



Intro to data analysis with R

Unlocking the power of R for public health data science

Lesson notes | Setting up R and RStudio

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Working locally vs. on the cloud
RStudio on the cloud
Set up on Windows
Download and install R
Download, install & run RStudio
Set up on macOS
Download and install R
Download, install & run RStudio
Wrap up

Learning objective

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

Introduction

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

Working locally vs. on the cloud

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

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

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

WATCH OUT



WATCH OUT

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

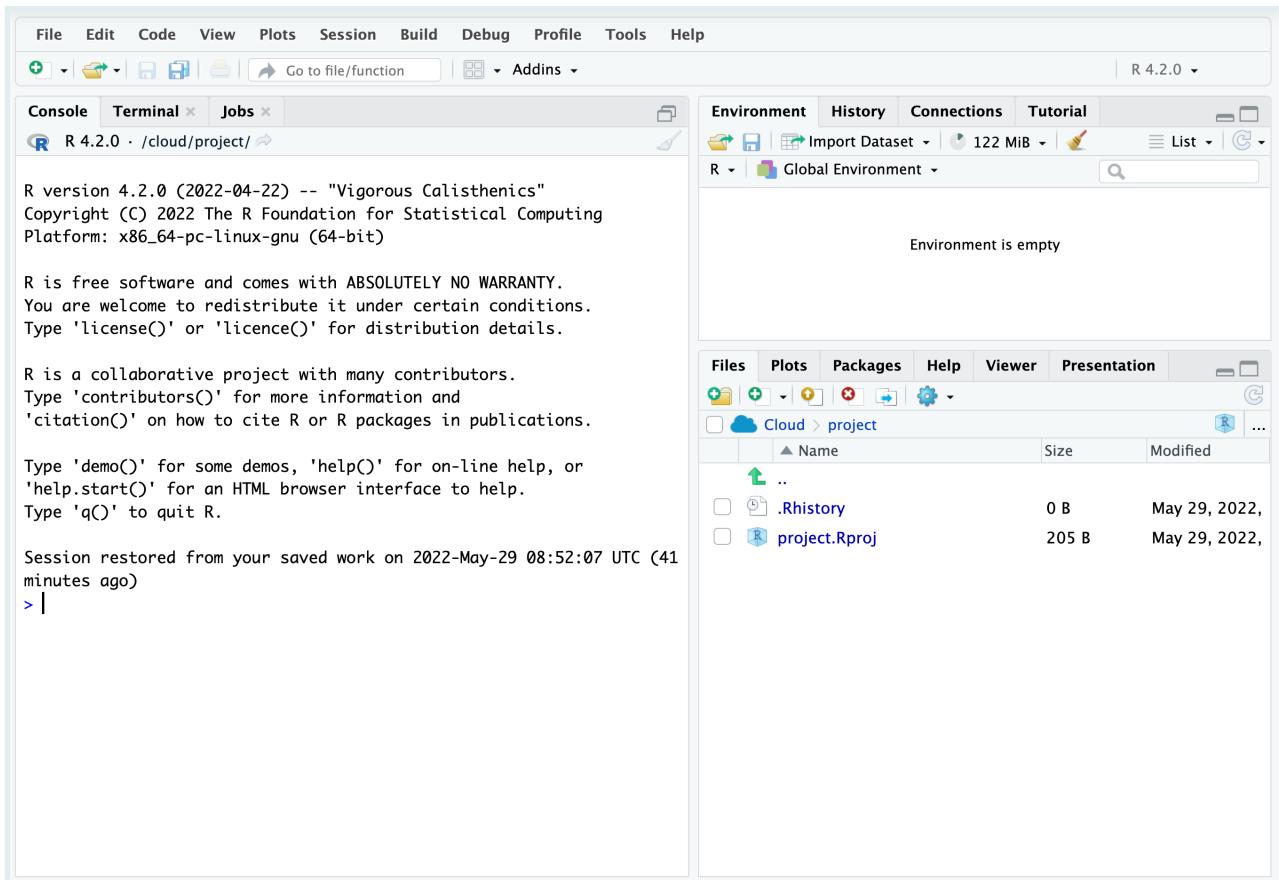
RStudio on the cloud

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

1. Go to the website rstudio.cloud and follow the instructions to sign up for a free account. (We recommend signing up with Google if you have a Google account, so you don't need to remember any new passwords).
2. Once you're done, click on the "New Project" icon at the top right, and select "New RStudio Project".

