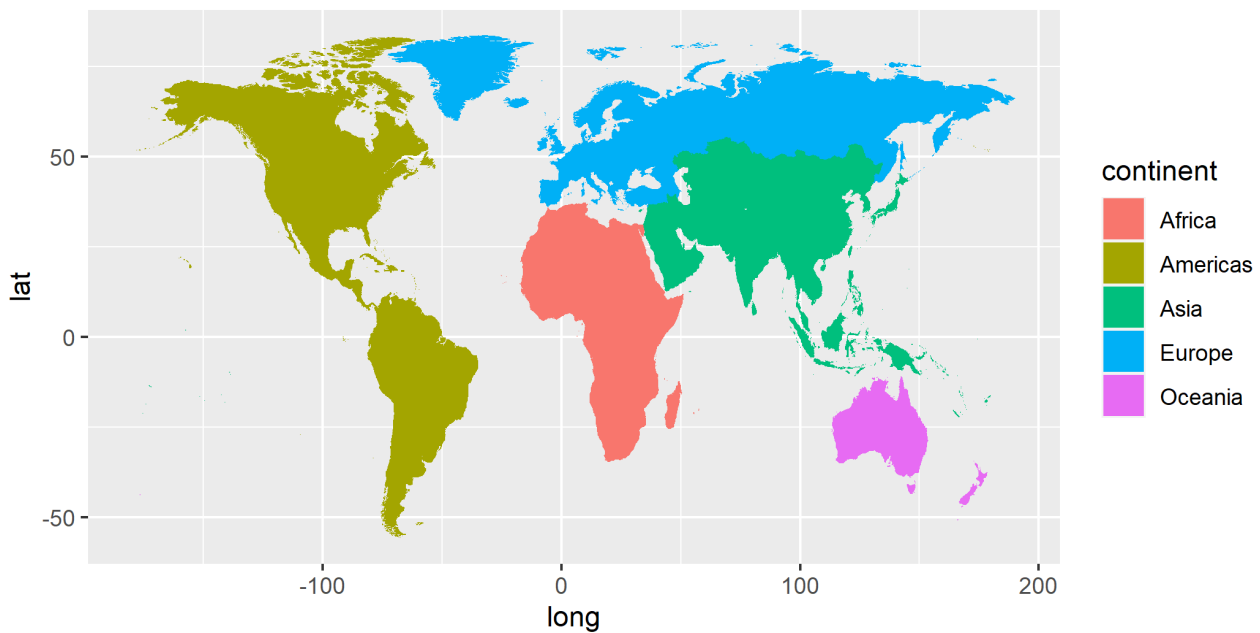
```
## Data structure
str(gapminder)
```

```
tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
 $ country  : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
 $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957
 $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

This version of the `gapminder` dataset contains information for **142 countries**, divided in to **5 continents**.

Gapminder world regions

Five regions in the `continent` variable of `gapminder`



```
## Data summary
summary(gapminder)
```

country	continent	year	lifeExp
Afghanistan: 12	Africa :624	Min. :1952	Min. :23.60
Albania : 12	Americas:300	1st Qu.:1966	1st Qu.:48.20
Algeria : 12	Asia :396	Median :1980	Median :60.71
Angola : 12	Europe :360	Mean :1980	Mean :59.47
Argentina : 12	Oceania : 24	3rd Qu.:1993	3rd Qu.:70.85
Australia : 12		Max. :2007	Max. :82.60
(Other) :1632			

pop	gdpPercap
Min. :6.001e+04	Min. : 241.2
1st Qu.:2.794e+06	1st Qu.: 1202.1
Median :7.024e+06	Median : 3531.8
Mean :2.960e+07	Mean : 7215.3
3rd Qu.:1.959e+07	3rd Qu.: 9325.5
Max. :1.319e+09	Max. :113523.1

Data are recorded every 5 years from 1952 to 2007 (a total of 12 years).

Let's say we want to visualize the relationship between time (year) and life expectancy (lifeExp).

For now let's just focus on one country - United States. First, we need to create a new data frame with only the data from this country.

```
## Select US cases
gap_US <- dplyr::filter(gapminder,
  country == "United States")
```

```
gap_US
```

```
# A tibble: 10 x 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	United States	Americas	1952	68.4	157553000	13990.
2	United States	Americas	1957	69.5	171984000	14847.
3	United States	Americas	1962	70.2	186538000	16173.
4	United States	Americas	1967	70.8	198712000	19530.
5	United States	Americas	1972	71.3	209896000	21806.
6	United States	Americas	1977	73.4	220239000	24073.
7	United States	Americas	1982	74.6	232187835	25010.
8	United States	Americas	1987	75.0	242803533	29884.
9	United States	Americas	1992	76.1	256894189	32004.
10	United States	Americas	1997	76.8	272911760	35767.

i Reminder

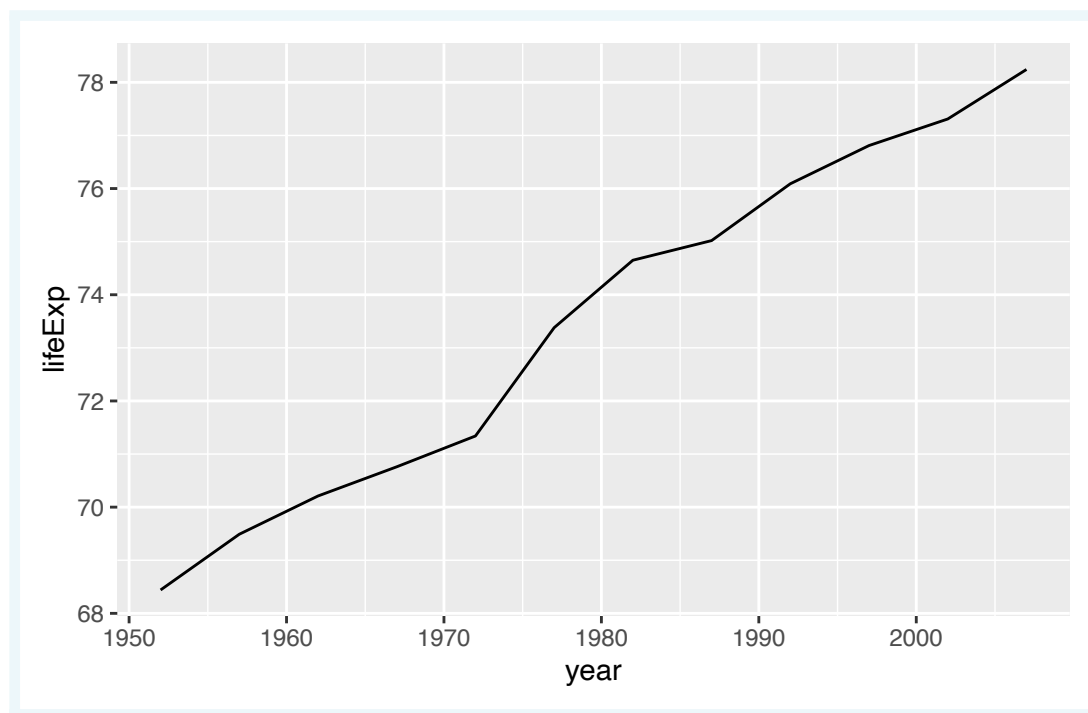
The code above is covered in our course on Data Wrangling using the {dplyr} package. Data wrangling is the process of transforming and modifying existing data with the intent of making it more appropriate for analysis purposes. For example, this code segments used the `filter()` function to create a new data frame (`gap_US`) by choosing only a subset of rows of original `gapminder` data frame (only those that have “United States” in the `country` column).

20.5 Line graphs via `geom_line()`

Now we’re ready to feed the `gap_US` data frame to `ggplot()`, mapping **time** in years on the horizontal x axis and **life expectancy** on the vertical y axis.

We can visualize this time series data by using `geom_line()` to create a line graph, instead of using `geom_point()` like we used previously to create scatterplots:

```
## Simple line graph
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line()
```



Much as with the `ggplot()` code that created the scatterplot of age and viral load with `geom_point()`, let's break down this code piece-by-piece in terms of the grammar of graphics:

Within the `ggplot()` function call, we specify two of the components of the grammar of graphics as arguments:

1. The data to be the `gap_US` data frame by setting `data = gap_US`.
2. The aesthetic mapping by setting `mapping = aes(x = year, y = lifeExp)`. Specifically, the variable `year` maps to the `x` position aesthetic, while the variable `lifeExp` maps to the `y` position aesthetic.

After telling R which data and aesthetic mappings we wanted to plot we then added the third essential component, the geometric object using the `+` sign. In this case, the geometric object was set to lines using `geom_line()`.

💡 Practice

Create a time series plot of the GDP per capita (`gdpPerCap`) recorded in the `gap_US` data frame by using `geom_line()` to create a line graph.

20.5.1 Fixed aesthetics in `geom_line()`

The color, line width and line type of the line graph can be customized making use of `color`, `size` and `linetype` arguments, respectively.

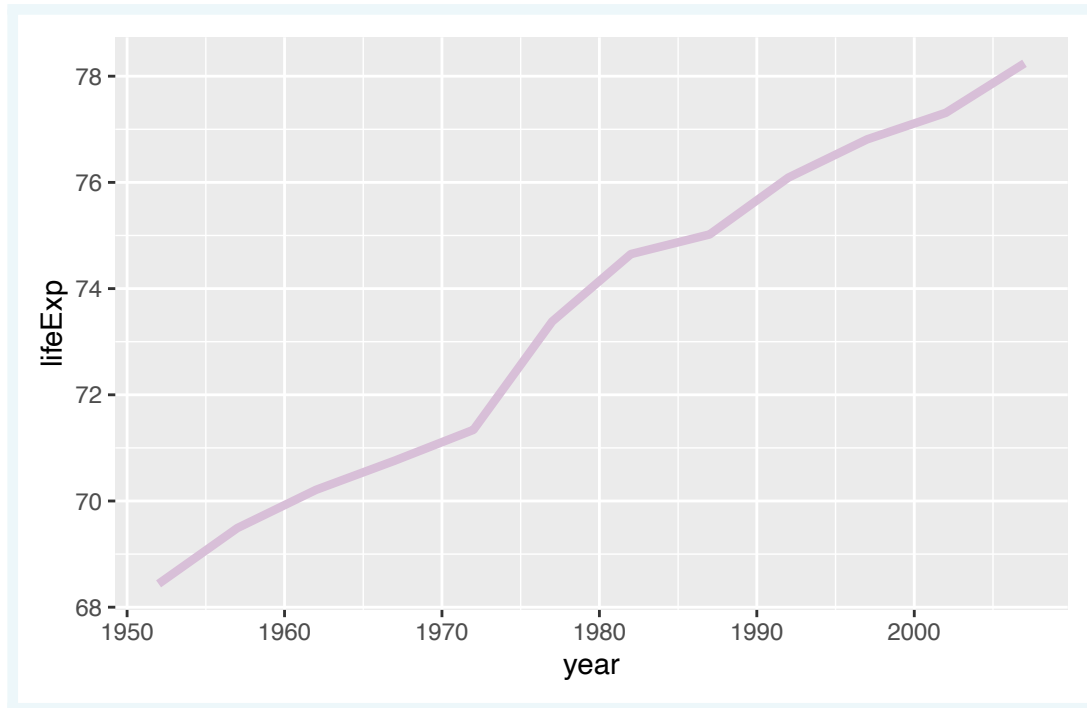
We've changed the color and size of geoms in previous lessons.

Here we will add these as fixed aesthetics:







```
## enhanced line graph with color and size as fixed aesthetics
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
```

```
geom_line(color = "thistle",
          size = 1.5)
```

Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
 i Please use `linewidth` instead.

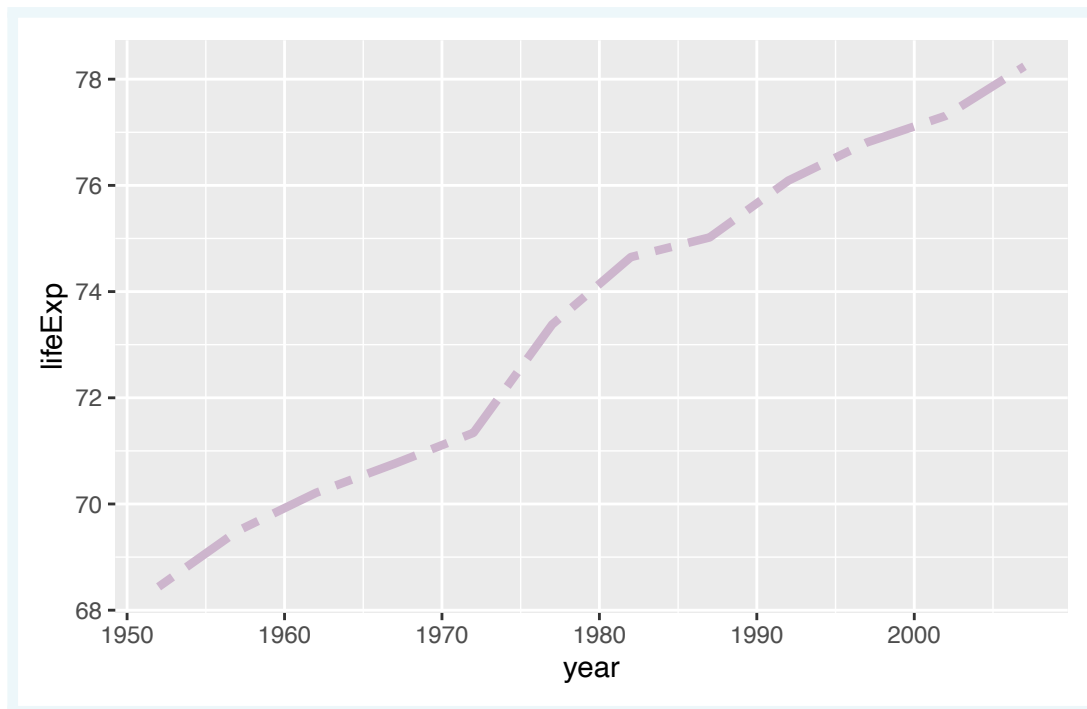


In this lesson we introduce a new fixed aesthetic that is specific to line graphs: `linetype` (or `lty` for short).

	<code>lty = 0</code> or 'blank'
	<code>lty = 1</code> or 'solid'
	<code>lty = 2</code> or 'dashed'
	<code>lty = 3</code> or 'dotted'
	<code>lty = 4</code> or 'dotdash'
	<code>lty = 5</code> or 'longdash'
	<code>lty = 6</code> or 'twodash'

Line type can be specified using a name or with an integer. Valid line types can be set using a human readable character string: "blank", "solid", "dashed", "dotted", "dotdash", "longdash", and "twodash" are all understood by `linetype` or `lty`.

```
## Enhanced line graph with color, size, and line type as fixed aesthetics
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(color = "thistle3",
           size = 1.5,
           linetype = "twodash")
```



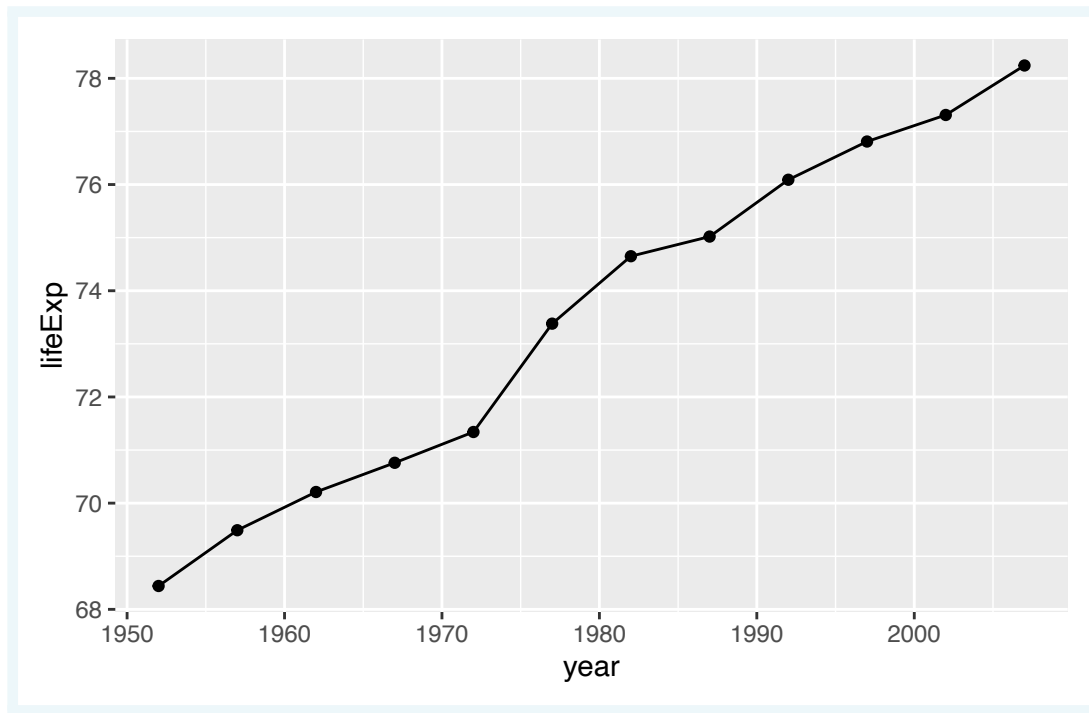
In these line graphs, it can be hard to tell where exactly there data points are. In the next plot, we'll add points to make this clearer.

20.6 Combining compatible geoms

As long as the geoms are compatible, we can layer them on top of one another to further customize a graph.

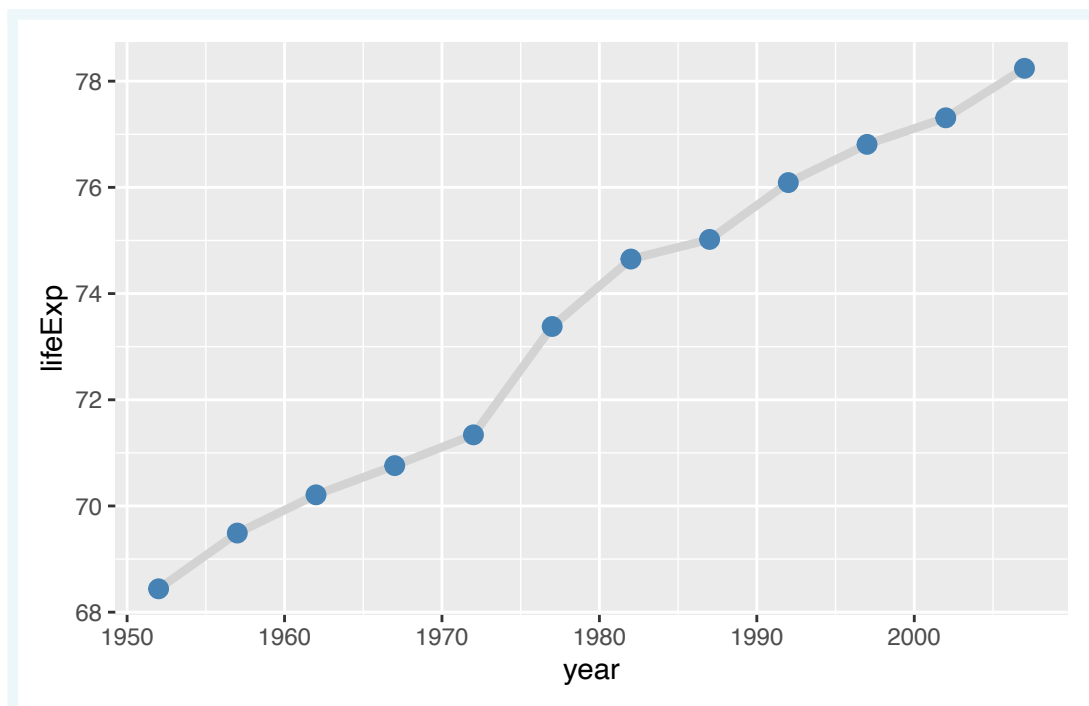
For example, we can add points to our line graph using the `+` sign to add a second geom layer with `geom_point()`:

```
## Simple line graph with points
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line() +
  geom_point()
```



We can create a more attractive plot by customizing the size and color of our geoms.

```
## Line graph with points and fixed aesthetics
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1.5,
            color = "lightgrey") +
  geom_point(size = 3,
             color = "steelblue")
```



Practice

Building on the code above, visualize the relationship between time and **GPD per capita** from the `gap_US` data frame.

Use both points and lines to represent the data.

Change the line type of the line and the color of the points to any valid values of your choice.

20.7 Mapping data to multiple lines

In the previous section, we only looked at data from one country, but what if we want to plot data for multiple countries and compare?

First let's add two more countries to our data subset:

```
## Create data subset for visualizing multiple categories
gap_mini <- filter(gapminder,
                   country %in% c("United States",
                                "Australia",
                                "Germany"))

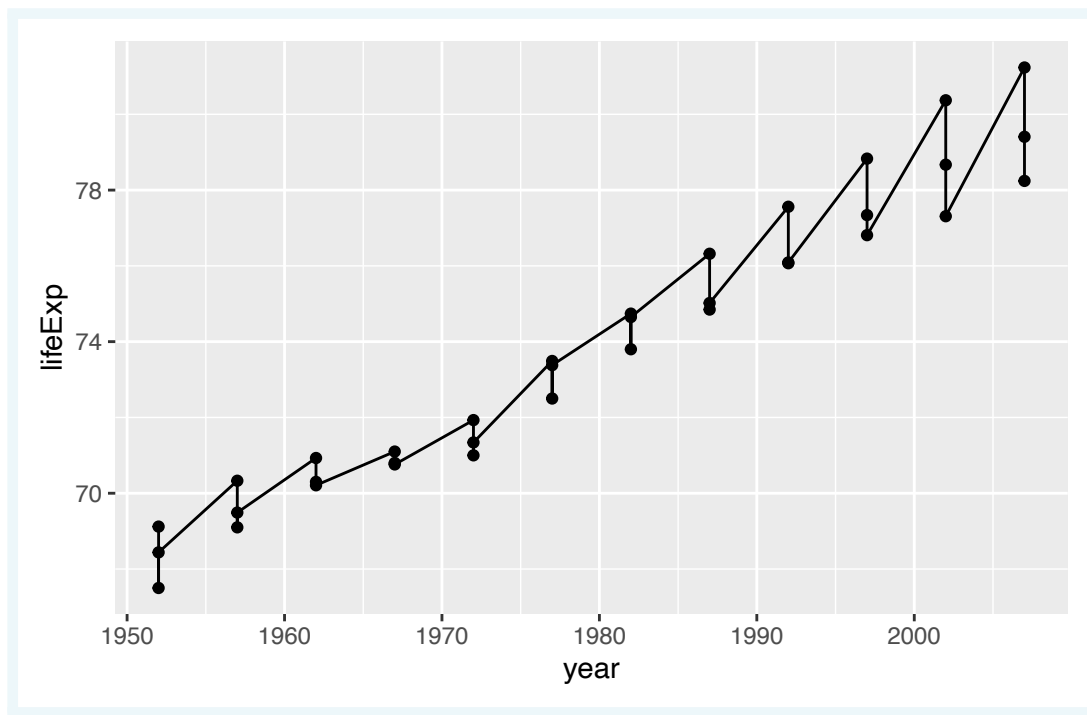
gap_mini
```

A tibble: 10 x 6

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Australia	Oceania	1952	69.1	8691212	10040.
2	Australia	Oceania	1957	70.3	9712569	10950.
3	Australia	Oceania	1962	70.9	10794968	12217.
4	Australia	Oceania	1967	71.1	11872264	14526.
5	Australia	Oceania	1972	71.9	13177000	16789.
6	Australia	Oceania	1977	73.5	14074100	18334.
7	Australia	Oceania	1982	74.7	15184200	19477.
8	Australia	Oceania	1987	76.3	16257249	21889.
9	Australia	Oceania	1992	77.6	17481977	23425.
10	Australia	Oceania	1997	78.8	18565243	26998.

If we simply enter it using the same code and change the data layer, the lines are not automatically separated by country:

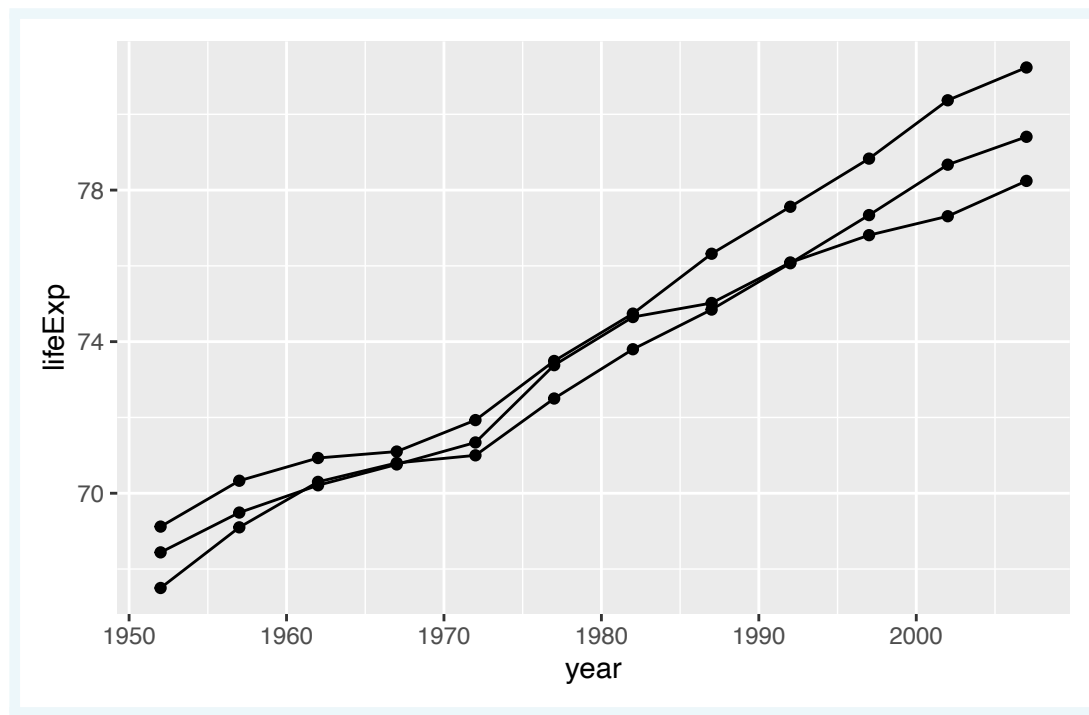
```
## Line graph with no grouping aesthetic
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                     x = year)) +
  geom_line() +
  geom_point()
```



This is not a very helpful plot for comparing trends between groups.

To tell `ggplot()` to map the data from each country separately, we can use the `group` argument as an aesthetic mapping:

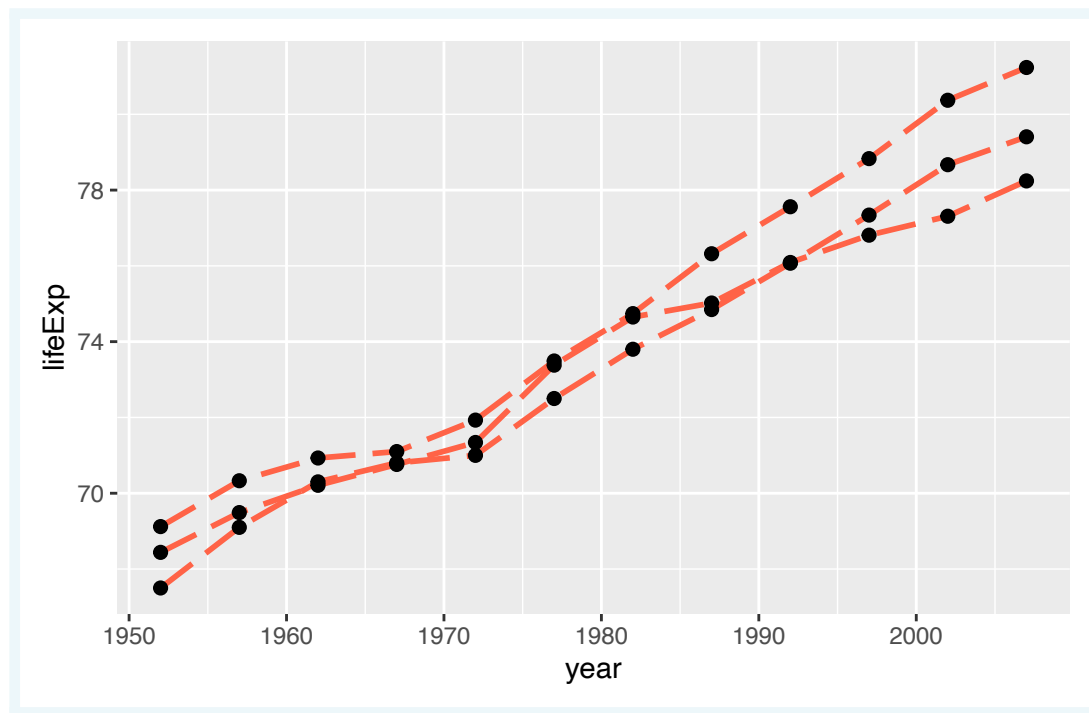
```
## Line graph with grouping by a categorical variable
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     group = country)) +
  geom_line() +
  geom_point()
```



Now that the data is grouped by country, we have 3 separate lines - one for each level of the country variable.

We can also apply fixed aesthetics to the geometric layers.

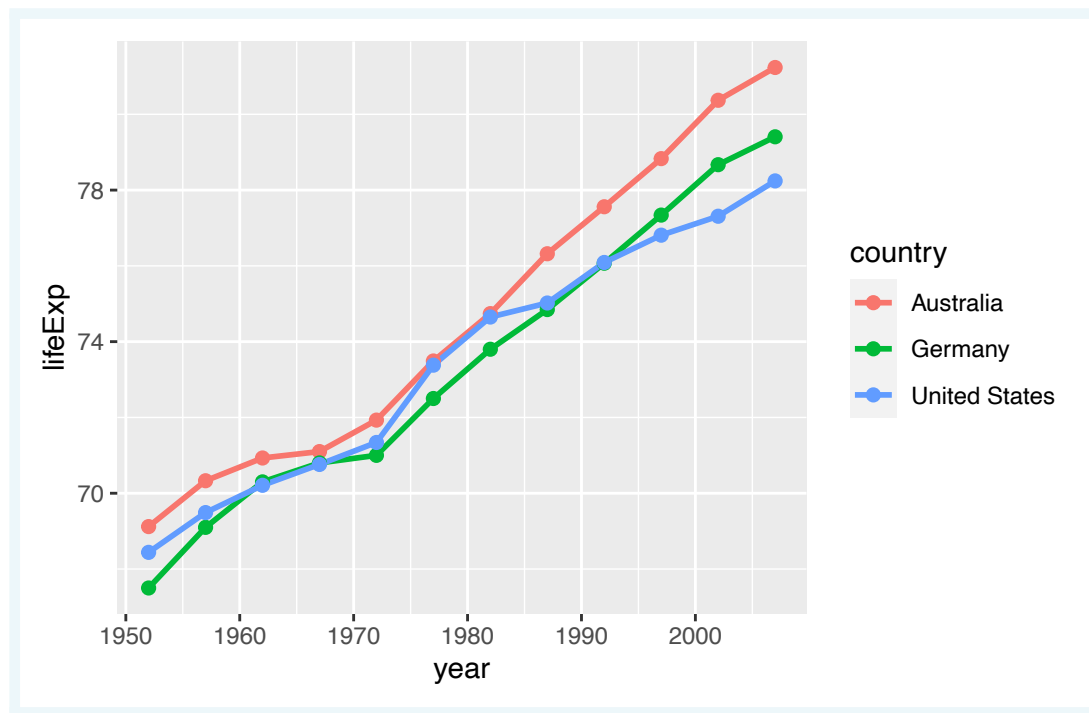
```
## Applying fixed aesthetics to multiple lines
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     group = country)) +
  geom_line(linetype="longdash",      # set line type
            color="tomato",           # set line color
            size=1) +                 # set line size
  geom_point(size = 2)                # set point size
```



In the graphs above, line types, colors and sizes are the same for the three groups.

This doesn't tell us which is which though. We should add an aesthetic mapping that can help us identify which line belongs to which country, like color or line type.

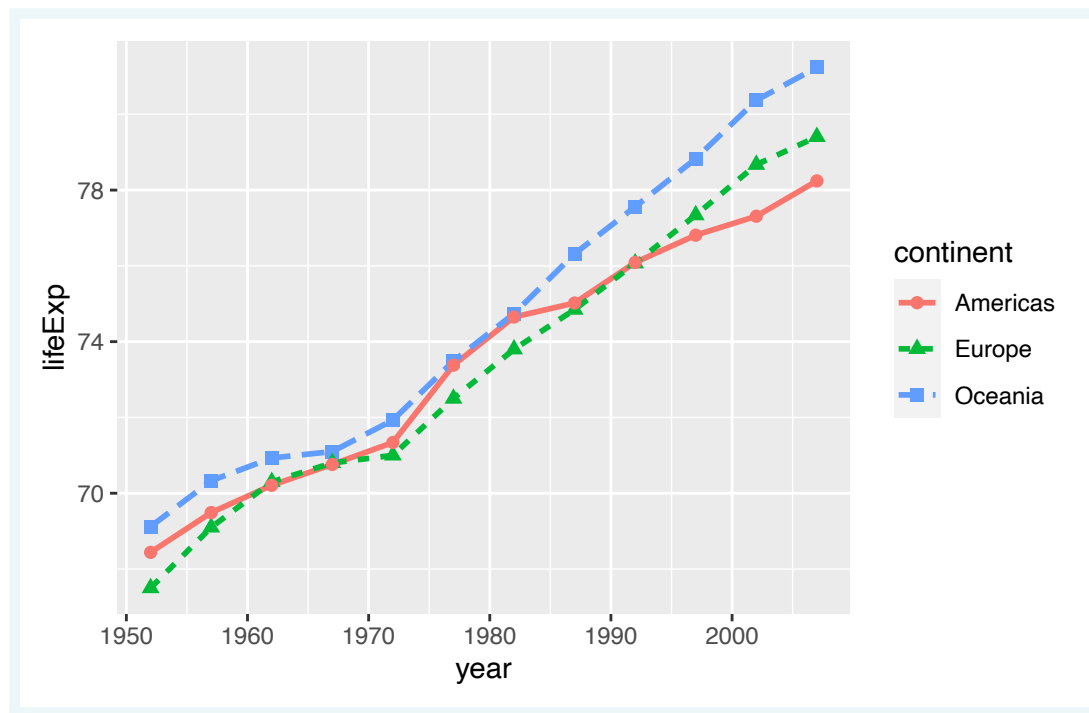
```
## Map country to color
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp, x = year,
                     group = country,
                     color = country)) +
  geom_line(size = 1) +
  geom_point(size = 2)
```



Aesthetic mappings specified within `ggplot()` function call are passed down to subsequent layers.

Instead of grouping by country, we can also group by continent:

```
## Map continent to color, line type, and shape
ggplot(data = gap_mini,
       mapping = aes(x = year,
                     y = lifeExp,
                     color = continent,
                     lty = continent,
                     shape = continent)) +
  geom_line(size = 1) +
  geom_point(size = 2)
```



When given multiple mappings and geoms, {ggplot2} can discern which mappings apply to which geoms.

Here color was inherited by both points and lines, but `lty` was ignored by `geom_point()` and shape was ignored by `geom_line()`, since they don't apply.

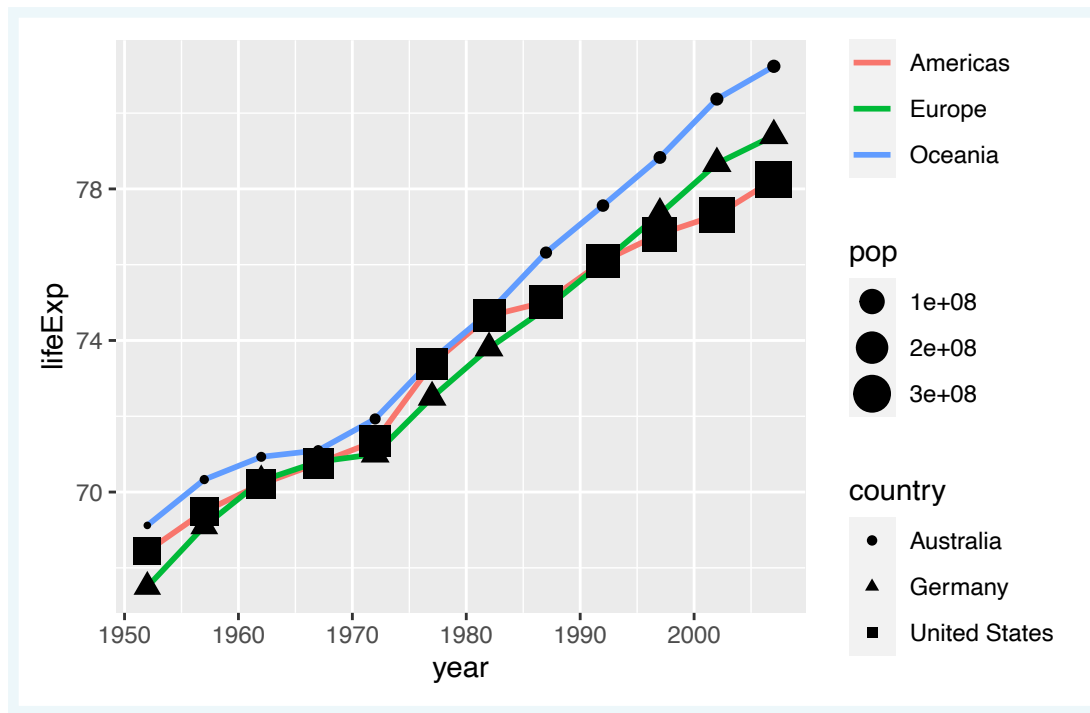
Challenge

Challenge

Mappings can either go in the `ggplot()` function or in `geom_*()` layer.

For example, aesthetic mappings can go in `geom_line()` and will only be applied to that layer:

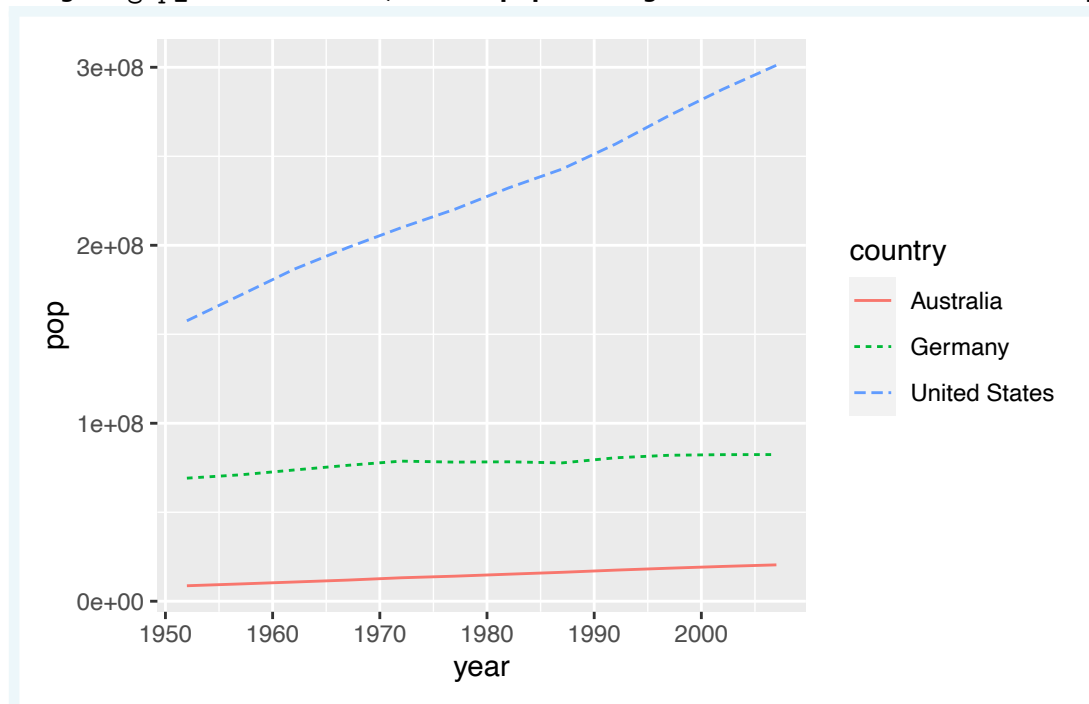
```
ggplot(data = gap_mini,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1, mapping = aes(color = continent)) +
  geom_point(mapping = aes(shape = country,
                          size = pop))
```



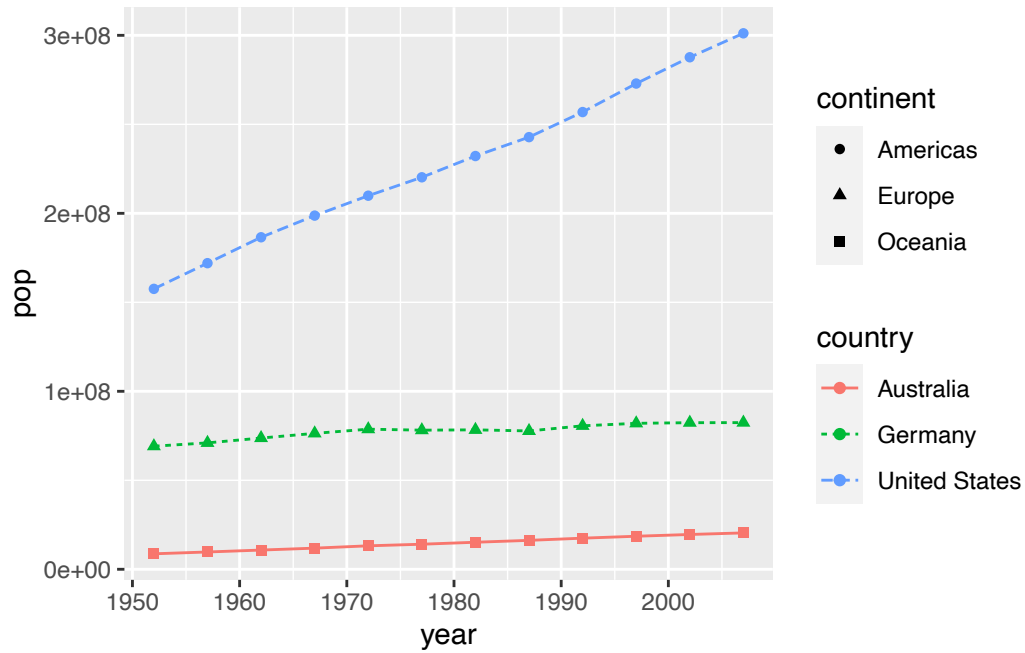
Try adding `mapping = aes()` in `geom_point()` and map continent to any valid aesthetic!

Practice

Using the `gap_mini` data frame, create a **population** growth chart with these aesthetic mappings:



Next, add a layer of points to the previous plot, and add the required aesthetic mappings to produce a plot that looks like this:

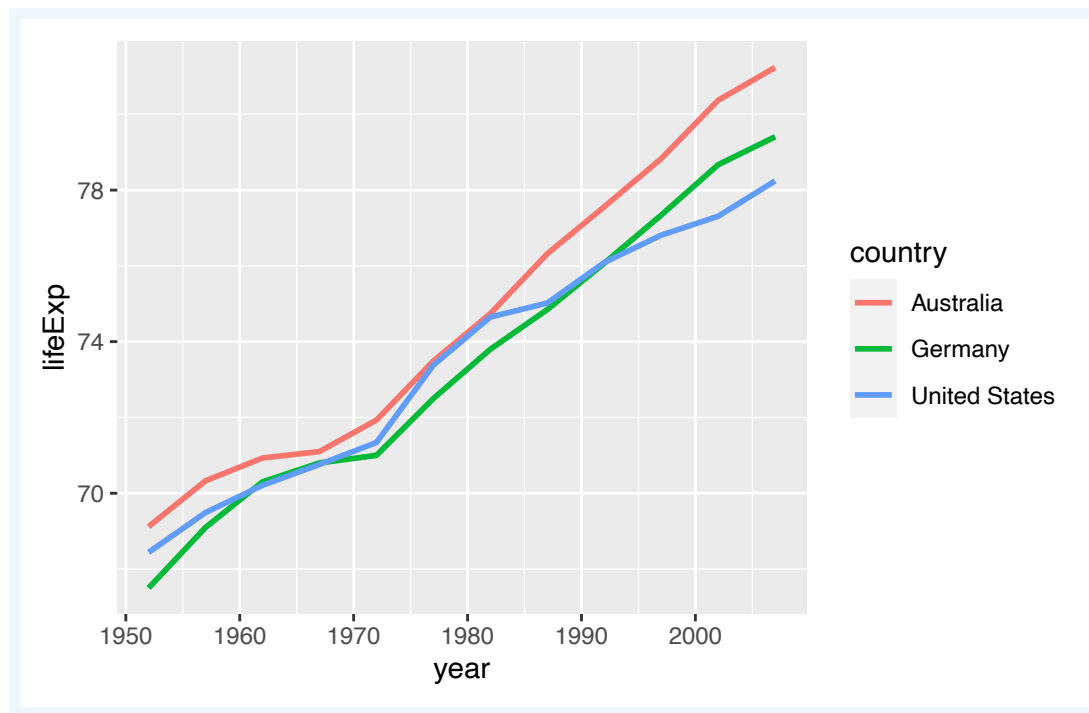


Don't worry about any fixed aesthetics, just make sure the mapping of data variables is the same.

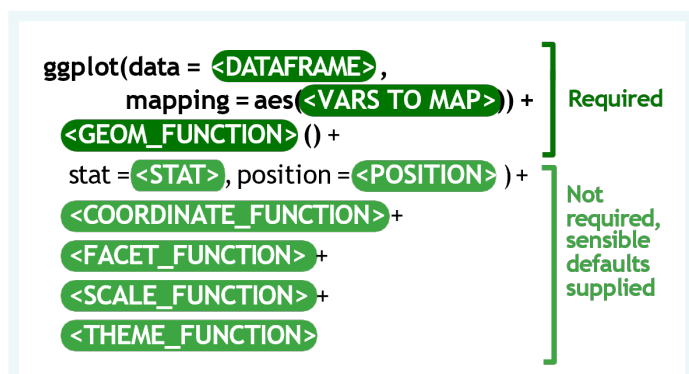
20.8 Modifying continuous x & y scales

{ggplot2} automatically scales variables to an aesthetic mapping according to type of variable it's given.

```
## Automatic scaling for x, y, and color
ggplot(data = gap_mini,
       mapping = aes(x = year,
                     y = lifeExp,
                     color = country)) +
  geom_line(size = 1)
```



In some cases the we might want to transform the axis scaling for better visualization. We can customize these scales with the `scale_*()` family of functions.



`scale_x_continuous()` and `scale_y_continuous()` are the default scale functions for continuous x and y aesthetics.

GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - map cont' values to visual ones

scale_*_discrete() - map discrete values to visual ones

scale_*_identity() - use data values as visual ones

scale_*_manual(values = c()) - map discrete values to manually chosen visual ones

scale_*_date(date_labels = "%m/%d"), date_breaks = "2 weeks") - treat data values as dates.

scale_*_datetime() - treat data x values as date times. Use same arguments as `scale_x_date()`. See `?strptime` for label formats.

X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

scale_x_log10() - Plot x on log10 scale

scale_x_reverse() - Reverse direction of x axis

scale_x_sqrt() - Plot x on square root scale

COLOR AND FILL SCALES (CONTINUOUS)

```
o <- c + geom_dotplot(aes(fill = ..x..))
```

```
o + scale_fill_distiller(palette = "Blues")
```

```
o + scale_fill_gradient(low="red", high="yellow")
```


```
o + scale_fill_gradient2(low="red", high="blue",
mid = "white", midpoint = 25)
```

```
o + scale_fill_gradientn(colours=topo.colors(6))
```

Also: `rainbow()`, `heat.colors()`, `terrain.colors()`,
`cm.colors()`, `RColorBrewer::brewer.pal()`

SHAPE AND SIZE SCALES

```
p <- e + geom_point(aes(shape = fl, size = cyl))
```

 `p + scale_shape() + scale_size()`

p + scale_shape_manual(values = c(3:7))

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

□○△+×◇▽□*◇⊕⊗⊗⊗⊗⊗□○△◇○●○□◇△▽

```
p + scale_radius(range = c(1,6))
p + scale_size_area(max_size =
```

`p + scale_size_area(max_size = 6)`

20.8.1 Scale breaks

Let's create a new subset of countries from `gapminder`, and this time we will plot changes in GDP over time.

```
## Data subset to include India, China, and Thailand
gap_mini2 <- filter(gapminder,
                     country %in% c("India",
                                     "China",
                                     "Thailand"))

gap_mini2
```

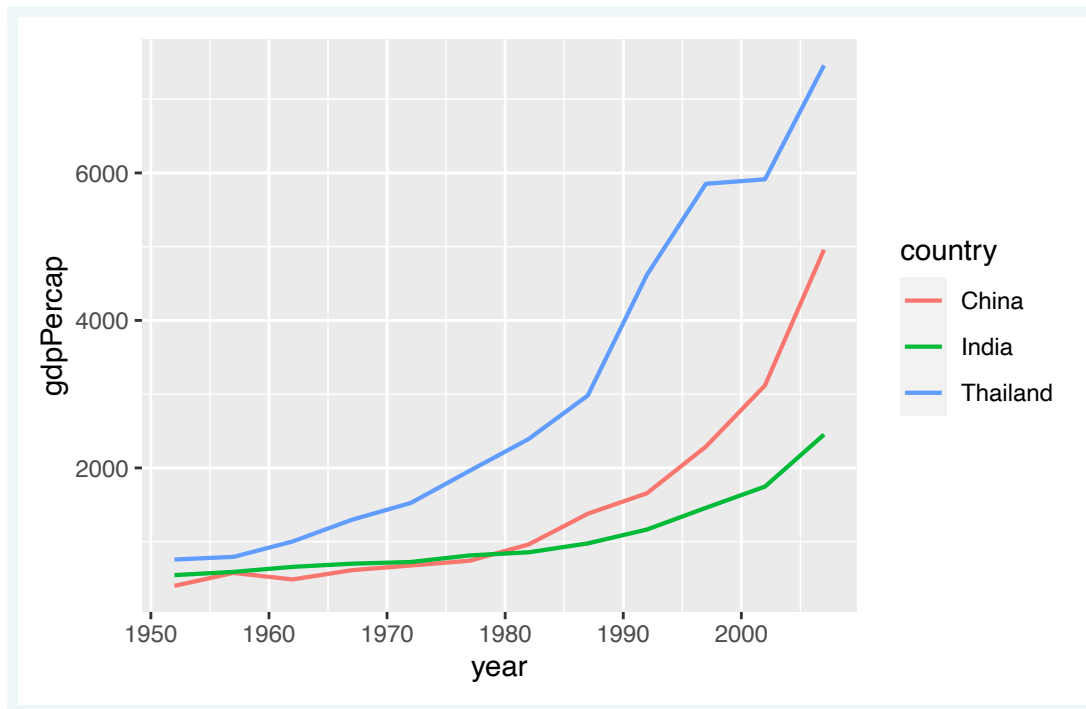
```
# A tibble: 10 x 6
```

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	China	Asia	1952	44	556263527	400.
2	China	Asia	1957	50.5	637408000	576.
3	China	Asia	1962	44.5	665770000	488.
4	China	Asia	1967	58.4	754550000	613.
5	China	Asia	1972	63.1	862030000	677.
6	China	Asia	1977	64.0	943455000	741.
7	China	Asia	1982	65.5	1000281000	962.
8	China	Asia	1987	67.3	1084035000	1379.
9	China	Asia	1992	68.7	1164970000	1656.
10	China	Asia	1997	70.4	1230075000	2289.