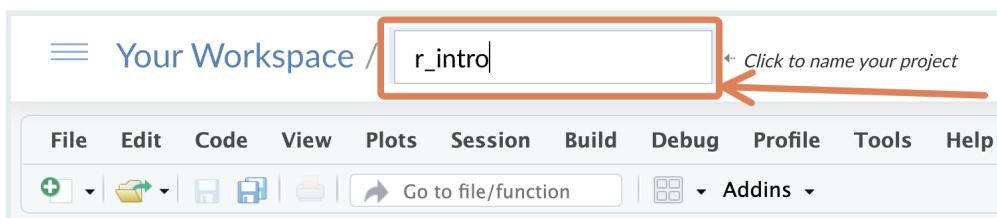
The screenshot shows the RStudio Cloud interface. On the left, there's a sidebar with 'Spaces' (Your Workspace selected) and 'Learn' (Guide). The main area is titled 'Your Projects' with '(0)' next to it. There are buttons for 'Your Projects' (highlighted in blue) and 'Trash'. At the top right, there's a user profile for 'Jane Doe'. Below the profile, there's a 'New Project' dropdown menu with three options: 'New RStudio Project' (selected and highlighted with a red box), 'New Jupyter Project', and 'New Project from Git Repository'. An orange arrow points from the text 'Once you're done, click on the "New Project" icon at the top right, and select "New RStudio Project"' to the 'New Project' button in the screenshot.

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' tab selected. The R startup message is displayed, followed by standard R documentation and a session restore message. A red arrow points from the text 'Write code here' to the command line area where the expression '`> 2 + 2`' is typed.

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

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

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

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

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

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

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

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

Set up on Windows

Download and install R

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

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

The Comprehensive R Archive Network

Download and Install R

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

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

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

2. Choose the “base” sub-directory.

R for Windows

Subdirectories:

[base](#)

[contrib](#)

[old contrib](#)

Binaries for base distribution. This is what you want to [install R for the first time](#).

Binaries of contributed CRAN packages (for R >= 3.4.x).

Binaries of contributed CRAN packages for outdated versions of R (for R < 3.4.x).

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

R-4.2.0 for Windows

[Download R-4.2.0 for Windows \(79 megabytes, 64 bit\)](#)

[README on the Windows binary distribution](#)

[New features in this version](#)