Here we will change the y-axis mapping from `lifeExp` to `gdpPerCap`:

```
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                     y = gdpPercap,
                     group = country,
```

```
color = country)) +  
geom_line(size = 0.75)
```



The x-axis labels for year in don't match up with the dataset.

```
gap_mini2$year %>% unique()
```

```
[1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

We can specify exactly where to label the axis by providing a numeric vector.

```
## You can manually enter scale breaks (don't do this)  
c(1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, 2002, 2007)
```

```
[1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

```
## It's better to create the vector with seq()  
seq(from = 1952, to = 2007, by = 5)
```

```
[1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

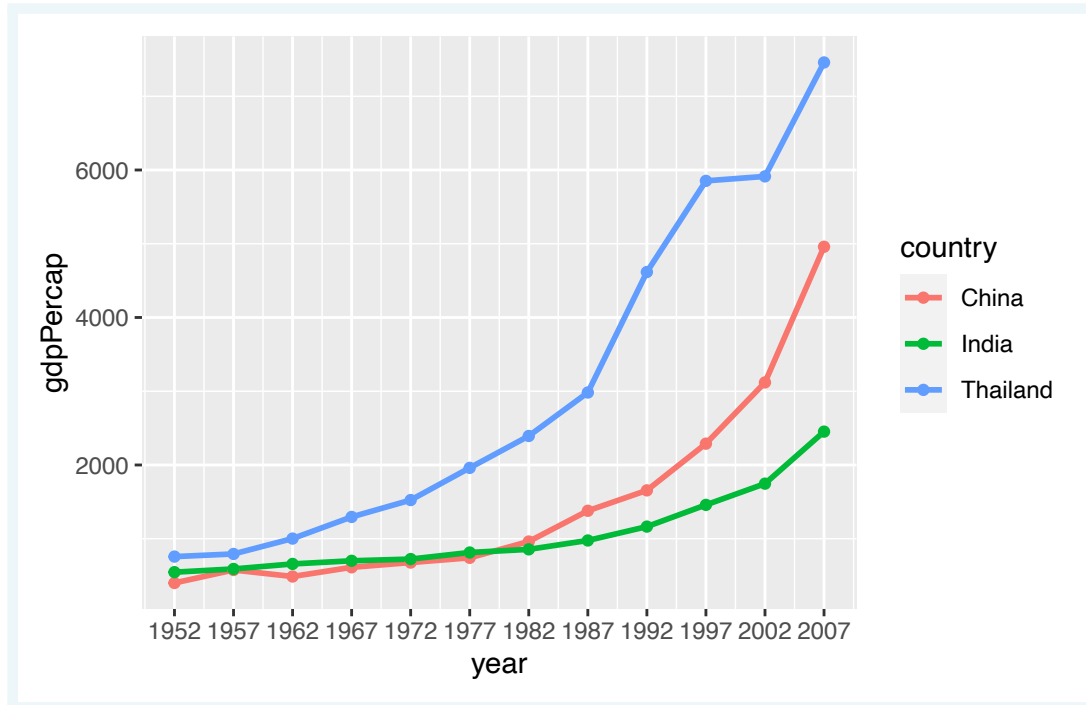
Use `scale_x_continuous` to make the axis breaks match up with the dataset:

```
## Customize x-axis breaks with `scale_x_continuous(breaks = VECTOR)`  
ggplot(data = gap_mini2,  
mapping = aes(x = year,
```

```

      y = gdpPercap,
      color = country)) +
geom_line(size = 1) +
scale_x_continuous(breaks = seq(from = 1952, to = 2007, by = 5)) +
geom_point()

```



Store scale break values as an R object for easier reference:

```

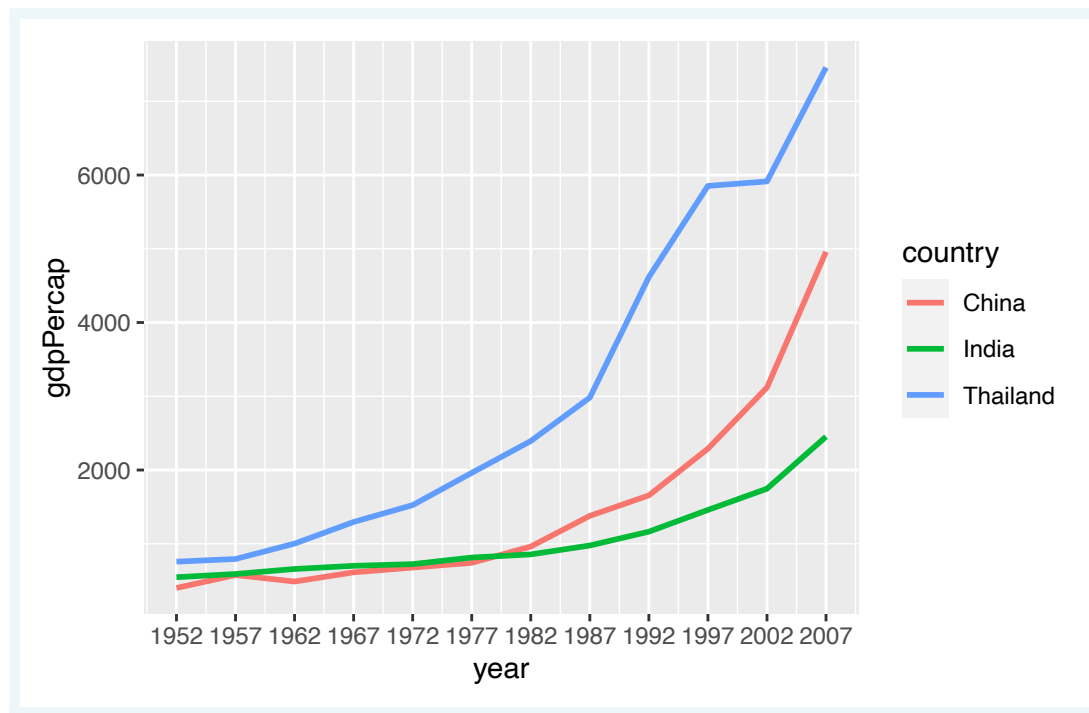
## Store numeric vector to a named object
gap_years <- seq(from = 1952, to = 2007, by = 5)

```

```

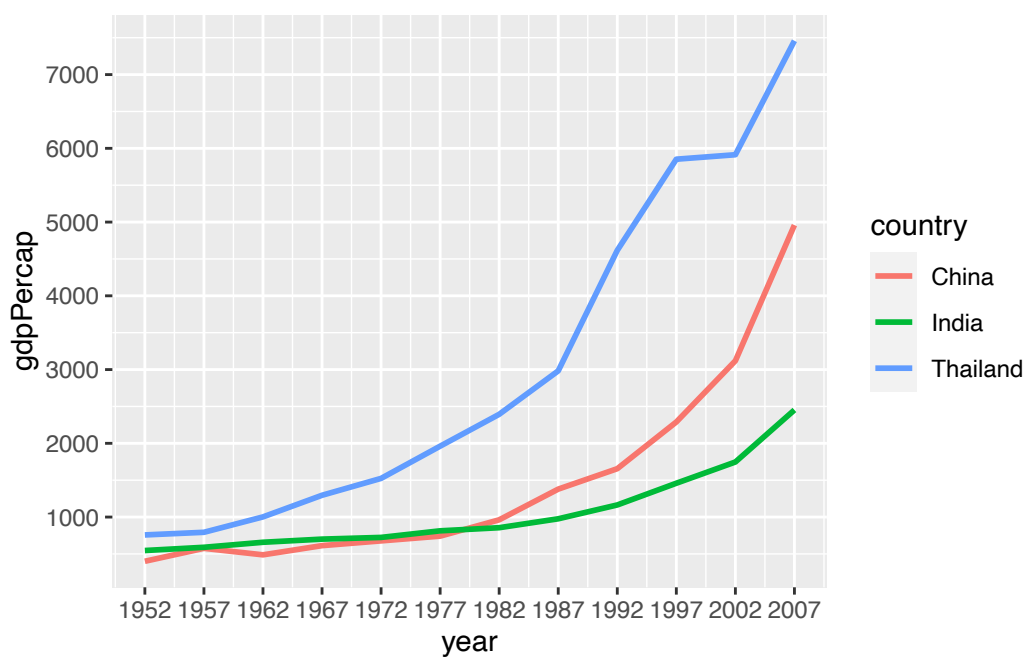
## Replace seq() code with named vector
ggplot(data = gap_mini2,
      mapping = aes(x = year,
                    y = gdpPercap,
                    color = country)) +
geom_line(size = 1) +
scale_x_continuous(breaks = gap_years)

```



Practice

We can customize scale breaks on a continuous **y-axis** values with `scale_y_continuous()`. Copy the code from the last example, and add `scale_y_continuous()` to add the following y-axis breaks:



20.8.2 Logarithmic scaling

In the last two mini sets, I chose three countries that had similar range of GDP or life expectancy for good scaling and readability so that we can make out these changes.

But if we add a country to the group that significantly differs, default scaling is not so great.

We'll look at an example plot where you may want to rescale the axes from linear to a log scale.

Let's add New Zealand to the previous set of countries and create `gap_mini3`:

```
## Data subset to include India, China, Thailand, and New Zealand
gap_mini3 <- filter(gapminder,
                    country %in% c("India",
                                   "China",
                                   "Thailand",
                                   "New Zealand"))

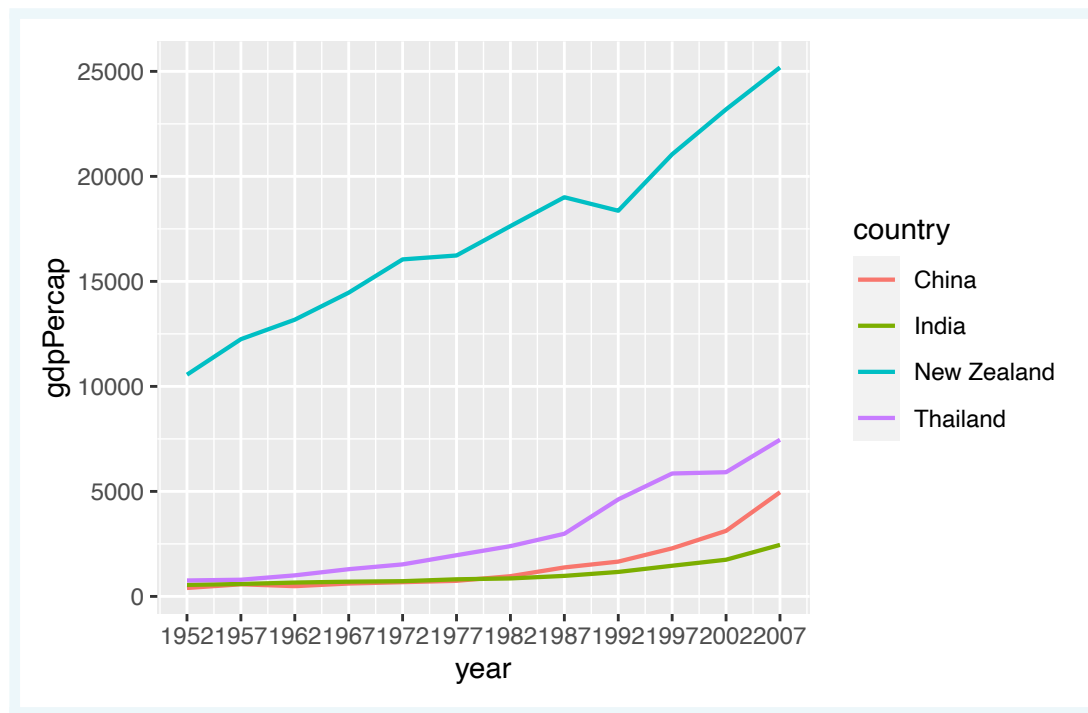
gap_mini3
```

A tibble: 10 x 6

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	China	Asia	1952	44	556263527	400.
2	China	Asia	1957	50.5	637408000	576.
3	China	Asia	1962	44.5	665770000	488.
4	China	Asia	1967	58.4	754550000	613.
5	China	Asia	1972	63.1	862030000	677.
6	China	Asia	1977	64.0	943455000	741.
7	China	Asia	1982	65.5	1000281000	962.
8	China	Asia	1987	67.3	1084035000	1379.
9	China	Asia	1992	68.7	1164970000	1656.
10	China	Asia	1997	70.4	1230075000	2289.

Now we will recreate the plot of GDP over time with the new data subset:

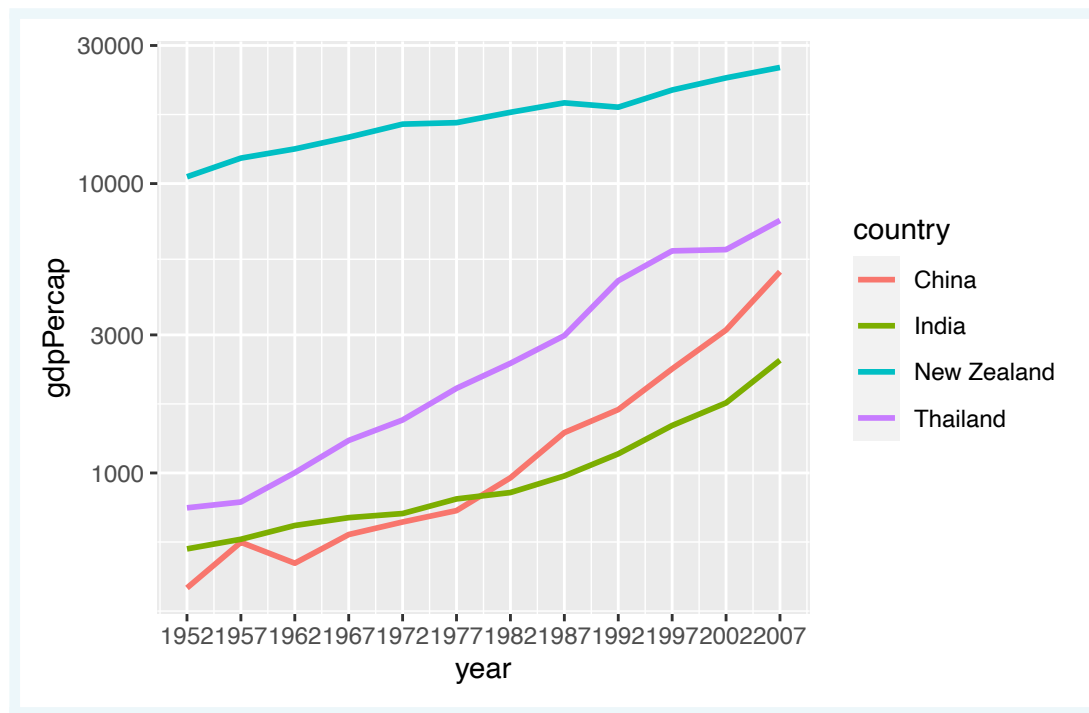
```
ggplot(data = gap_mini3,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 0.75) +
  scale_x_continuous(breaks = gap_years)
```



The curves for India and China show an exponential increase in GDP per capita. However, the y-axis values for these two countries are much lower than that of New Zealand, so the lines are a bit squashed together. This makes the data hard to read. Additionally, the large empty area in the middle is not a great use of plot space.

We can address this by log-transforming the y-axis using `scale_y_log10()`, which log-scales the y-axis (as the name suggests). We will add this function as a new layer after a `+` sign, as usual:

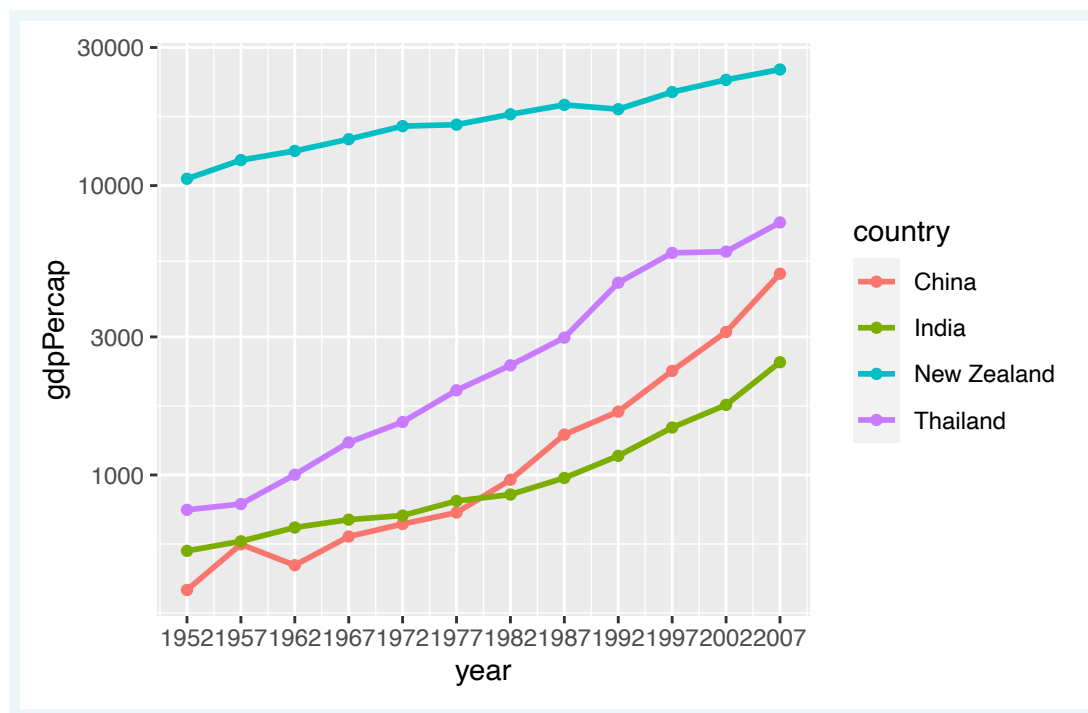
```
## Add scale_y_log10()
ggplot(data = gap_mini3,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 1) +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10()
```



Now the y-axis values are rescaled, and the scale break labels tell us that it is nonlinear.

We can add a layer of points to make this clearer:

```
ggplot(data = gap_mini3,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 1) +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10() +
  geom_point()
```



Practice

First subset `gapminder` to only the rows containing data for **Uganda**:

Now, use `gap_Uganda` to create a time series plot of population (`pop`) over time (`year`). Transform the y axis to a log scale, edit the scale breaks to `gap_years`, change the line color to `forestgreen` and the size to 1mm.

Next, we can change the text of the axis labels to be more descriptive, as well as add titles, subtitles, and other informative text to the plot.

20.9 Labeling with labs()

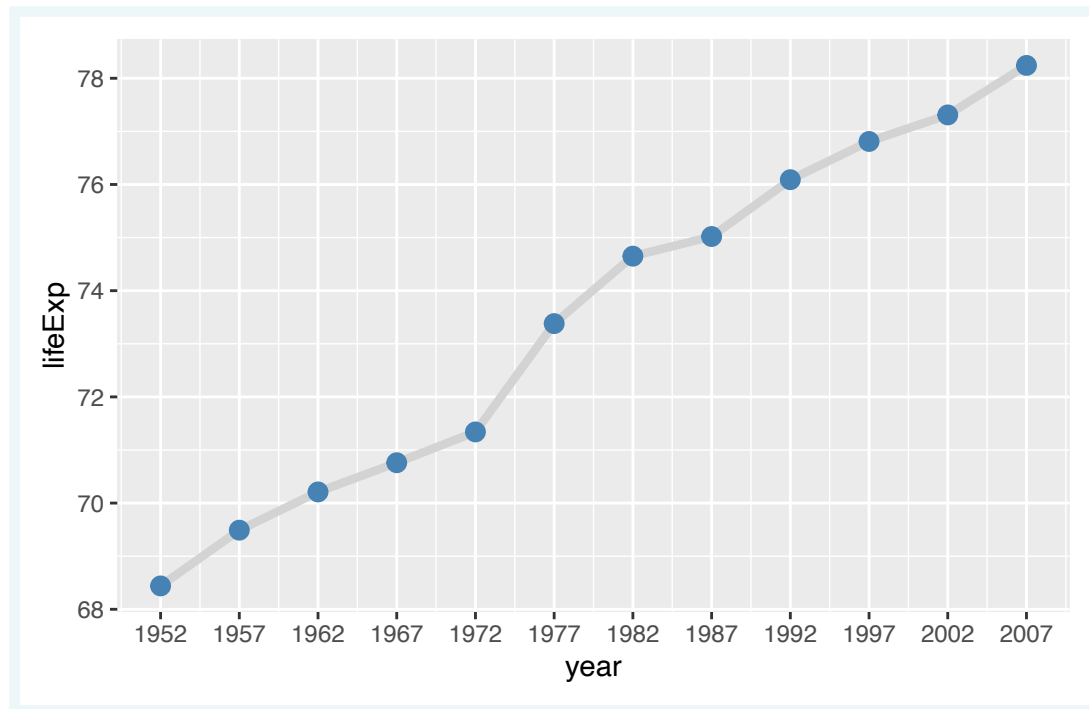
You can add labels to a plot with the `labs()` function. Arguments we can specify with the `labs()` function include:

- `title`: Change or add a title
- `subtitle`: Add subtitle below the title
- `x`: Rename x-axis
- `y`: Rename y-axis
- `caption`: Add caption below the graph

Let's start with this plot and start adding labels to it:

```
## Time series plot of life expectancy in the United States
ggplot(data = gap_US,
        mapping = aes(x = year,
                       y = lifeExp)) +
  geom_line(size = 1.5,
            color = "lightgrey") +
  geom_point(size = 3,
```

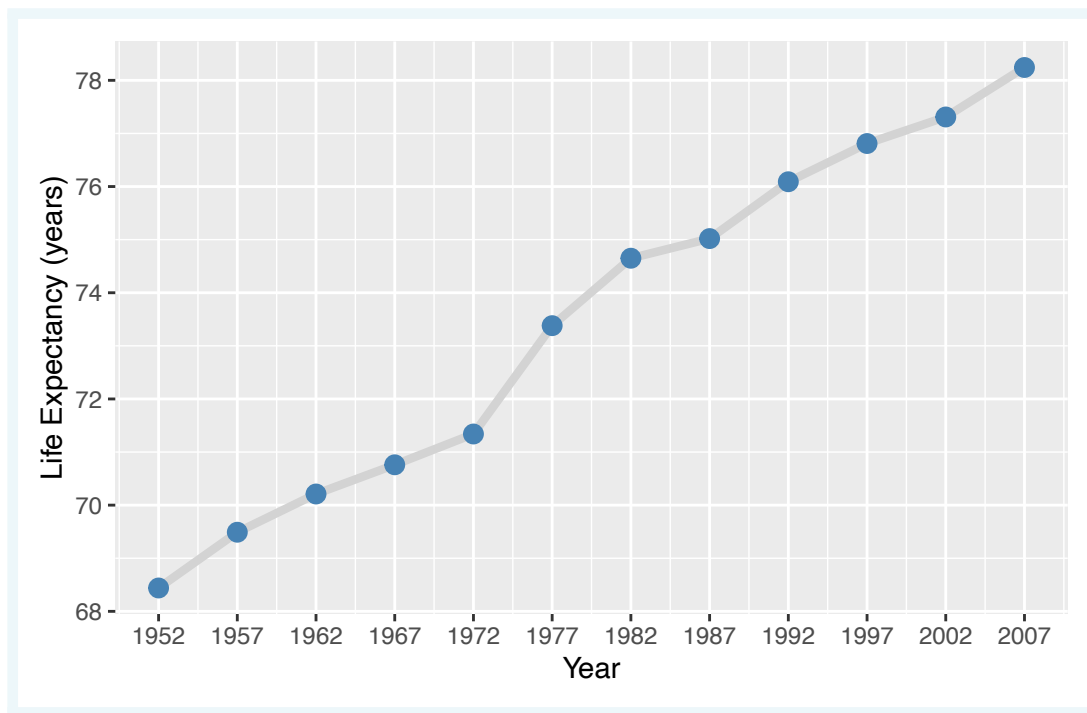
```
color = "steelblue") +
scale_x_continuous(breaks = gap_years)
```



We add the `labs()` to our code using a `+` sign.

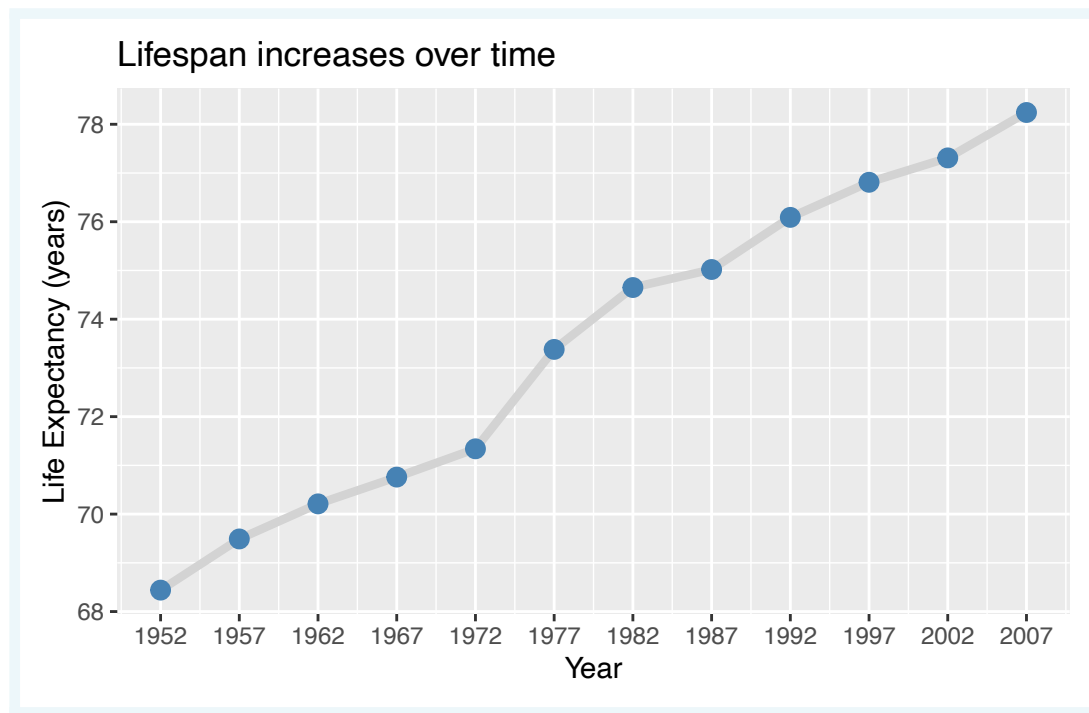
First we will add the `x` and `y` arguments to `labs()`, and change the axis titles from the default (variable name) to something more informative.

```
## Rename axis titles
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1.5,
           color = "lightgrey") +
  geom_point(size = 3,
            color = "steelblue") +
  scale_x_continuous(breaks = gap_years) +
  labs(x = "Year",
       y = "Life Expectancy (years)")
```



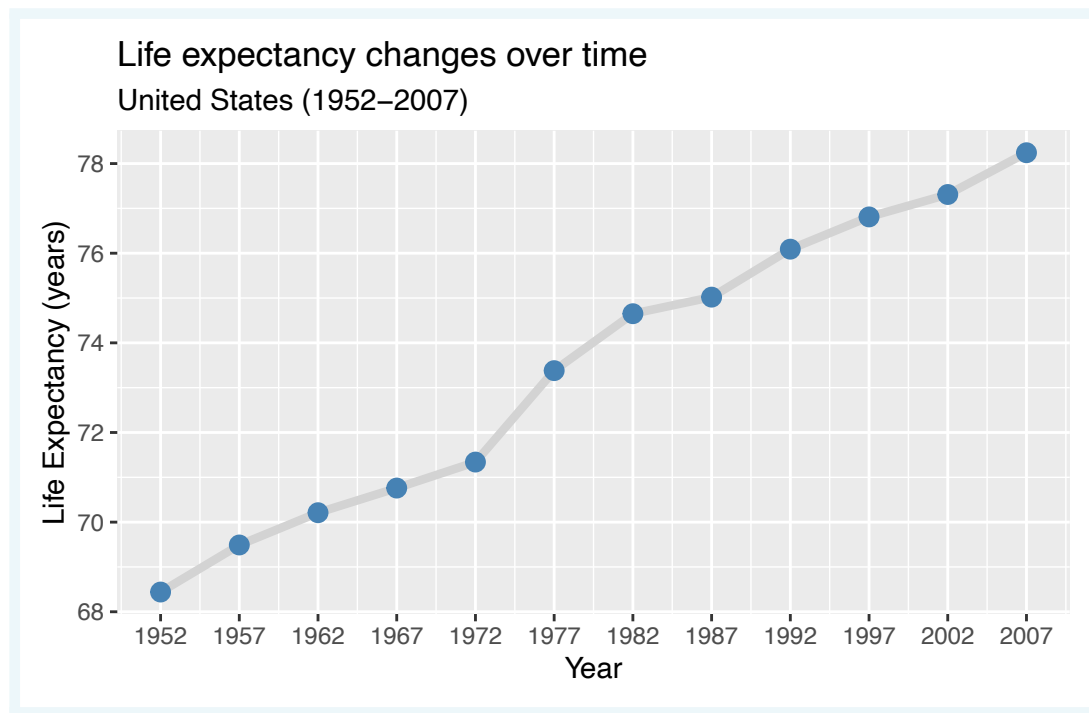
Next we supply a character string to the title argument to add large text above the plot.

```
## Add main title: "Lifespan increases over time"
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1.5,
           color = "lightgrey") +
  geom_point(size = 3,
            color = "steelblue") +
  scale_x_continuous(breaks = gap_years) +
  labs(x = "Year",
       y = "Life Expectancy (years)",
       title = "Lifespan increases over time")
```



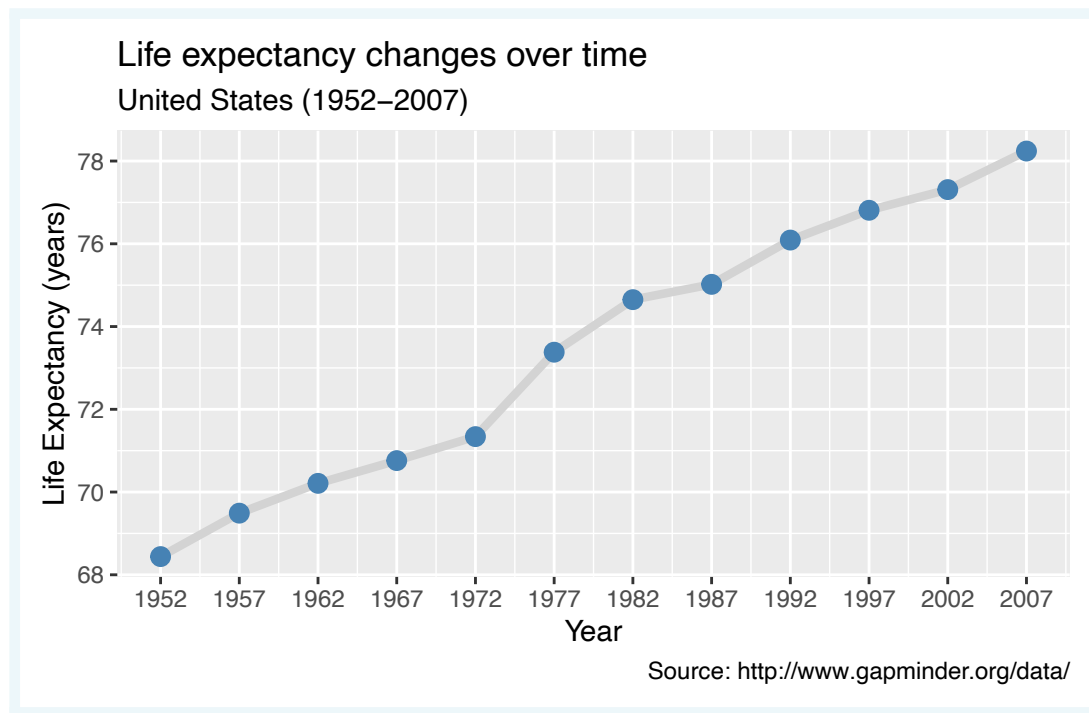
The subtitle argument adds smaller text below the main title.

```
## Add subtitle with location and time frame
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1.5,
           color = "lightgrey") +
  geom_point(size = 3,
            color = "steelblue") +
  scale_x_continuous(breaks = gap_years) +
  labs(x = "Year",
       y = "Life Expectancy (years)",
       title = "Life expectancy changes over time",
       subtitle = "United States (1952-2007)")
```



Finally, we can supply the caption argument to add small text to the bottom-right corner below the plot.

```
## Add caption with data source: "Source: www.gapminder.org/data"
ggplot(data = gap_US,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1.5,
            color = "lightgrey") +
  geom_point(size = 3,
             color = "steelblue") +
  scale_x_continuous(breaks = gap_years) +
  labs(x = "Year",
       y = "Life Expectancy (years)",
       title = "Life expectancy changes over time",
       subtitle = "United States (1952–2007)",
       caption = "Source: http://www.gapminder.org/data/")
```

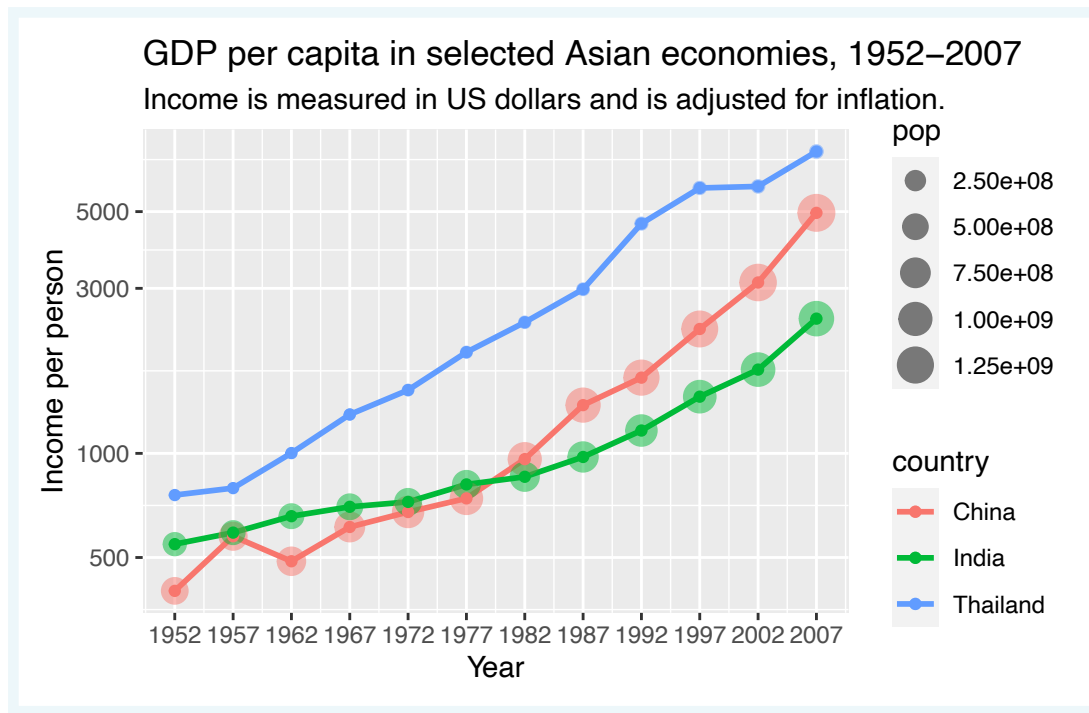


i Challenge

When you use an aesthetic mapping (e.g., color, size), {ggplot2} automatically scales the given aesthetic to match the data and adds a legend.

Here is an updated version of the `gap_mini3` plot we made before. We are changing the of points *and* lines by setting `aes(color = country)` in `ggplot()`. Then the size of *points* is scaled to the *pop* variable. See that `labs()` is used to change the title, subtitle, and axis labels.

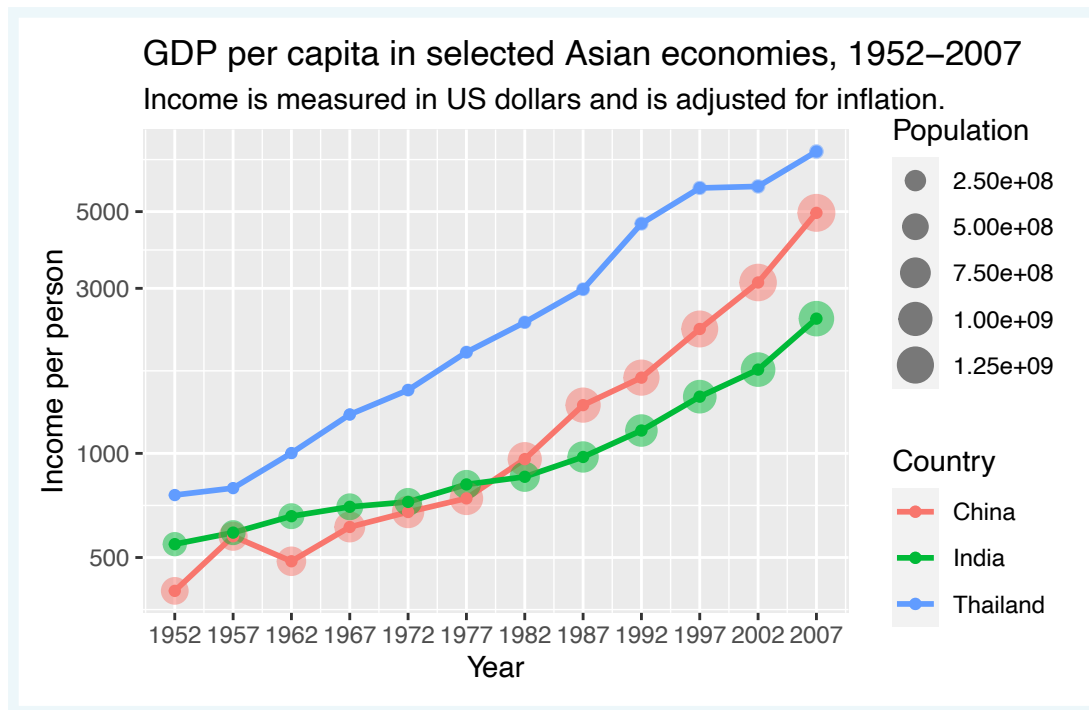
```
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 1) +
  geom_point(mapping = aes(size = pop),
            alpha = 0.5) +
  geom_point() +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10() +
  labs(x = "Year",
       y = "Income per person",
       title = "GDP per capita in selected Asian economies, 1952-2007",
       subtitle = "Income is measured in US dollars and is adjusted for inflation.")
```



The default title of a legend or key is the name of the data variable it corresponds to. Here the color legend is titled `country`, and the size legend is titled `pop`.

We can also edit these in `labs()` by setting `AES_NAME = "CUSTOM_TITLE"`.

```
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 1) +
  geom_point(mapping = aes(size = pop),
            alpha = 0.5) +
  geom_point() +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10() +
  labs(x = "Year",
       y = "Income per person",
       title = "GDP per capita in selected Asian economies, 1952–2007",
       subtitle = "Income is measured in US dollars and is adjusted for inflation.",
       color = "Country",
       size = "Population")
```



The same syntax can be used to edit legend titles for other aesthetic mappings. A common mistake is to use the variable name instead of the aesthetic name in `labs()`, so watch out for that!

Practice

Create a time series plot comparing the trends in life expectancy from 1952-2007 for **three countries** in the `gapminder` data frame.

First, subset the data to three countries of your choice:

Use `my_gap_mini` to create a plot with the following attributes:

- Add points to the line graph
- Color the lines and points by country
- Increase the width of lines to 1mm and the size of points to 2mm
- Make the lines 50% transparent
- Change the x-axis scale breaks to match years in dataset

Finally, add the following labels to your plot:

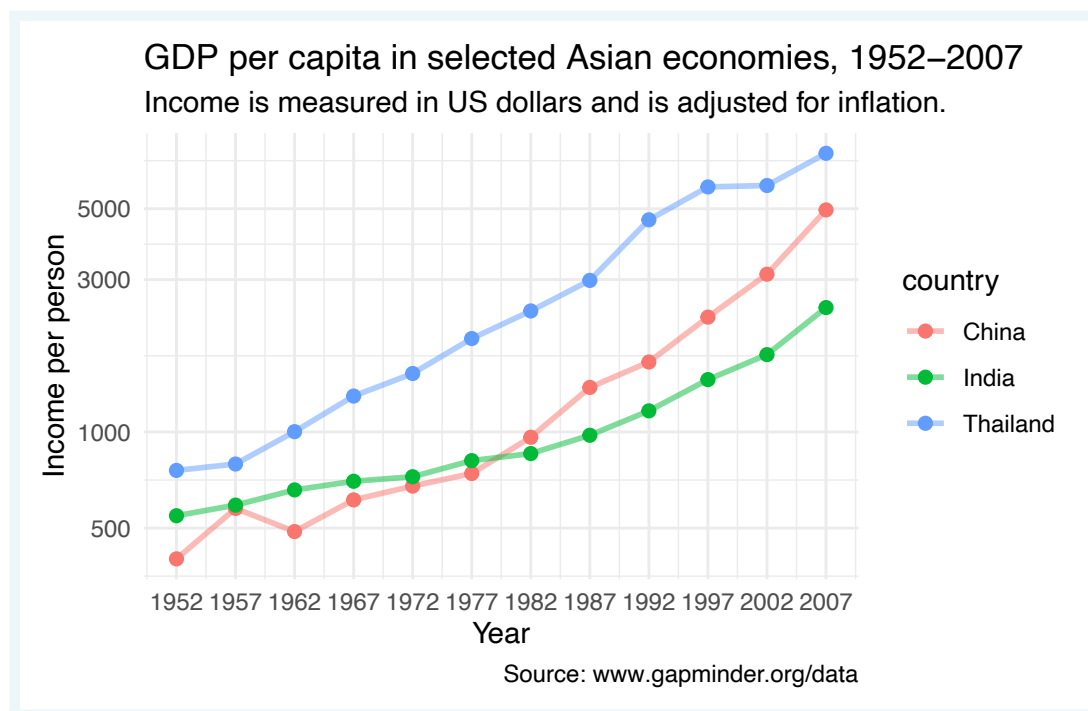
- Title: "Health & wealth of nations"
- Axis titles: "Longevity" and "Year"
- Capitalize legend title

(Note: subtitle requirement has been removed.)

20.10 Preview: Themes

In the next lesson, you will learn how to use theme functions.

```
## Use theme_minimal()
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(size = 1, alpha = 0.5) +
  geom_point(size = 2) +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10() +
  labs(x = "Year",
       y = "Income per person",
       title = "GDP per capita in selected Asian economies, 1952–2007",
       subtitle = "Income is measured in US dollars and is adjusted for inflation.",
       caption = "Source: www.gapminder.org/data") +
  theme_minimal()
```



20.11 Wrap up

Line graphs, just like scatterplots, display the relationship between two numerical variables. When one of the two variables represents time, a line graph can be a more effective method of displaying relationship. Therefore, it is preferred to use line graphs over scatterplots when the variable on the x-axis (i.e., the explanatory variable) has an inherent ordering, such as some notion of time, like the year variable of gapminder.

We can change scale breaks and transform scales to make plots easier to read, and label them to add more information.

Hope you found this lesson helpful!

References

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

- Ismay, Chester, and Albert Y. Kim. 2022. *A ModernDive into R and the Tidyverse*. <https://moderndive.com/>.
- Kabacoff, Rob. 2020. *Data Visualization with R*. <https://rkabacoff.github.io/datavis/>.
- https://www.rebeccabarter.com/blog/2017-11-17-ggplot2_tutorial/

20.12 Solutions

```
.SOLUTION_q1()
```

```
ggplot(gap_US,  
       mapping = aes(x = year,  
                     y = gdpPercap)) +  
  geom_line()
```

```
.SOLUTION_q2()
```

```
ggplot(gap_US,  
       mapping = aes(x = year,  
                     y = gdpPercap)) +  
  geom_line(lty = "dotted") +  
  geom_point(color = "aquamarine")
```

```
.SOLUTION_q3()
```

```
ggplot(gap_mini,  
       aes(x = year,  
           y = pop,  
           color = country,  
           linetype = country)) +  
  geom_line()
```

```
.SOLUTION_q4()
```

```
ggplot(gap_mini,  
       aes(x = year,  
           y = pop,  
           color = country,  
           shape = continent,  
           lty = country)) +  
  geom_line() +  
  geom_point()
```

`.SOLUTION_q5()`

```
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                     y = gdpPercap,
                     color = country)) +
  geom_line(linewidth = 1) +
  scale_x_continuous(breaks = gap_years) +
  scale_y_continuous(breaks = seq(from = 1000, to = 7000, by = 1000))
```

`.SOLUTION_q6()`

```
ggplot(data = gap_Uganda, mapping = aes(x = year, y = pop)) +
  geom_line(linewidth = 1, color = "forestgreen")+
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10()
```

`.SOLUTION_q7()`

```
ggplot(data = my_gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     color = country)) +
  geom_line(linewidth = 1, alpha = 0.5) +
  geom_point(size = 2) +
  scale_x_continuous(breaks = gap_years)
```

`.SOLUTION_q8()`

```
ggplot(data = my_gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     color = country)) +
  geom_line(linewidth = 1, alpha = 0.5) +
  geom_point(size = 2) +
  scale_x_continuous(breaks = gap_years) +
  labs(x = "Year",
       y = "Longevity",
       title = "Health & wealth of nations",
       color = "Color")
```

Chapter 21

Histograms with {ggplot2}

21.1 Histograms with {ggplot2}

21.2 Learning Objectives

By the end of this lesson, you will be able to:

1. Plot a histogram to visualize the distribution of continuous variables using `geom_histogram()`.
2. Adjust the number or size of bins on a histogram by with the `bins` or `binwidth` arguments.
3. Shift and align bins on a histogram with the `boundary` argument.
4. Set bin boundaries to a sequence of values with the `breaks` argument.

21.3 Introduction

A histogram is a plot that visualizes the *distribution* of a numerical value as follows:

1. We first cut up the x-axis into a series of *bins*, where each bin represents a range of values.
2. For each bin, we count the number of observations that fall in the range corresponding to that bin.
3. Then for each bin, we draw a bar whose height marks the corresponding count.

21.4 Packages

```
pacman::p_load(tidyverse,  
               here)
```

21.5 Childhood diarrheal diseases in Mali

We will visualize distributions of numerical variables in the `malidd` data frame, which we've seen in previous lessons.

```
## Import data from CSV  
malidd <- read_csv(here::here("data/clean/malidd.csv"))
```

i Recap

These data were collected as part of an observational study of acute diarrhea in children aged 0-59 months. The study was conducted in Mali and in early 2020. The dataset records demographic and clinical information for 150 patients.

```
## View first few rows of the data frame  
head(malidd)
```

n	admit_date	sex	age_months	height_cm	muac_cm	breastfeed...	vomit	fever	bloody_stool
1	2020-01-16	M	5	61.2	11.3	1	0	0	0
2	2020-01-17	F	12	70.6	13.2	1	1	1	0
3	2020-01-17	M	11	71.1	13.5	1	1	0	0
4	2020-01-17	M	9	68.5	12.6	1	0	0	0
5	2020-01-21	F	16	78.7	14.2	1	1	0	0
6	2020-01-21	M	6	67.7	14.5	1	0	0	0

The dataframe has 21 variables, many of which are continuous, like `height_cm`, `viral_load`, and `freqrespi`.

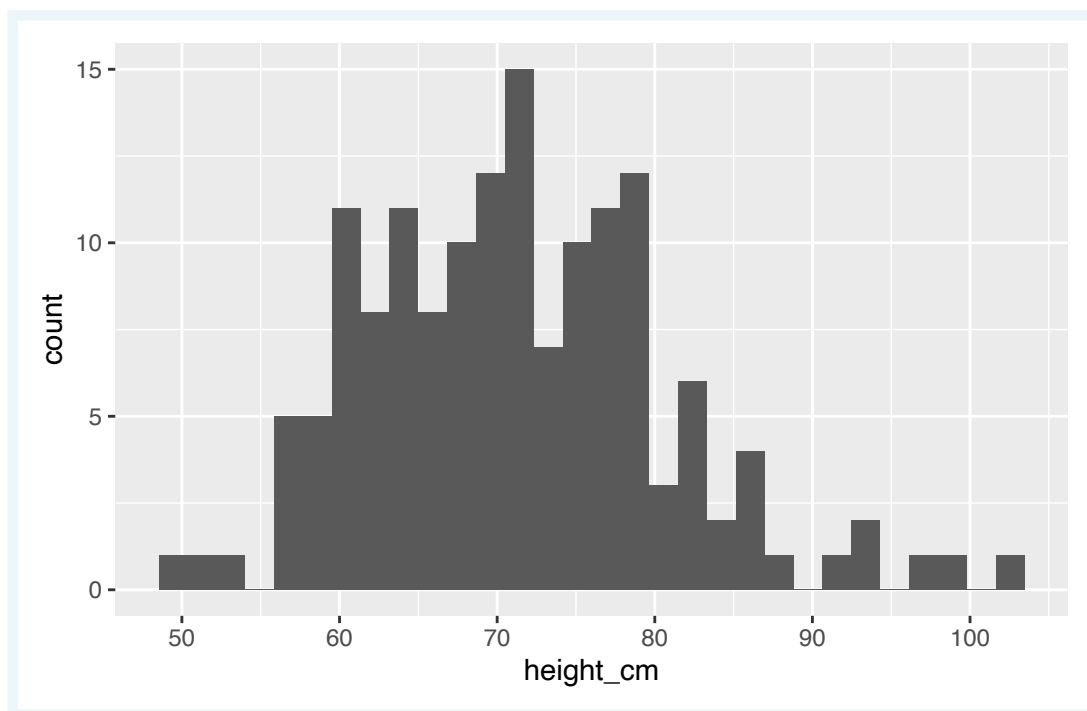
21.6 Basic histograms with `geom_histogram()`

Now let's use {ggplot2} to plot the distribution of childrens' heights, which is recorded in the `height_cm` column of `malidd`.

The `geom_*()` function used for histograms is `geom_histogram()`

```
## Simple histogram showing the distribution of height_cm
ggplot(data = malidd,
       mapping = aes(x = height_cm)) +
  geom_histogram()
```

``stat_bin()`` using ``bins = 30``. Pick better value with ``binwidth``.



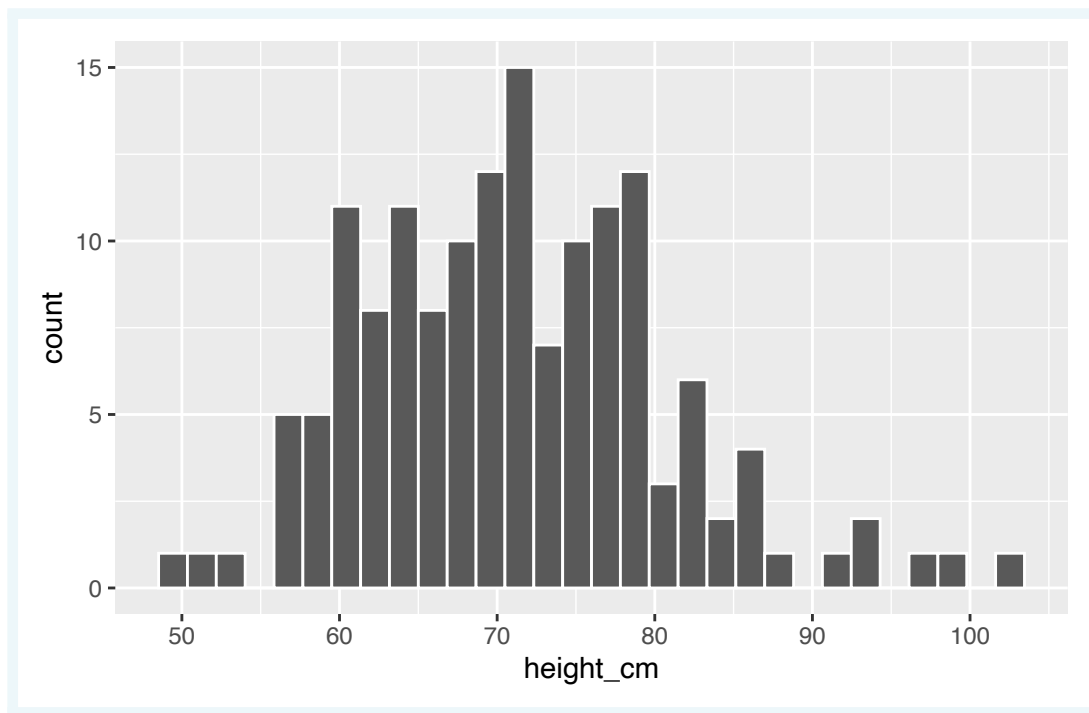
i Side Note

If we don't adjust the bins in `geom_histogram()`, we get a warning message. You can ignore this warning message for now, and will learn how to customize bins in the next section.

In the previous histogram, it's hard to where the boundaries for each bin start and end since everything is one big amorphous blob. So let's add borders around the bins:

```
## Set border color to "white"
ggplot(data = malidd ,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white")
```

``stat_bin()`` using ``bins = 30``. Pick better value with ``binwidth``.

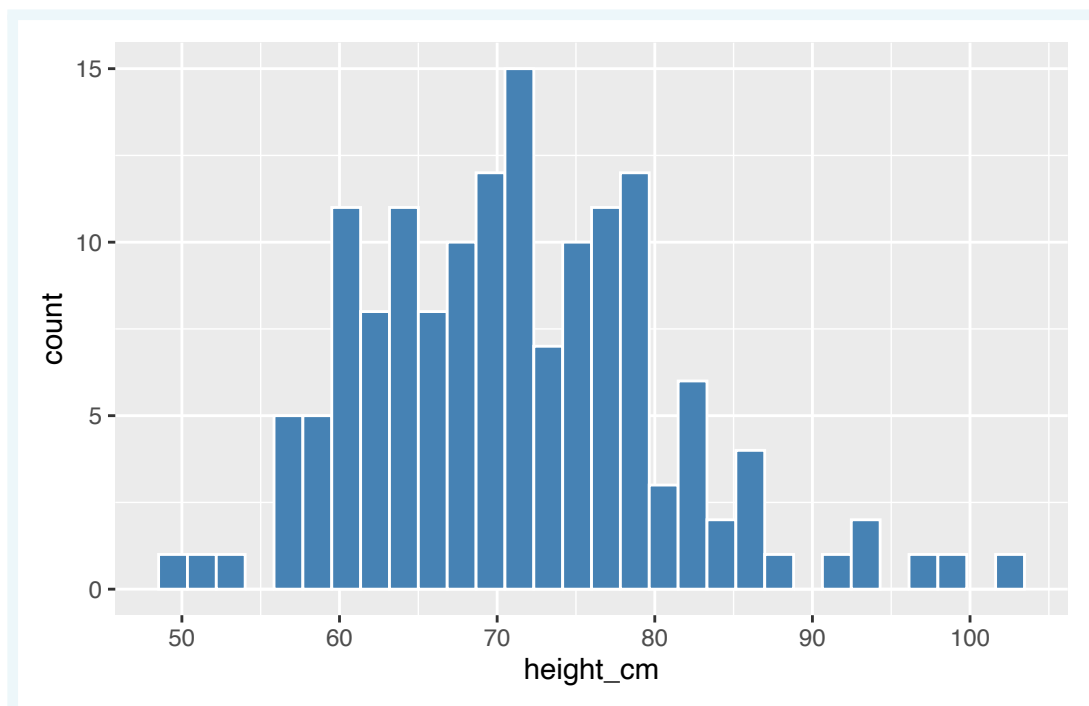


We now have an easier time associating ranges of cases to each of the bins.

We can also vary the color of the bars by setting the `fill` argument:

```
## Set fill color to "steelblue"
ggplot(data = malidd ,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white",
                fill = "steelblue")
```

``stat_bin()`` using ``bins = 30``. Pick better value with ``binwidth``.



Now that we can see the bars more clearly, let's unpack the resulting histogram. Some questions we might want to answer are:

1. What are the smallest and largest values?
2. What is the "center" or "most typical" value?
3. How do the values spread out?
4. What are frequent and infrequent values?

We can see that heights range from 50 to 105cm. The center is around 70cm, most patients fall in the 60-80cm range, with very few below 55cm or above 90cm. Observe that the histogram has a bell shape, meaning that the variable has a *normal distribution* (more or less).

Practice

- Plot a histogram showing the distribution of age (age_months) in malidd. Make the borders and fill of the bars "seagreen", and reduce opacity to 40%.
- Building on your code for the previous plot, modify the axis titles to "Age (months)" and "Number of children", respectively.

21.7 Adjusting bins in a histogram

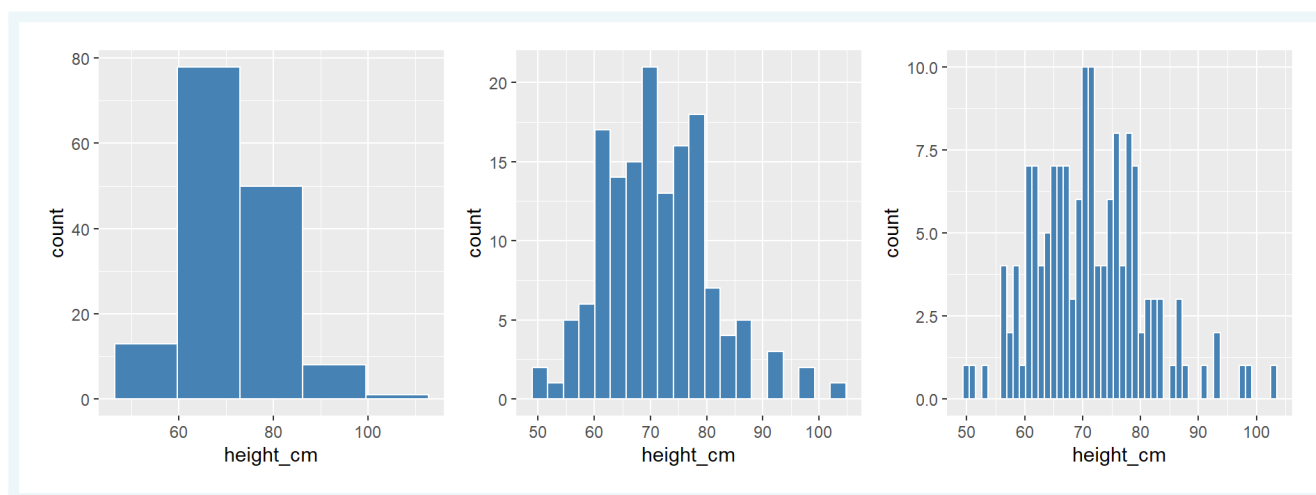
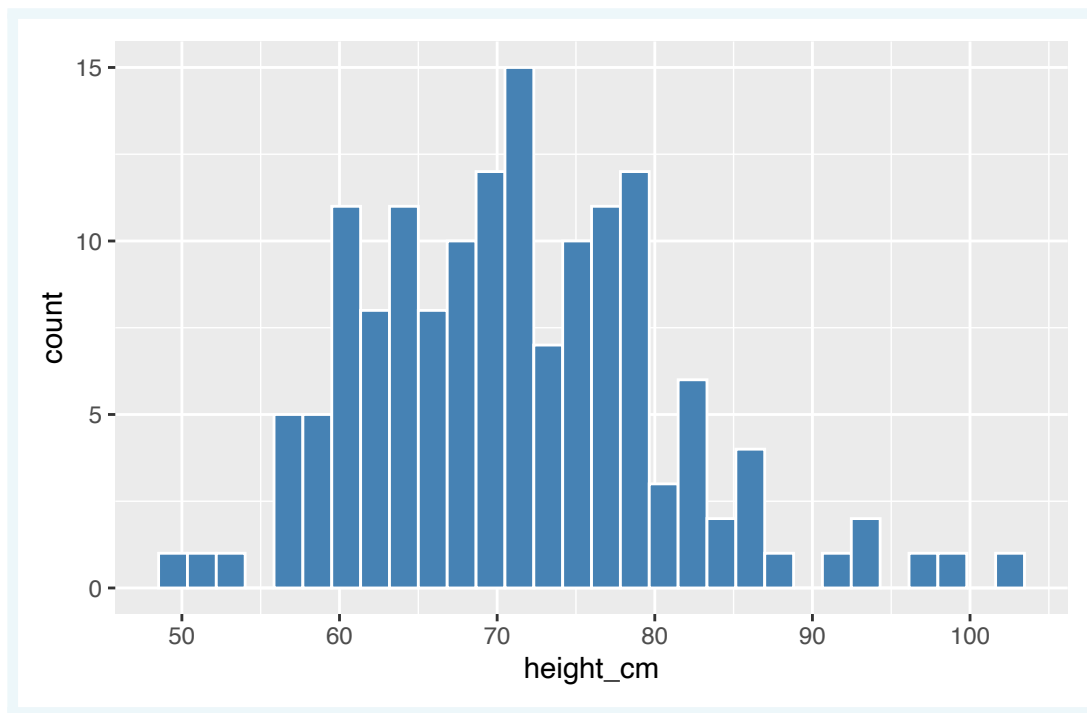


Figure 21.1: Histograms plotting the same variable with different bin settings.

After running code in previous examples, we got a histogram as well as a warning message about bins and bin width. The warning message is telling us that the histogram was constructed using `bins = 30` for 30 equally spaced bins.

```
## Warning message tells us to change the default of 30 bins
ggplot(data = malidd ,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white",
                fill = "steelblue")
```

``stat_bin()`` using ``bins = 30``. Pick better value with ``binwidth``.



Unless you override this default number of bins with a number you specify, R will keep giving this message.

We can change the number of bins to another value using one of these three arguments to `geom_histogram()`:

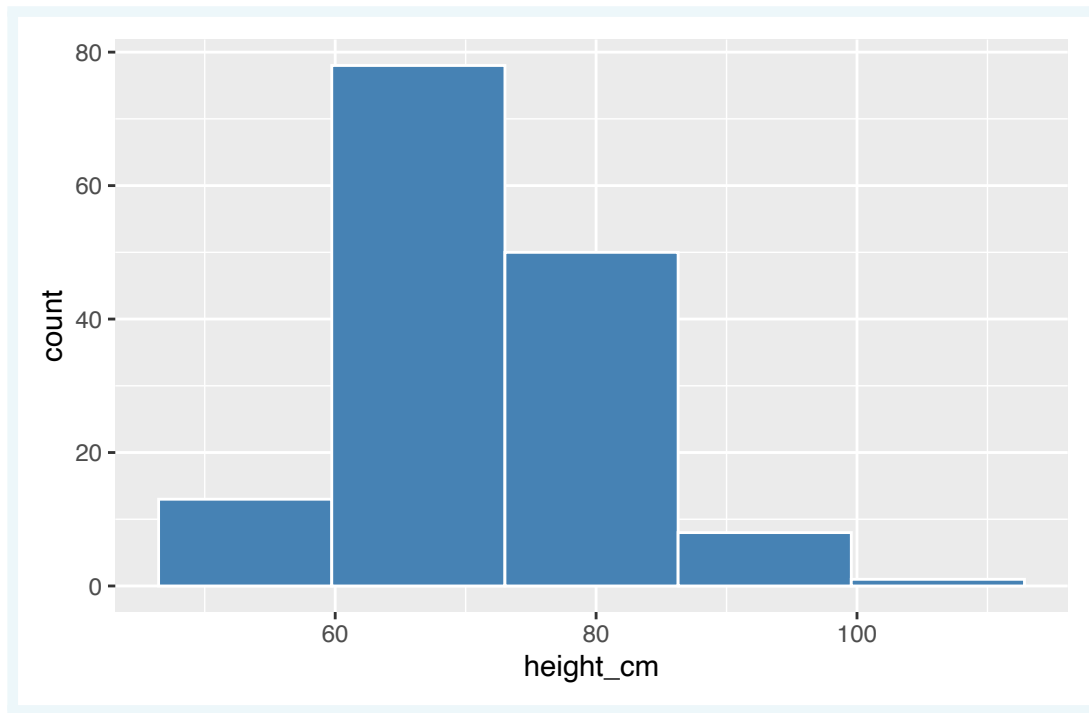
1. Set the number of bins with `bins`
2. Set the width of the bins with `binwidth`
3. Set bin boundaries breaks

21.7.1 Set the number of bins with `bins`

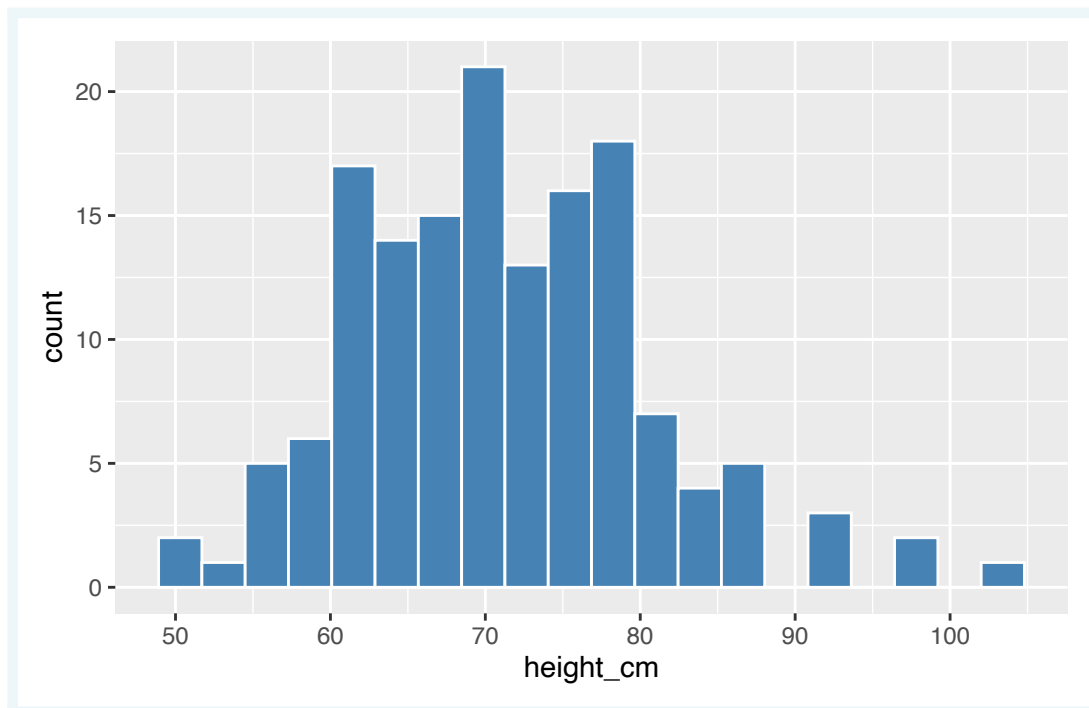
Using the first method, we have the power to specify how many bins we would like to cut the x-axis up in by setting `bins = INTEGER`:

```
## Try different numbers of bins

ggplot(data = malidd ,
       mapping = aes(x = height_cm)) +
  geom_histogram(bins = 5,
                 color = "white",
                 fill = "steelblue")
```

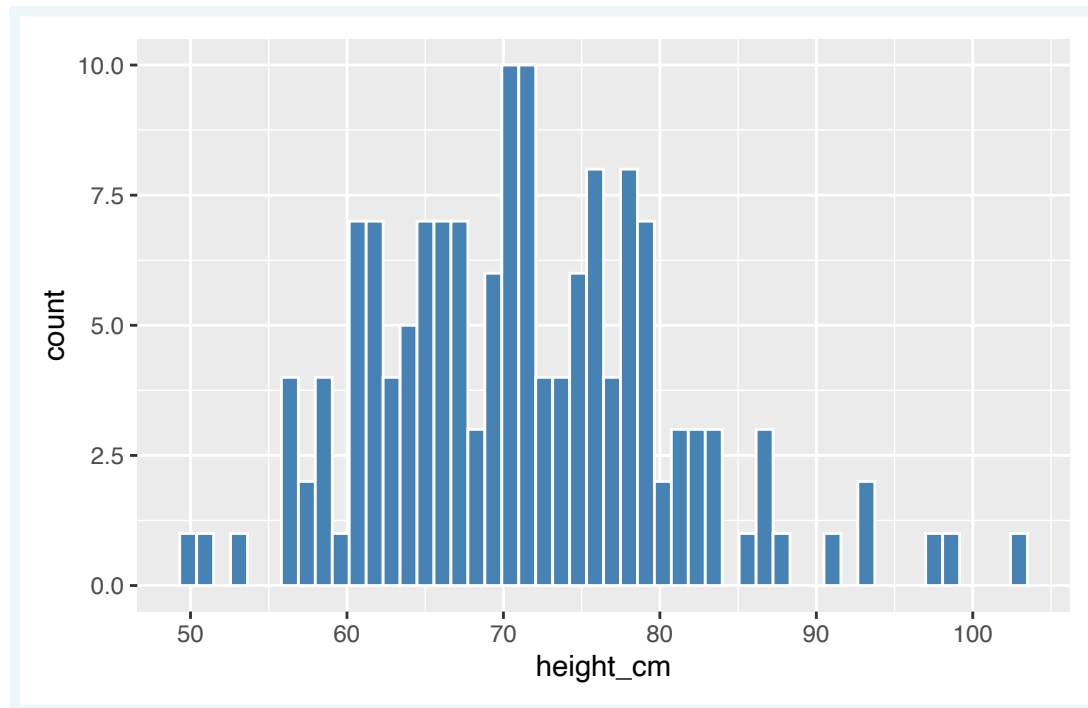


```
ggplot(data = malidd ,  
       mapping = aes(x = height_cm)) +  
  geom_histogram(bins = 20,  
                color = "white",  
                fill = "steelblue")
```



```
ggplot(data = malidd ,  
       mapping = aes(x = height_cm)) +  
  geom_histogram(bins = 50,
```

```
color = "white",
fill = "steelblue")
```



Practice

Make a histogram of frequency of respiration (`freqrespi`), which is measured in breaths per minute. Set the interior color to “`indianred3`”, and border color to “`lightgray`”.

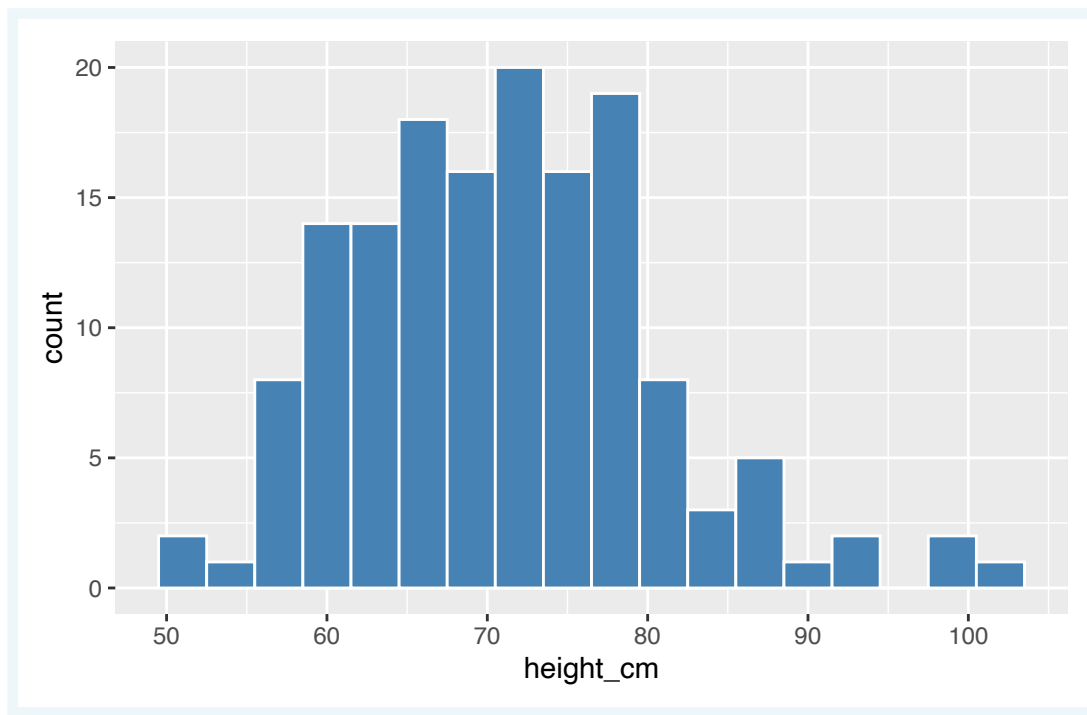
Notice that with the default of 30 bins, there are some intervals for which no bar is plotted (i.e., there were no observations in that range).

Low the number of bins until there are no empty intervals. You should choose the highest value of bins for which there are no empty spaces.

21.7.2 Set the width of bins with `binwidth`

Using the second method, instead of specifying the number of bins, we specify the width of the bins by using the `binwidth` argument in `geom_histogram()`.

```
## Try different bin widths
ggplot(data = malidd,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white",
                fill = "steelblue",
                binwidth = 3)
```

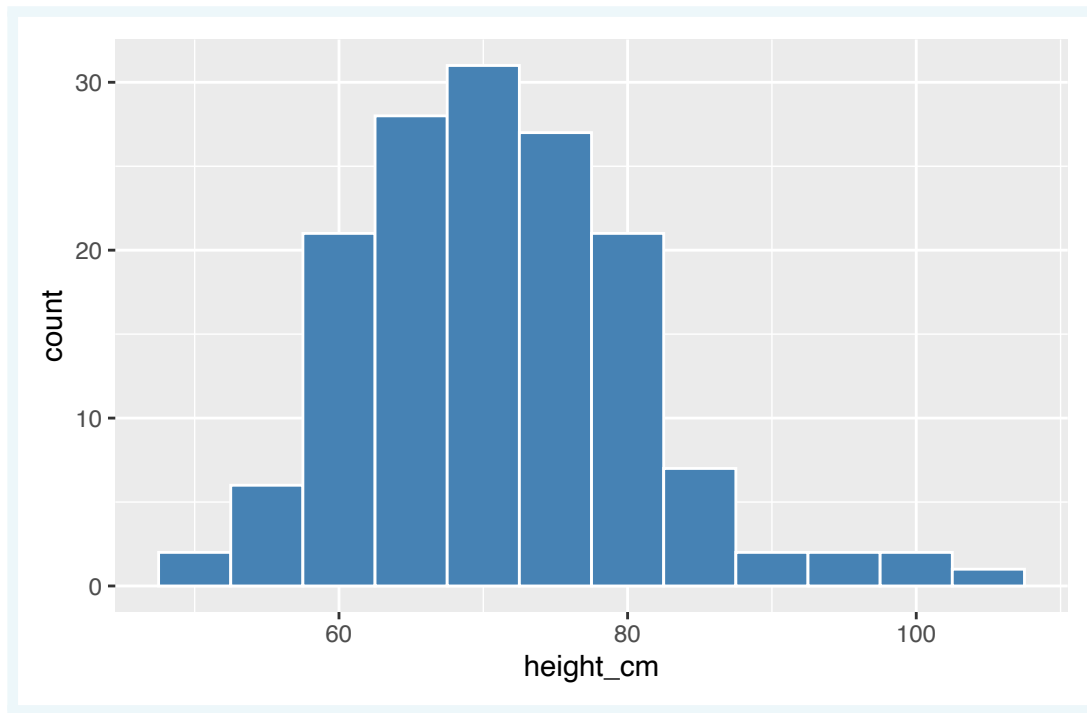


Looking at the range of the variable can help us choose an appropriate bin width.

```
range(malidd$height_cm)
```

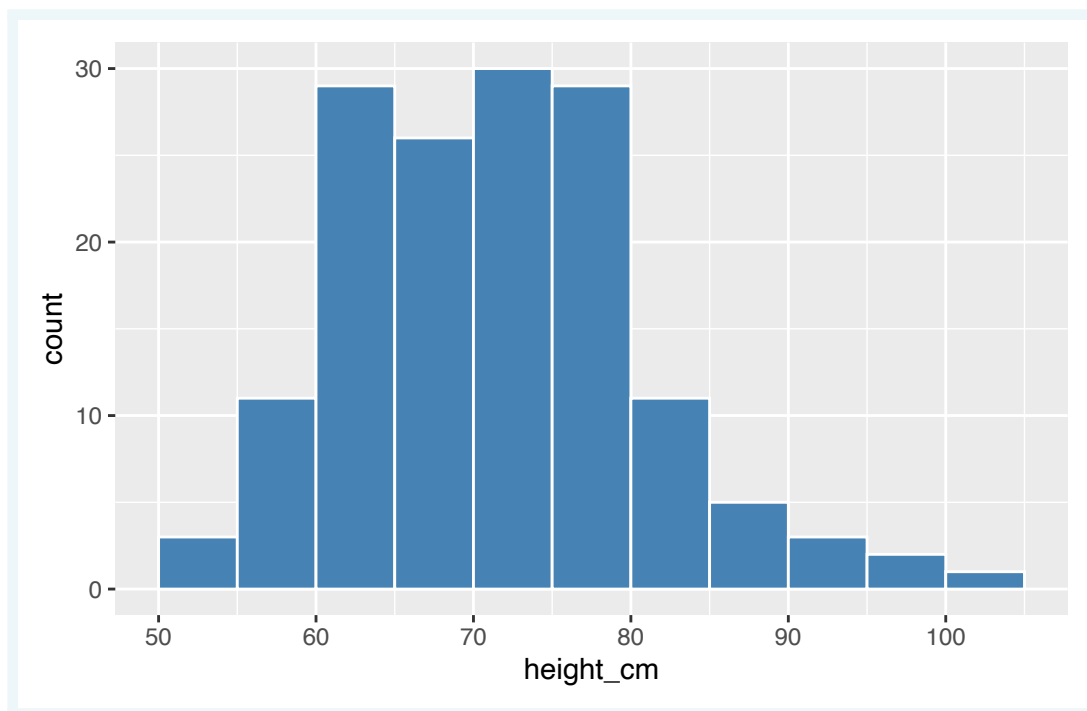
```
[1] 50.3 103.4
```

```
ggplot(data = malidd,  
       mapping = aes(x = height_cm)) +  
  geom_histogram(color = "white",  
                fill = "steelblue",  
                binwidth = 5)
```



We can use the `boundary` argument to align the bins to the x-axis intervals.

```
## Set `boundary` equal to the low end of the variable
ggplot(data = malidd,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white",
                fill = "steelblue",
                binwidth = 5,
                boundary = 50)
```



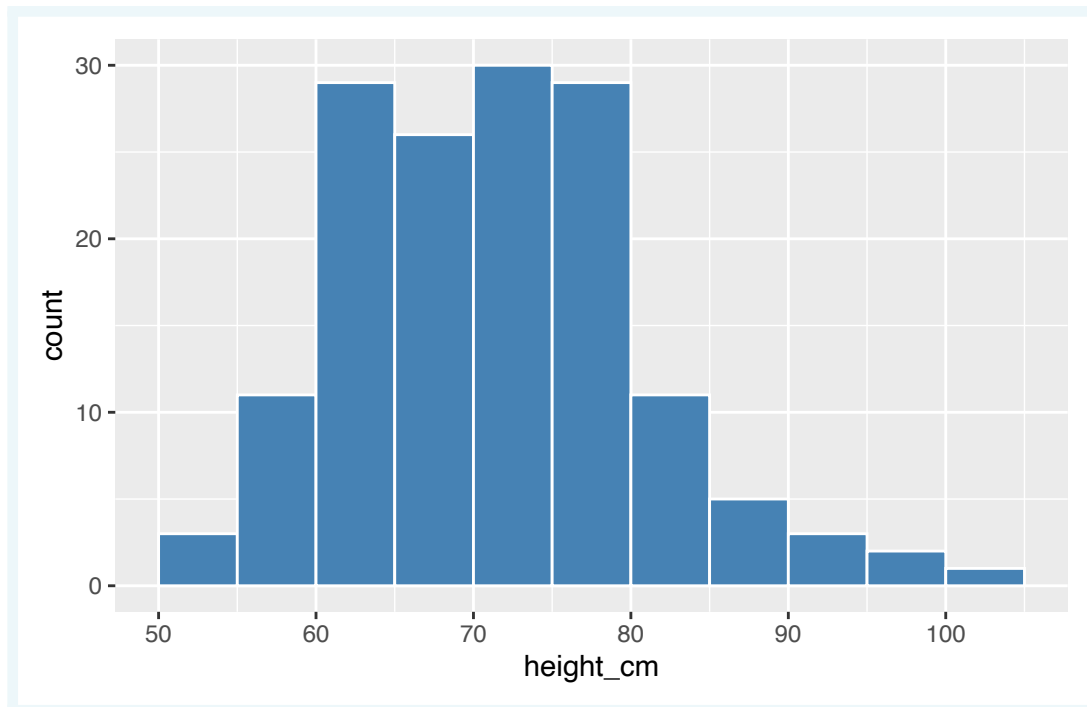
Practice

Create the same `freqrespi` histogram from the last practice question, but this time set the bin width to a value that results in 18 bins. Then align the bars to the x axis breaks by adjusting the bin boundaries.

21.7.3 Modify bin boundaries with breaks

Set breaks equal to a **numeric vector** in `geom_histogram()`:

```
## Supply a vector that covers the range of values in height_cm
ggplot(data = malidd,
       mapping = aes(x = height_cm)) +
  geom_histogram(color = "white",
                fill = "steelblue",
                breaks = seq(50, 105, 5))
```



Practice

Plot the `freqrespi` histogram with bin breaks that range from the lowest value of `freqrespi` to the highest, with intervals of 4.

Next, adjust the x-axis scale breaks by adding a `scale_*()` function. Set the range to 24-60, with an intervals of 8.

21.8 Summary

Histograms, unlike scatterplots and linegraphs, present information on only a single numerical variable. Specifically, they are visualizations of the distribution of the numerical variable in question.

References

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

- Ismay, Chester, and Albert Y. Kim. 2022. *A ModernDive into R and the Tidyverse*. <https://moderndive.com/>.
- Chang, Winston. 2013. *R Graphics Cookbook: Practical Recipes for Visualizing Data*. 1st edition. Beijing Köln: O'Reilly Media.

21.9 Solutions

```
.SOLUTION_q1()
```

```
ggplot(data = malidd,  
       mapping = aes(x = age_months)) +  
  geom_histogram(fill = "seagreen",  
                color = "seagreen",  
                alpha = 0.4)`
```

```
.SOLUTION_q2()
```

```
ggplot(data = .malidd,  
       mapping = aes(x = age_months)) +  
  geom_histogram(fill = "seagreen",  
                color = "seagreen",  
                alpha = 0.4) +  
  labs(x = "Age (months)",  
       y = "Number of children")
```

```
.SOLUTION_q3()
```

```
ggplot(data = malidd,  
       mapping = aes(x = freqrespi)) +  
  geom_histogram(fill = "indianred3",  
                color = "lightgray",  
                bins = 20)
```

```
.SOLUTION_q4()
```

```
ggplot(data = malidd,  
       mapping = aes(x = freqrespi)) +  
  geom_histogram(binwidth = 2,  
                fill = "indianred3",  
                color = "lightgray",  
                boundary = 24)
```

```
.SOLUTION_q5()
```

```
ggplot(data = malidd,  
       mapping = aes(x = freqrespi)) +  
  geom_histogram(fill = "indianred3",  
                color = "lightgray",  
                binwidth = 4) +  
  scale_x_continuous(breaks = seq(24, 60, 8))
```

Chapter 22

Boxplots with {ggplot2}

22.1 Boxplots with {ggplot2}

22.1.1 Learning Objectives

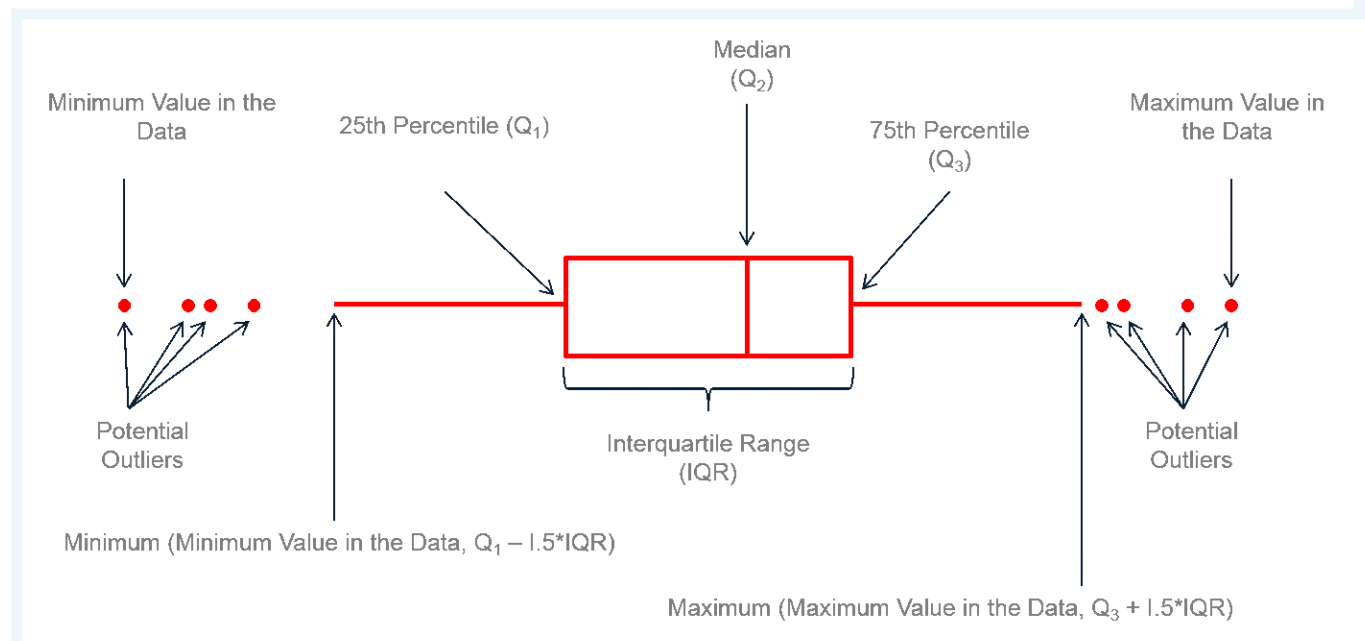
By the end of this lesson, you will be able to:

1. Plot a boxplot to visualize the distribution of continuous data using `geom_boxplot()`.
2. Reorder side-by-side boxplots with the `reorder()` function.
3. Add a layer of data points on a boxplot using `geom_jitter()`.

22.1.2 Introduction

22.1.2.1 Anatomy of a boxplot

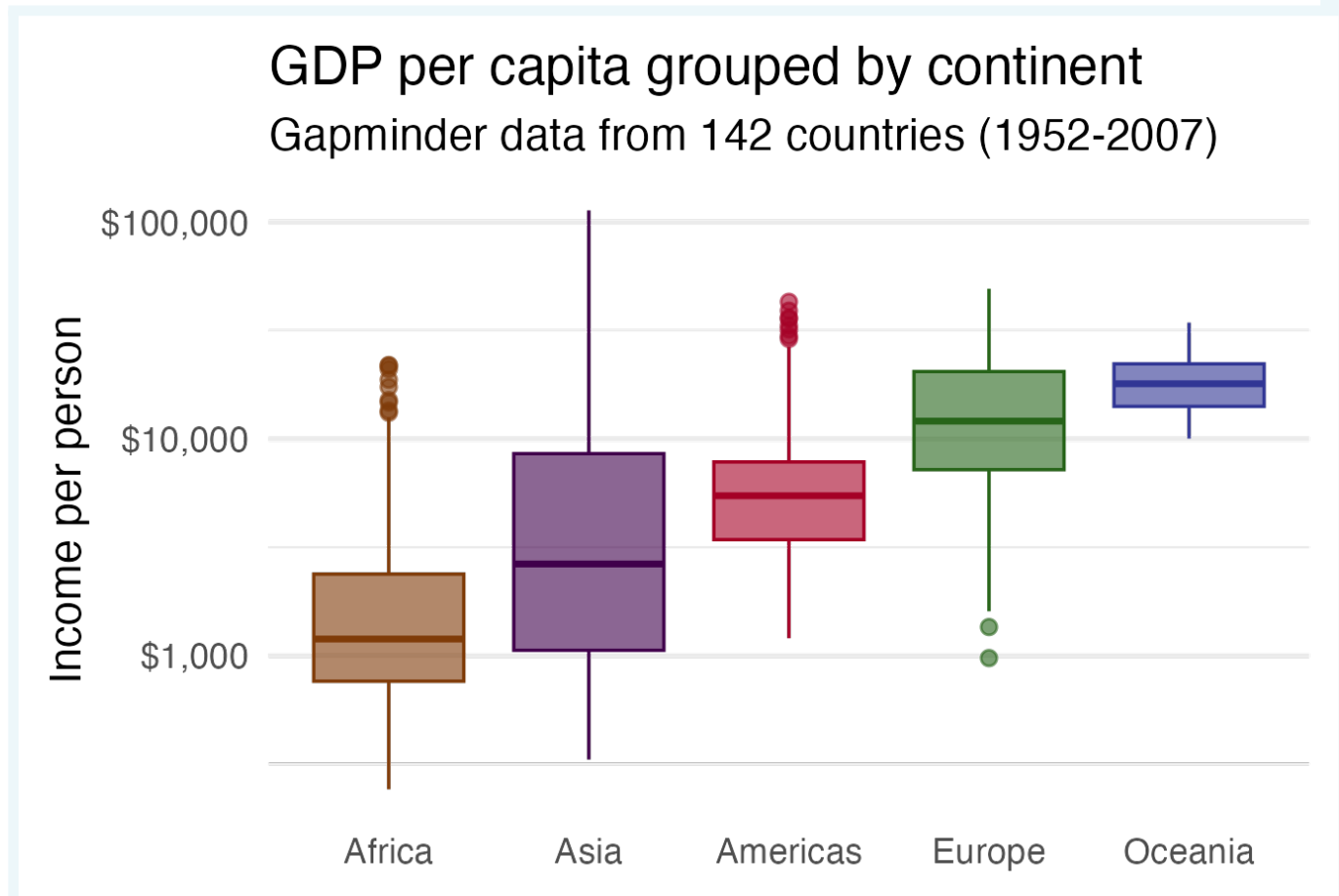
A boxplot allows us to visualize the **distribution** of **numeric** variables.



It consists of two parts:

1. **Box** — Extends from the first to the third quartile (Q1 to Q3) with a line in the middle that represents the *median*. The range of values between Q1 and Q3 is also known as an *Interquartile range (IQR)*.
2. **Whiskers** — Lines extending from both ends of the box indicate variability outside Q1 and Q3. The minimum/maximum whisker values are calculated as $Q1 - 1.5 \times IQR$ to $Q3 + 1.5 \times IQR$. Everything outside is represented as an *outlier* using dots or other markers.

This is *side-by-side boxplot*. It lets us compare the distribution of a numerical variable split by the values of another variable.

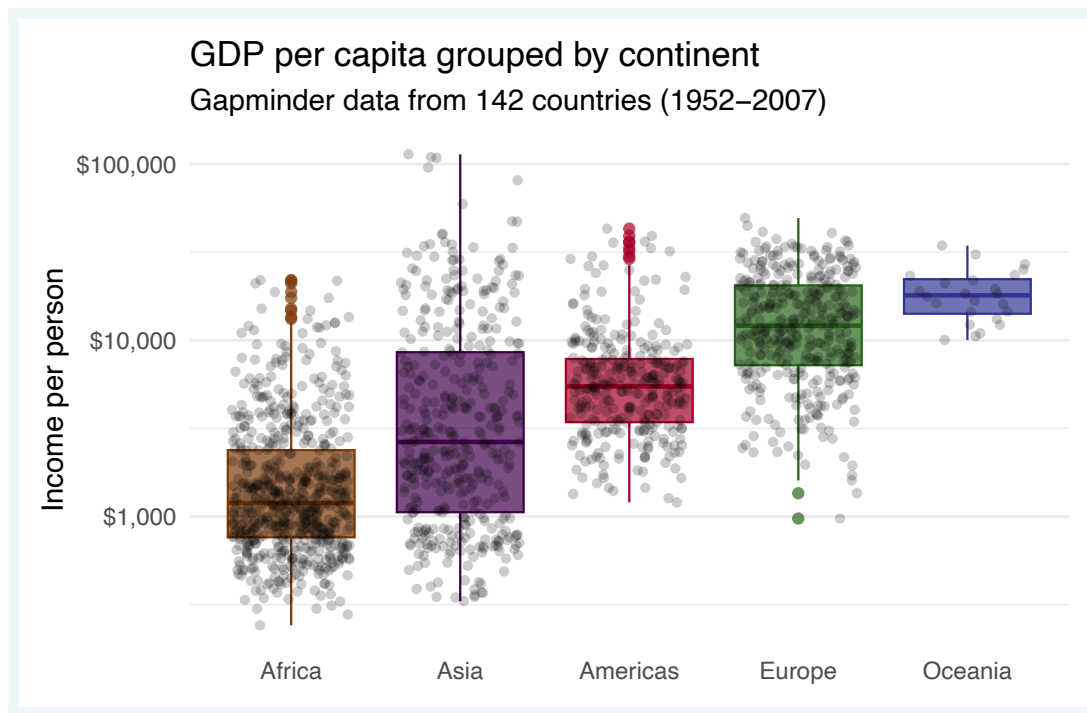


Here we are looking at the variation in GDP per capita – which is a continuous variable – split by different world regions – a categorical variable.

22.1.2.2 Potential pitfalls

Boxplots summarize the data into five numbers, so we might miss important characteristics of the data.

If the amount of data you are working with is not too large, adding individual data points can make the graphic more insightful.



22.1.3 Load packages

```
pacman::p_load(tidyverse,  
               gapminder,  
               here)
```

22.1.4 The gapminder dataset

For this lesson, we will be visualizing global health and economic data from the `gapminder` data frame, which we've encountered in previous lessons.

```
## View first few rows of the data  
head(gapminder)
```

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.801	8425333	779.4453145
Afghanistan	Asia	1957	30.332	9240934	820.8530296
Afghanistan	Asia	1962	31.997	10267083	853.10071
Afghanistan	Asia	1967	34.02	11537966	836.1971382
Afghanistan	Asia	1972	36.088	13079460	739.9811058
Afghanistan	Asia	1977	38.438	14880372	786.11336

i Recap

Gapminder is a country-year dataset with information on 142 countries, divided in to 5 “continents” or world regions.

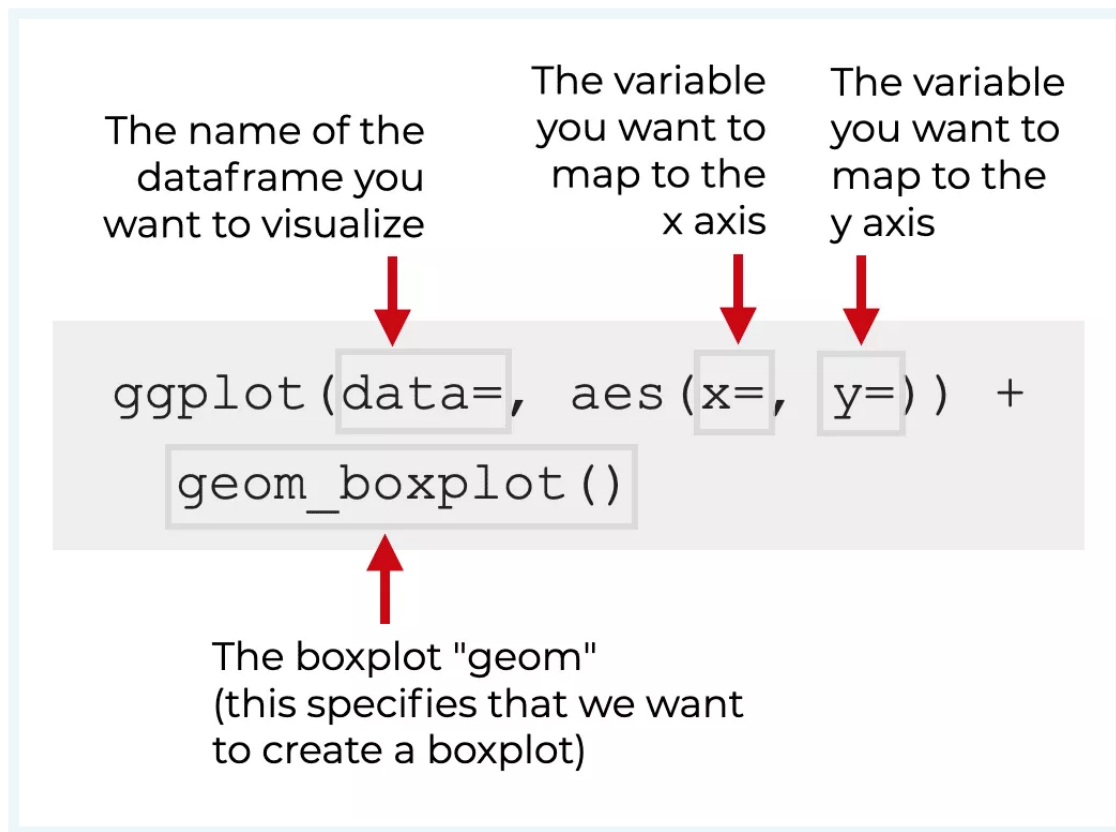
```
## Data summary
summary(gapminder)
```

country	continent	year	lifeExp
Afghanistan: 12	Africa :624	Min. :1952	Min. :23.60
Albania : 12	Americas:300	1st Qu.:1966	1st Qu.:48.20
Algeria : 12	Asia :396	Median :1980	Median :60.71
Angola : 12	Europe :360	Mean :1980	Mean :59.47
Argentina : 12	Oceania : 24	3rd Qu.:1993	3rd Qu.:70.85
Australia : 12		Max. :2007	Max. :82.60
(Other) :1632			
pop	gdpPercap		
Min. :6.001e+04	Min. : 241.2		
1st Qu.:2.794e+06	1st Qu.: 1202.1		
Median :7.024e+06	Median : 3531.8		
Mean :2.960e+07	Mean : 7215.3		
3rd Qu.:1.959e+07	3rd Qu.: 9325.5		
Max. :1.319e+09	Max. :113523.1		

Data are recorded every 5 years from 1952 to 2007 (a total of 12 years).

22.1.5 Basic boxplots with `geom_boxplot()`

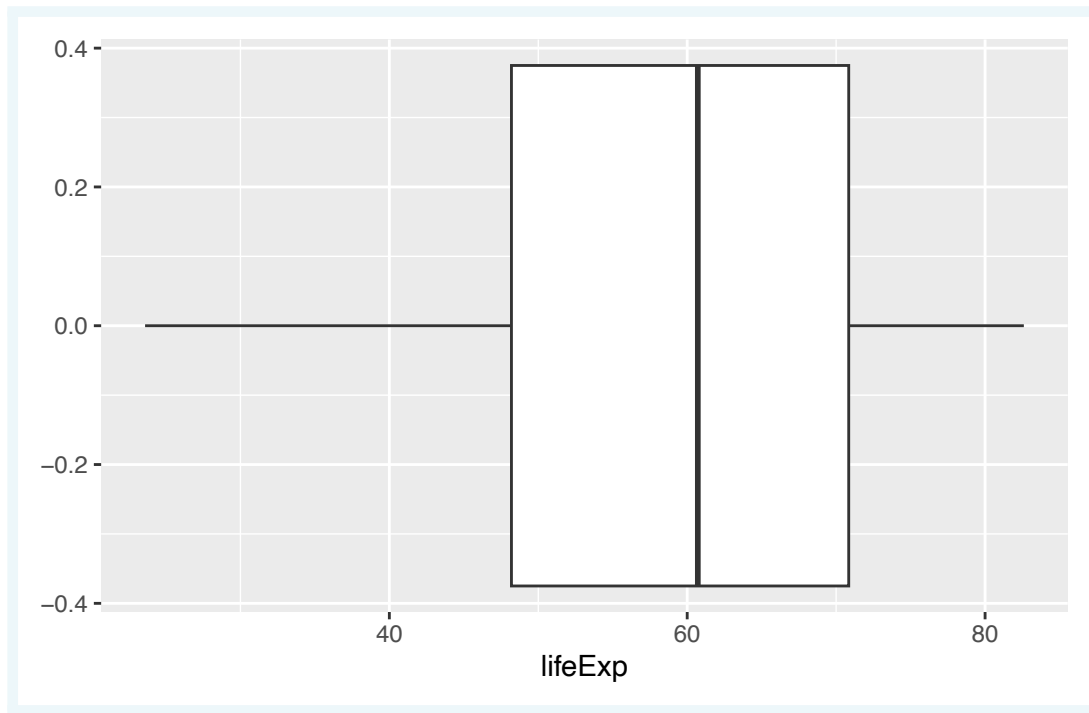
The function for creating boxplots in {ggplot2} is `geom_boxplot()`.



We're going to make a base boxplot and then then add more aesthetics and layers.

Let's start with a simple boxplot by mapping one numeric variable from `gapminder`, life expectancy (`lifeExp`) to the x position.

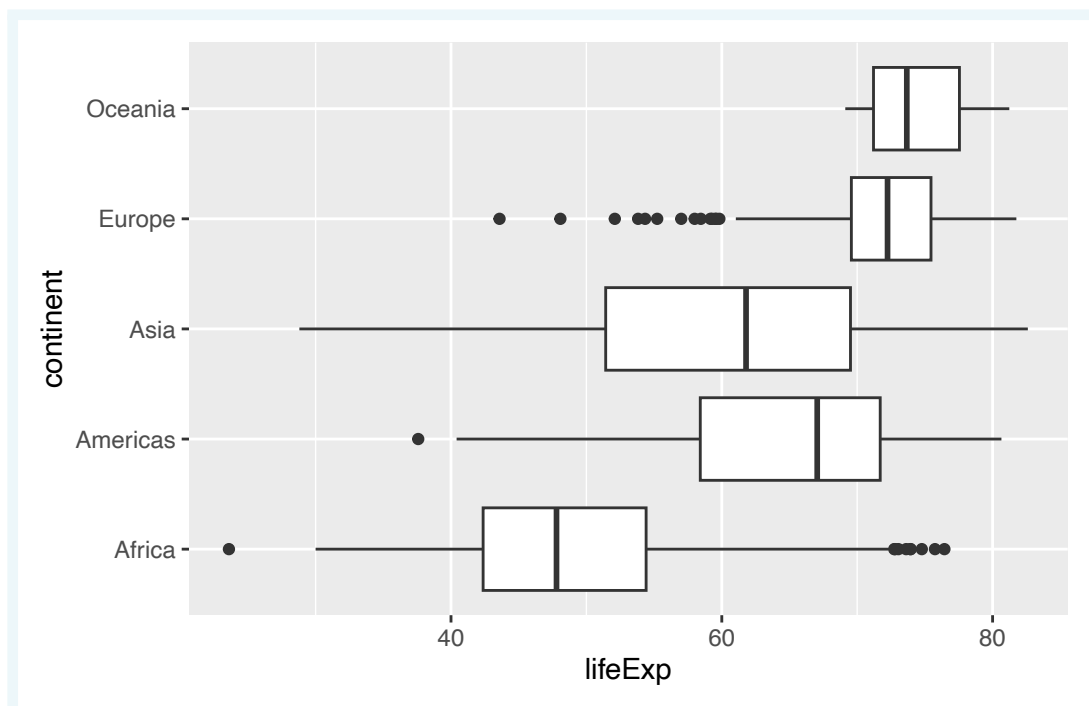
```
## Simple boxplot of lifeExp  
ggplot(data = gapminder,  
       mapping = aes(x = lifeExp)) +  
  geom_boxplot()
```



To create a side-by-side boxplot (which is what we usually want), we need to add a categorical variable to the y position aesthetic.

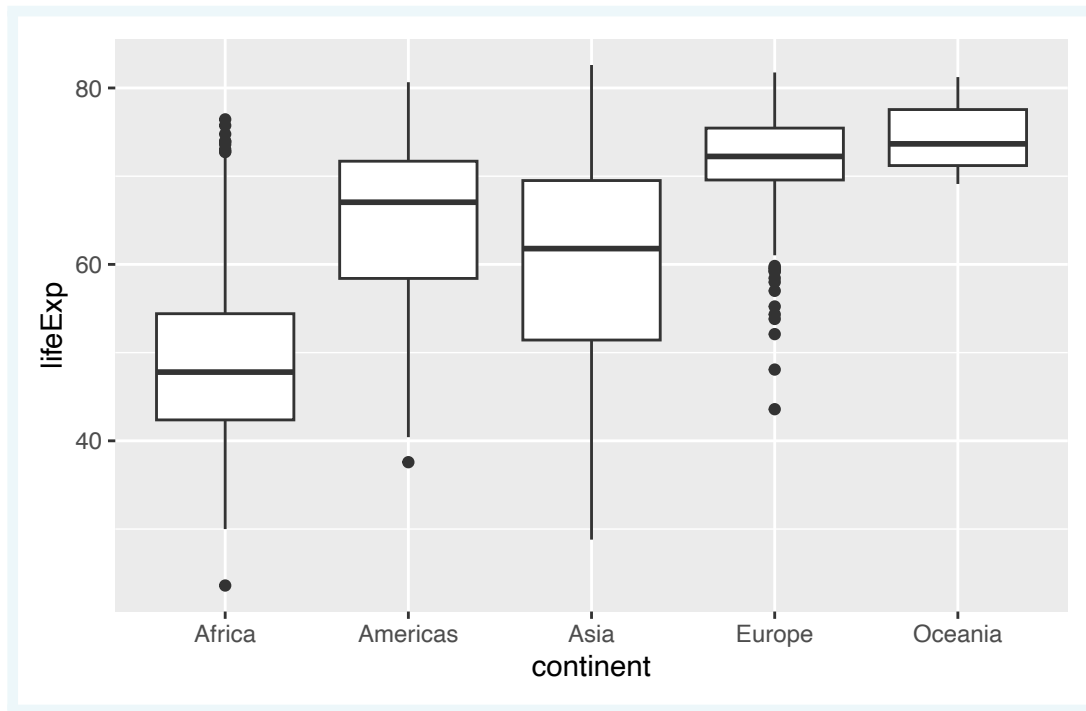
Let's compare life expectancy distributions between continents - i.e., split `lifeExp` by the `continent` variable.

```
## Side-by-side boxplot of lifeExp by continent
ggplot(gapminder,
  aes(x = lifeExp,
    y = continent)) +
  geom_boxplot()
```



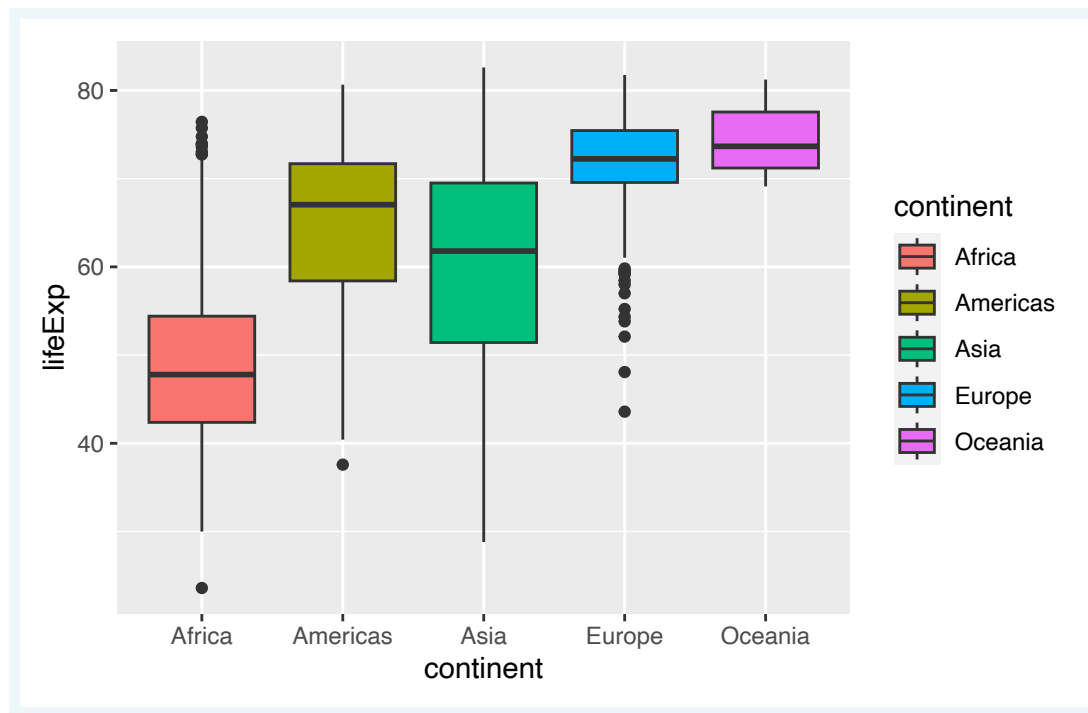
The result is a basic boxplot of `lifeExp` for multiple continents.

```
## Side-by-side boxplot of lifeExp by continent (vertical)
ggplot(data = gapminder,
       mapping = aes(x = continent,
                     y = lifeExp)) +
  geom_boxplot()
```



Let us color in the boxes. We can map the `continent` variable to `fill` so that each box is colored according to which continent it represents.

```
## Fill each continent with a different color
ggplot(gapminder,
       aes(x = continent,
           y = lifeExp,
           fill = continent)) +
  geom_boxplot()
```

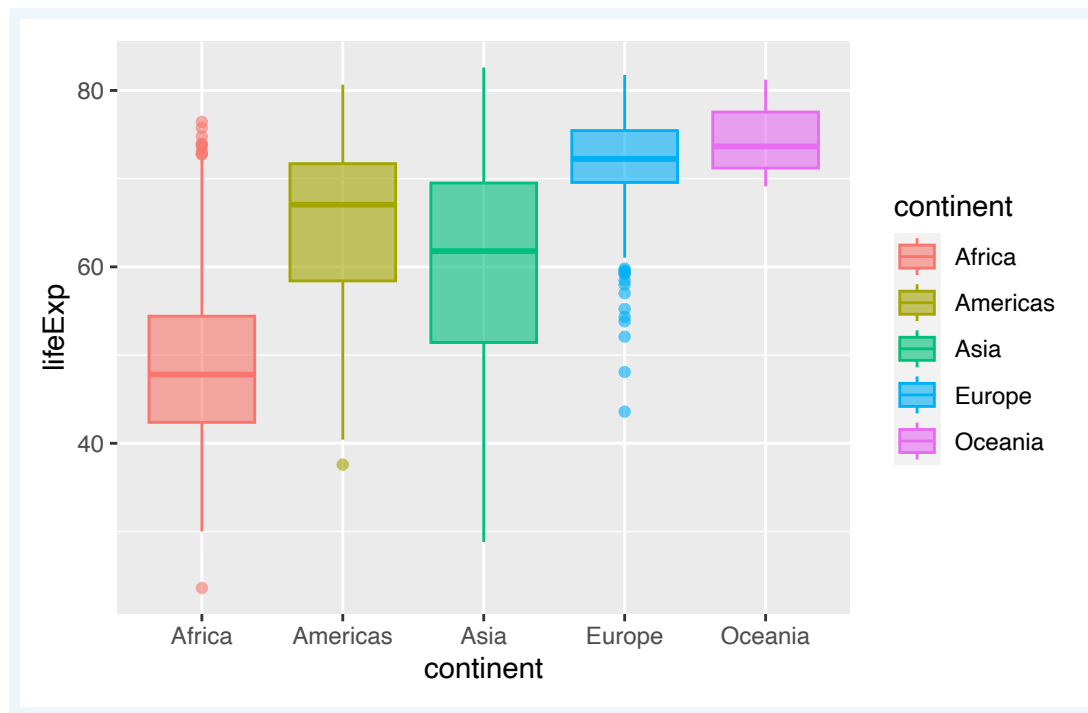


i Reminder

{ggplot2} allows you to color by specifying a variable. We can use `fill` argument inside the `aes()` function to specify which variable is mapped to fill color.

We can also add the `color` and `alpha` aesthetics to change outline color and transparency.

```
## Change outline color and increase transparency
ggplot(gapminder,
  aes(x = continent,
    y = lifeExp,
    fill = continent,
    color = continent)) +
  geom_boxplot(alpha = 0.6)
```



Practice

- Using the `gapminder` data frame create a boxplot comparing the distribution of **GDP per capita** (`gdpPerCap`) across continents. Map the **fill color** of the boxes to continent, and set the **line width** to 1.
- Building on your code from the last question, add a `scale_*()` function that transforms the y-axis to a logarithmic scale.

22.1.6 Reordering boxes with `reorder()`

The values of the `continent` variable are ordered alphabetically by default. If you look at the x-axis, it starts with Africa and goes alphabetically to Oceania.

It might be more useful to order them according to life expectancy, the y-axis variable.

We can change the levels of a factor in R using the `reorder()` function. If we reorder the levels of the `continent` variable, the boxplots will be plotted on the x-axis in that order.

`reorder()` treats its first argument as a categorical variable, and reorders its levels based on the values of a second numeric variable.

To reorder the levels of the `continent` variable based on `lifeExp`, we will use the syntax `reorder(CATEGORIAL_VAR, NUMERIC_VAR)`. Like this: `reorder(continent, lifeExp)`.

Here we will edit the `x` argument and tell `ggplot()` to reorder the variable.

```
ggplot(gapminder,
  aes(x = reorder(continent, lifeExp),
    y = lifeExp,
    fill = continent,
    color = continent)) +
```

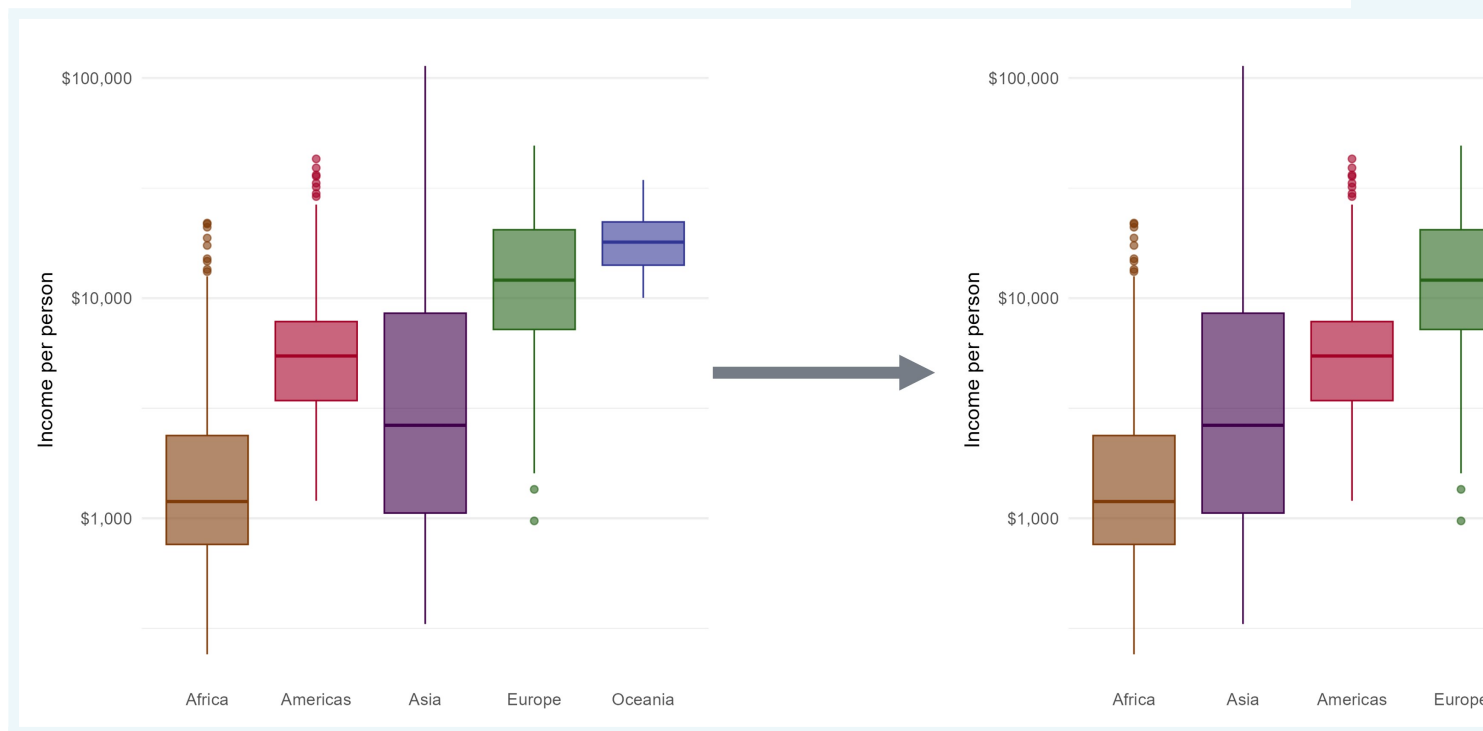
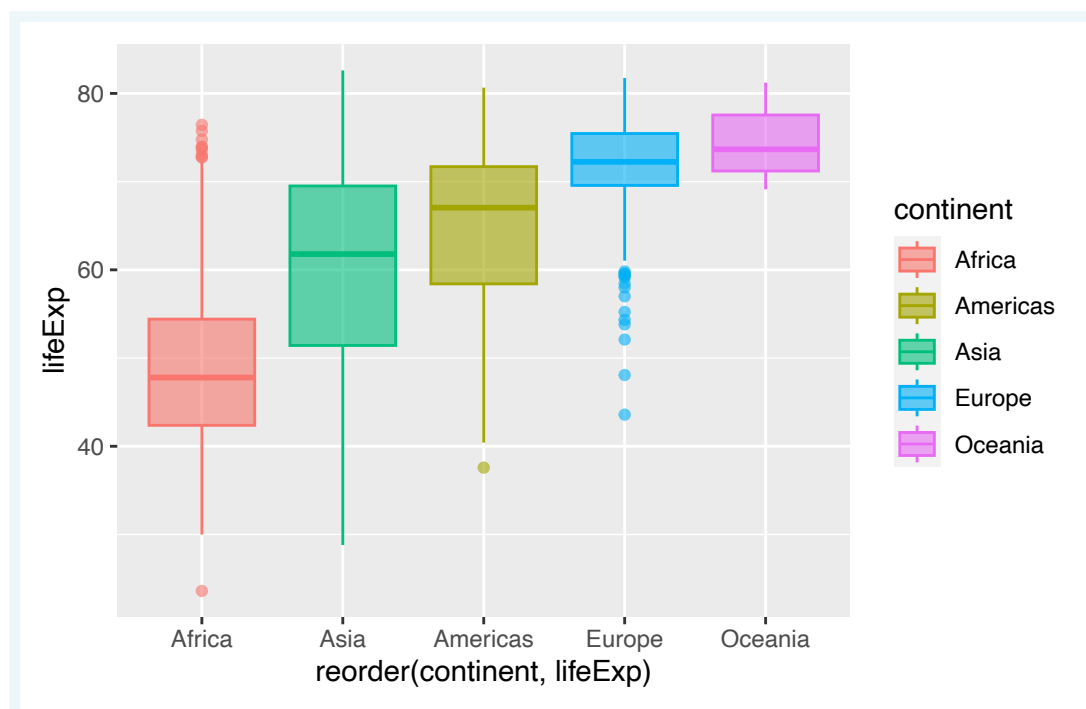


Figure 22.1: Reorder boxplots by life expectancy instead of alphabetical order.

```
geom_boxplot(alpha = 0.6)
```



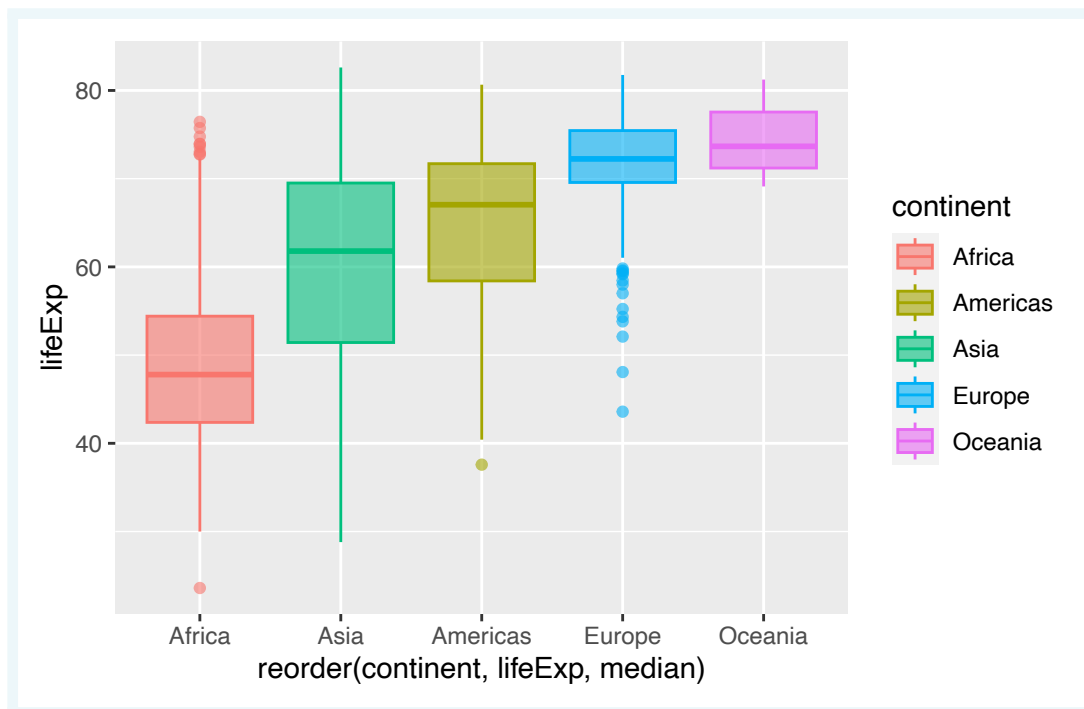
We can clearly see that there are notable differences in median life expectancy between continents. However, there is a lot of overlap between the range of values from each continent. For example, the median life expectancy for the continent of Africa is lower than that of Europe, but several African countries have life expectancy values higher than the majority of European countries.

22.1.6.1 Reordering by function

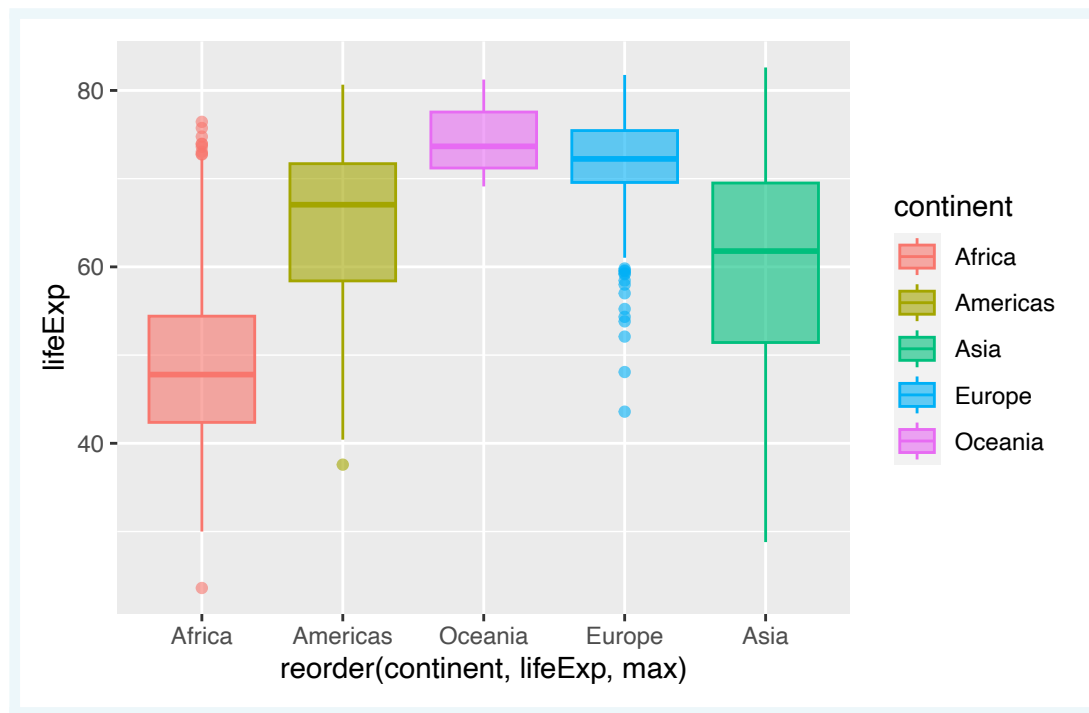
The default method reorders factor based on the **mean** of the numeric variable.

We can add a third argument to choose a different method, like the **median** or **maximum**.

```
## Arrange boxplots by median life expectancy
ggplot(gapminder,
       aes(x = reorder(continent, lifeExp, median),
           y = lifeExp,
           fill = continent,
           color = continent)) +
  geom_boxplot(alpha = 0.6)
```



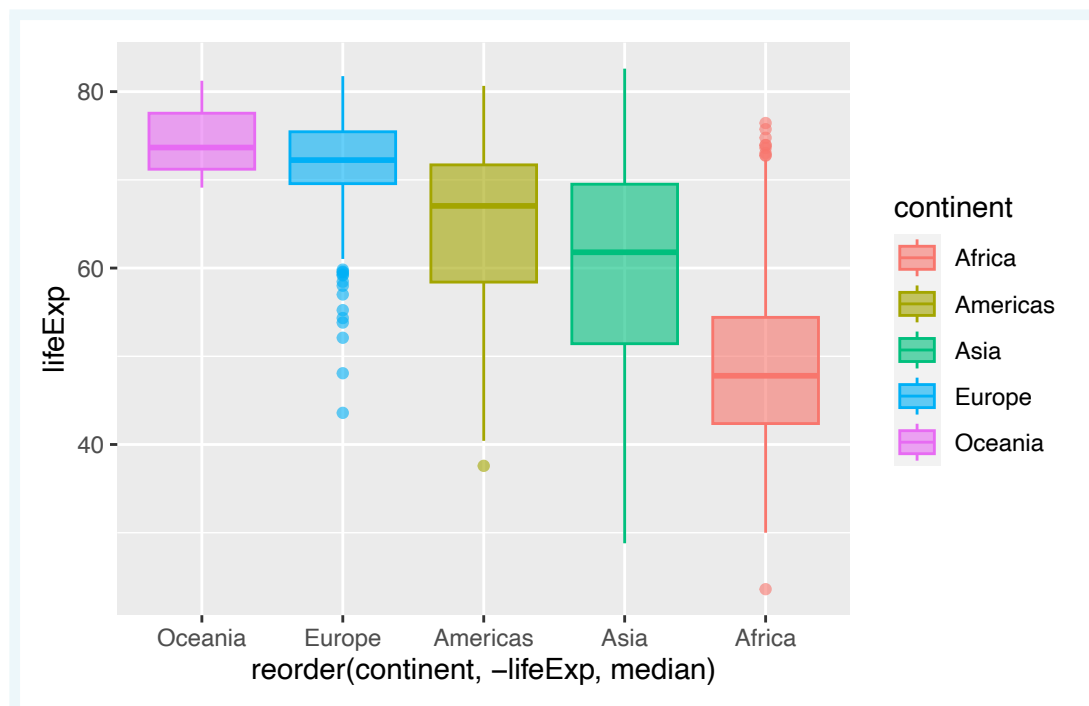
```
## Arrange boxplots by max life expectancy
ggplot(gapminder,
       aes(x = reorder(continent, lifeExp, max),
           y = lifeExp,
           fill = continent,
           color = continent)) +
  geom_boxplot(alpha = 0.6)
```



The boxplots are arranged in **increasing** order.

To sort boxes in boxplot in **descending** order, we add **negation** to lifeExp within the `reorder()` function.

```
## Arrange boxplots by descending median life expectancy
ggplot(gapminder,
  aes(x = reorder(continent, -lifeExp, median),
    y = lifeExp,
    fill = continent,
    color = continent)) +
  geom_boxplot(alpha = 0.6)
```



Practice

Create the boxplot showing the distribution of GDP per capita for each continent, like you did in practice question 2. Retain the fill, line width, and scale from that plot.

Now, **reorder** the boxes by **mean** `gdpPercap`, in **descending** order.

Building on the code from the previous question, add **labels** to your plot.

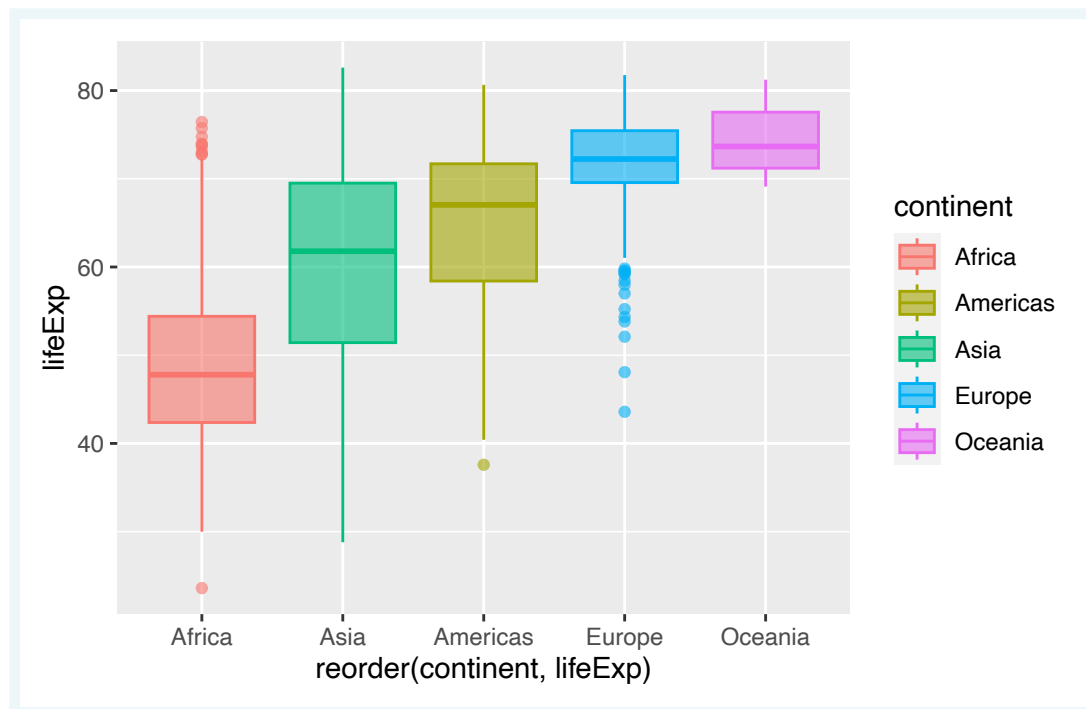
- Set the **main title** to “Variation in GDP per capita across continents (1952-2007)”
- Change the **x-axis title** to “Continent”, and
- Change the **y-axis title** to “Income per person (USD)”.

22.1.7 Adding data points with `geom_jitter()`

Boxplots give us a very high-level summary of the distribution of a numeric variable for several groups. The problem is that summarizing also means losing information.

If we consider our `lifeExp` boxplot, it is easy to conclude that Oceania has a higher value than the others. However, we cannot see the underlying distribution of data points in each group or their number of observations.

```
## Basic lifeExp boxplot from earlier
ggplot(gapminder,
  aes(x = reorder(continent, lifeExp),
    y = lifeExp,
    fill = continent,
    color = continent)) +
  geom_boxplot(alpha = 0.6)
```

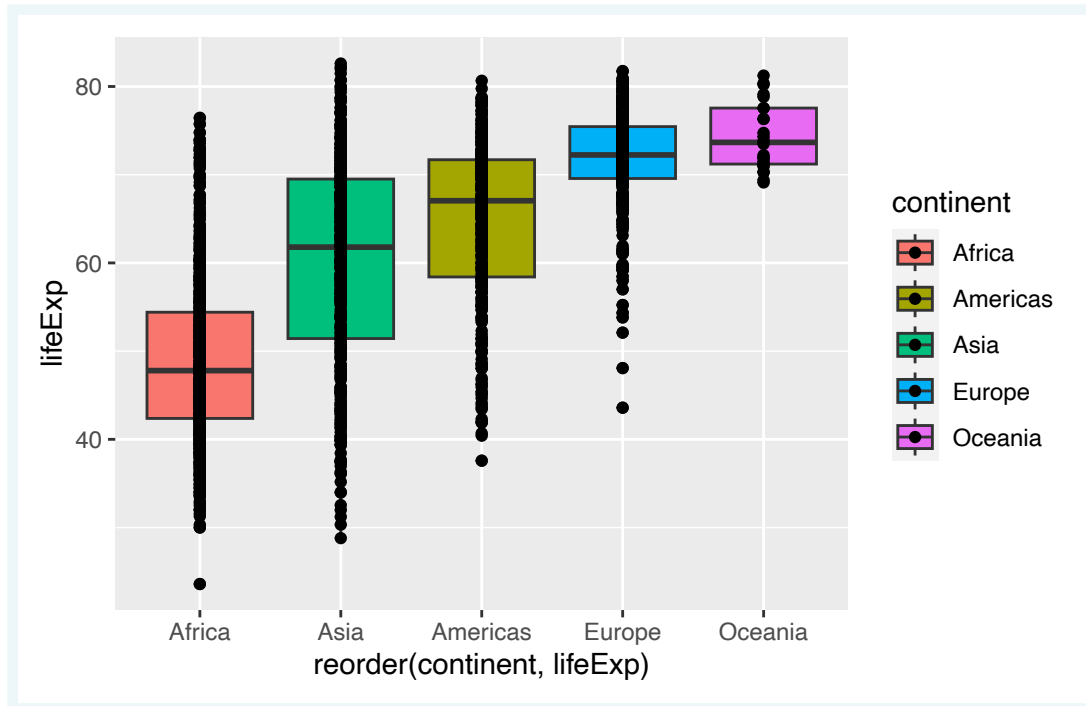


Let's see what happens when the boxplot is improved using additional elements.

One way to display the distribution of individual data points is to plot an additional **layer of points** on top of the boxplot.

We *could* do this by simply adding the `geom_point()` function.

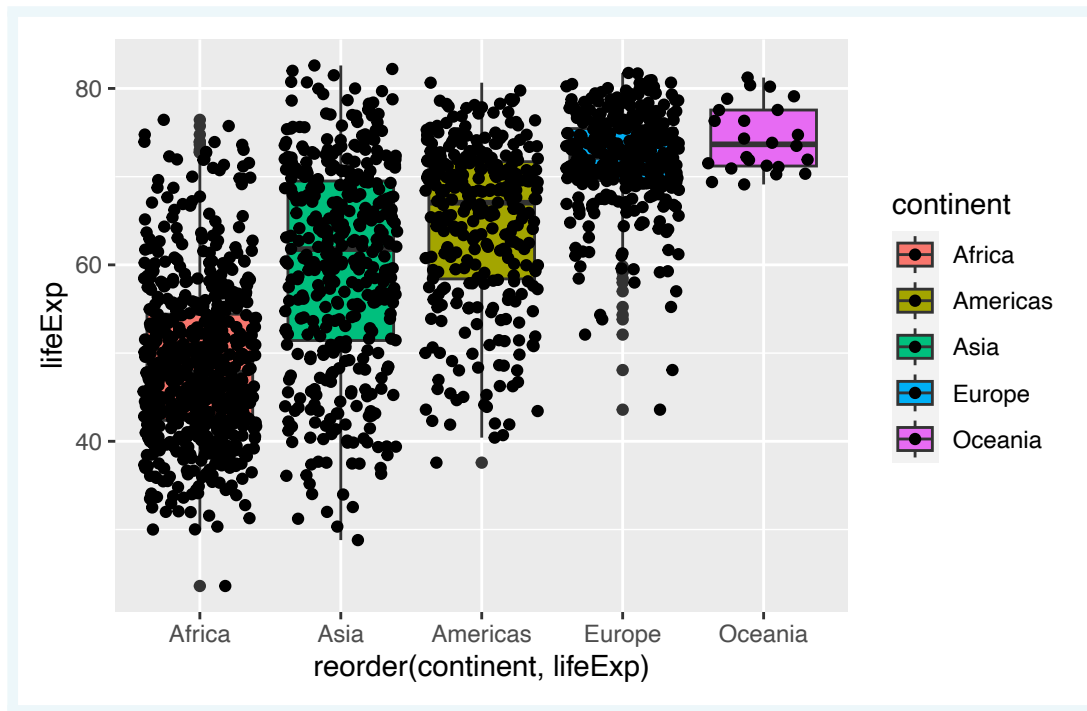
```
ggplot(gapminder,
  aes(x = reorder(continent, lifeExp),
    y = lifeExp,
    fill = continent)) +
  geom_boxplot() +
  geom_point()
```



However, `geom_point()` as has plotted all the data points on a vertical line. That's not very useful since all the points with same life expectancy value directly overlap and are plotted on top of each other.

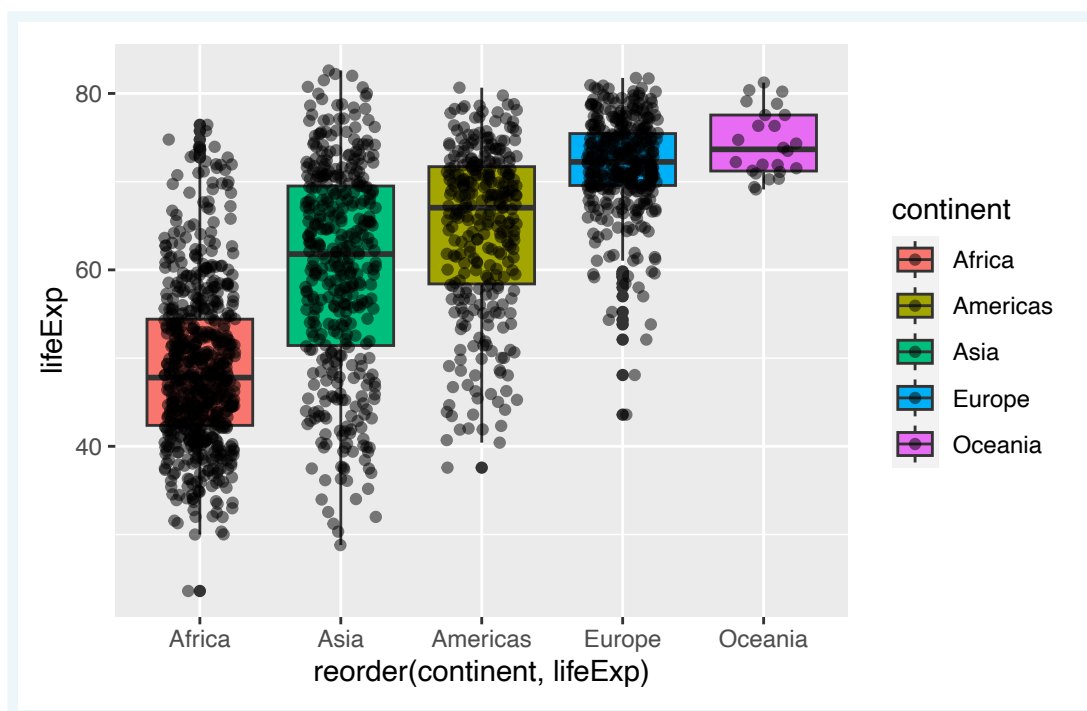
One solution for this is to randomly "jitter" data points horizontally. `ggplot` allows you to do that with the `geom_jitter()` function.

```
ggplot(gapminder,
  aes(x = reorder(continent, lifeExp),
    y = lifeExp,
    fill = continent)) +
  geom_boxplot() +
  geom_jitter()
```



You can also control the amount of jittering with `width` argument and specify opacity of points with `alpha`.

```
ggplot(gapminder,
  aes(x = reorder(continent, lifeExp),
    y = lifeExp,
    fill = continent)) +
  geom_boxplot() +
  geom_jitter(width = 0.25,
    alpha = 0.5)
```



Here some new patterns appear clearly. Oceania has a small sample size compared to the other groups. This is definitely something you want to find out before saying that Oceania has higher life expectancy than the others.

i Recap

Boxplots have the limitation that they summarize the data into five numbers: the 1st quartile, the median (the 2nd quartile), the 3rd quartile, and the upper and lower whiskers. By doing this, we might miss important characteristics of the data. One way to avoid this is by showing the data with points.

💡 Practice

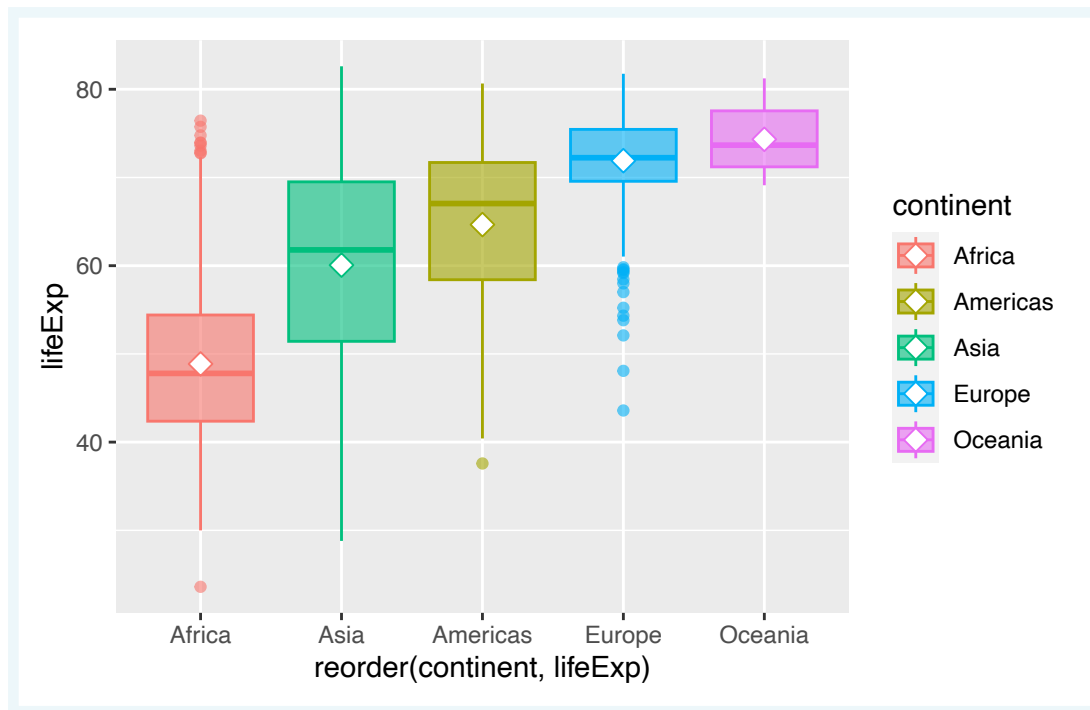
- Create the boxplot showing the distribution of GDP per capita for each continent, like you did in practice question 3. Then add a layer of jittered points.
- Adapt your answer to question 4 to make the points 45% transparent and change the width of the jitter to 0.3mm.

i Challenge

Adding mean markers to a boxplot

You may want to visualize the mean (average) value of the distributions on a boxplot. We can do this by adding a statistics layer using the `stat_summary()` function.

```
## Add a marker to show the mean
ggplot(gapminder,
       aes(x = reorder(continent, lifeExp),
           y = lifeExp,
           fill = continent,
           color = continent)) +
  geom_boxplot(alpha = 0.6) +
  stat_summary(fun = "mean",
              geom = "point",
              size = 3,
              shape = 23,
              fill = "white")
```



22.1.8 Wrap up

Side-by-side boxplots provide us with a way to compare the distribution of a continuous variable across multiple values of another variable. One can see where the median falls across the different groups by comparing the solid lines in the center of the boxes.

To study the spread of a continuous variable within one of the boxes, look at both the length of the box and also how far the whiskers extend from either end of the box. Outliers are even more easily identified when looking at a boxplot than when looking at a histogram as they are marked with distinct points.

22.1.9 Learning Outcomes

1. You can plot a boxplot to visualize the distribution of continuous data using `geom_boxplot()`.
2. You can reorder side-by-side boxplots with the `reorder()` function.
3. You can add a layer of individual data points on a boxplot using `geom_jitter()`.

References

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

- Ismay, Chester, and Albert Y. Kim. 2022. *A ModernDive into R and the Tidyverse*. <https://moderndive.com/>.

22.2 Solutions

```
.SOLUTION_q1()
```

```
ggplot(data = gapminder,  
  mapping = aes(x = continent, y = gdpPercap, fill = continent)) +  
  geom_boxplot(linewidth = 1)
```

```
.SOLUTION_q2()
```

```
ggplot(data = gapminder,  
  mapping = aes(x = continent, y = gdpPercap, fill = continent)) +  
  geom_boxplot(linewidth = 1) +  
  scale_y_log10()
```

```
.SOLUTION_q3()
```

```
ggplot(data = gapminder,  
  mapping = aes(  
    x = reorder(continent, -gdpPercap),  
    y = gdpPercap,  
    fill = continent)) +  
  geom_boxplot(linewidth = 1) +  
  scale_y_log10()
```

```
.SOLUTION_q4()
```

```
ggplot(data = gapminder,  
  mapping = aes(  
    x = reorder(continent, -gdpPercap),  
    y = gdpPercap,  
    fill = continent)) +  
  geom_boxplot(linewidth = 1) +  
  scale_y_log10() +  
  labs(title = "Variation in GDP per capita across continents (1952-2007)",  
    x = "Continent",  
    y = "Income per person (USD)")
```

```
.SOLUTION_q5()
```

```
ggplot(data = gapminder,  
  mapping = aes(  
    x = reorder(continent, -gdpPercap),  
    y = gdpPercap,
```

```
fill = continent)) +  
geom_boxplot(linewidth = 1) +  
scale_y_log10() +  
labs(title = "Variation in GDP per capita across continents (1952-2007)",  
      x = "Continent",  
      y = "Income per person (USD)") +  
geom_jitter()
```

```
.SOLUTION_q6()
```

```
ggplot(data = gapminder,  
       mapping = aes(  
         x = reorder(continent, -gdpPercap),  
         y = gdpPercap,  
         fill = continent)) +  
geom_boxplot(linewidth = 1) +  
scale_y_log10() +  
labs(title = "Variation in GDP per capita across continents (1952-2007)",  
      x = "Continent",  
      y = "Income per person (USD)") +  
geom_jitter(width = 0.3, alpha = 0.55)
```