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

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

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

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

Download, install & run RStudio

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

2. Download RStudio Desktop. Recommended for your system:

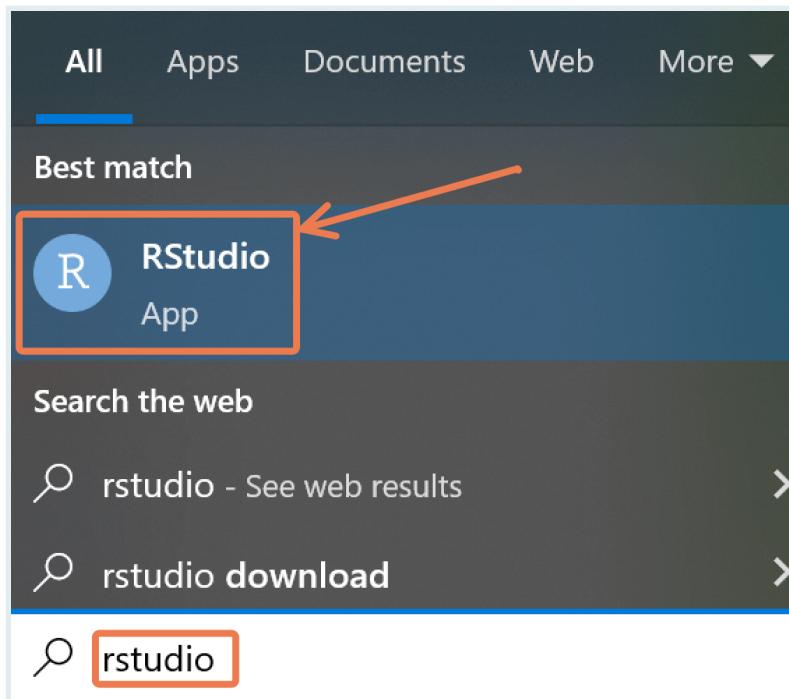


DOWNLOAD RSTUDIO FOR WINDOWS

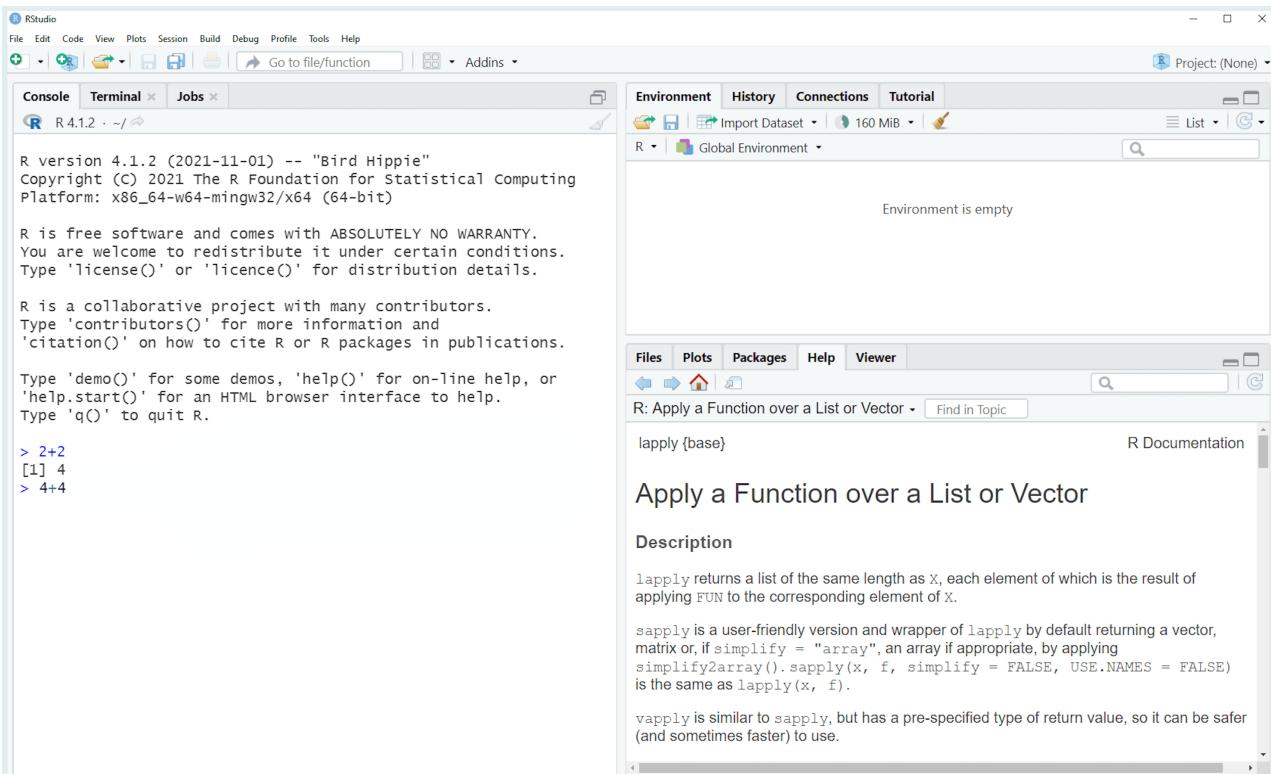
2022.02.0+443 | 176.76MB

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

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

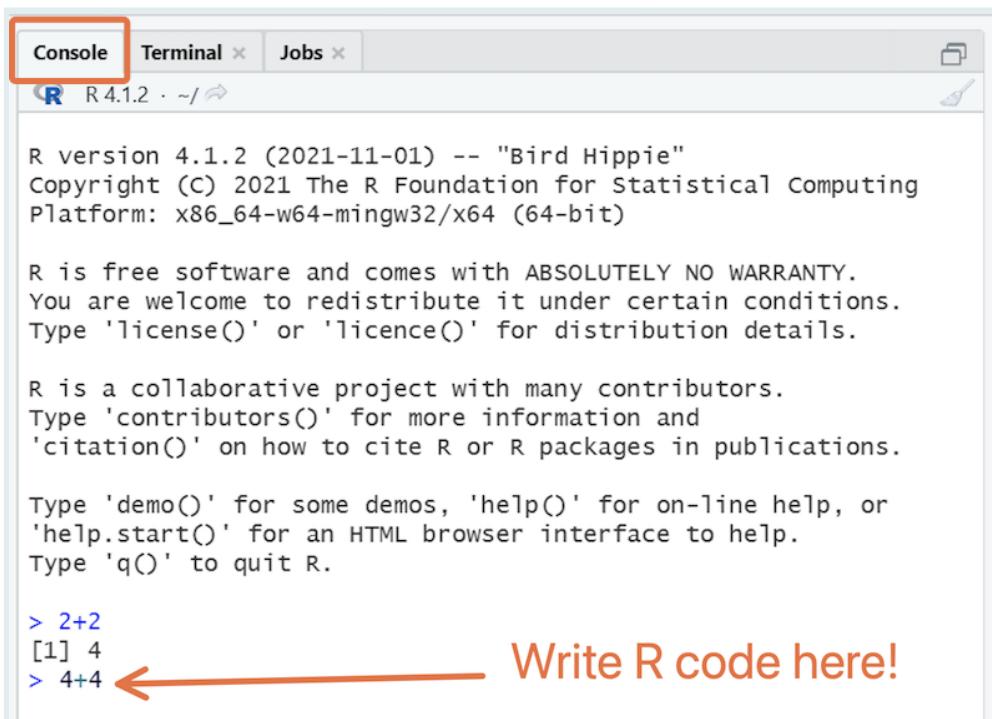


You should see a window like this:



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.

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

Set up on macOS

Download and install R

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

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

The Comprehensive R Archive Network

Download and Install R

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

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

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

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

[R-4.2.0.pkg](#) (notarized and signed)
SHA1-hash: 2a90fb8629e44f729d89da9bac9b71564587d7
(ca. 90MB) for Intel Macs

Latest version for Intel Macs

Latest release:

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 `tcltk` R package or build package documentation from sources.

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

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

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

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

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 `tcltk`) requires [XQuartz](#) (version 2.8.1 or later). Always re-install XQuartz when upgrading your macOS to a new major version.

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

[NEWS](#) (for Mac GUI)

News features and changes in the R.app Mac GUI

[Mac-GUI-1.78.tar.gz](#)

SHA1-hash: 23b5c41b7eb6771640fd504a75e5782792dddb2bc

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

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

Binaries for legacy OS X systems:

[R-3.6.3.mn.pkg](#) (signed)
SHA1-hash: c462c9b19b45d778f05b8d9aa25a9123b3557c4
(ca. 77MB)

For older macs

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 `tcltk` R package or build package documentation from sources.

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

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

Download, install & run RStudio

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

2. Download RStudio Desktop. Recommended for your system:



[DOWNLOAD RSTUDIO FOR MAC](#)

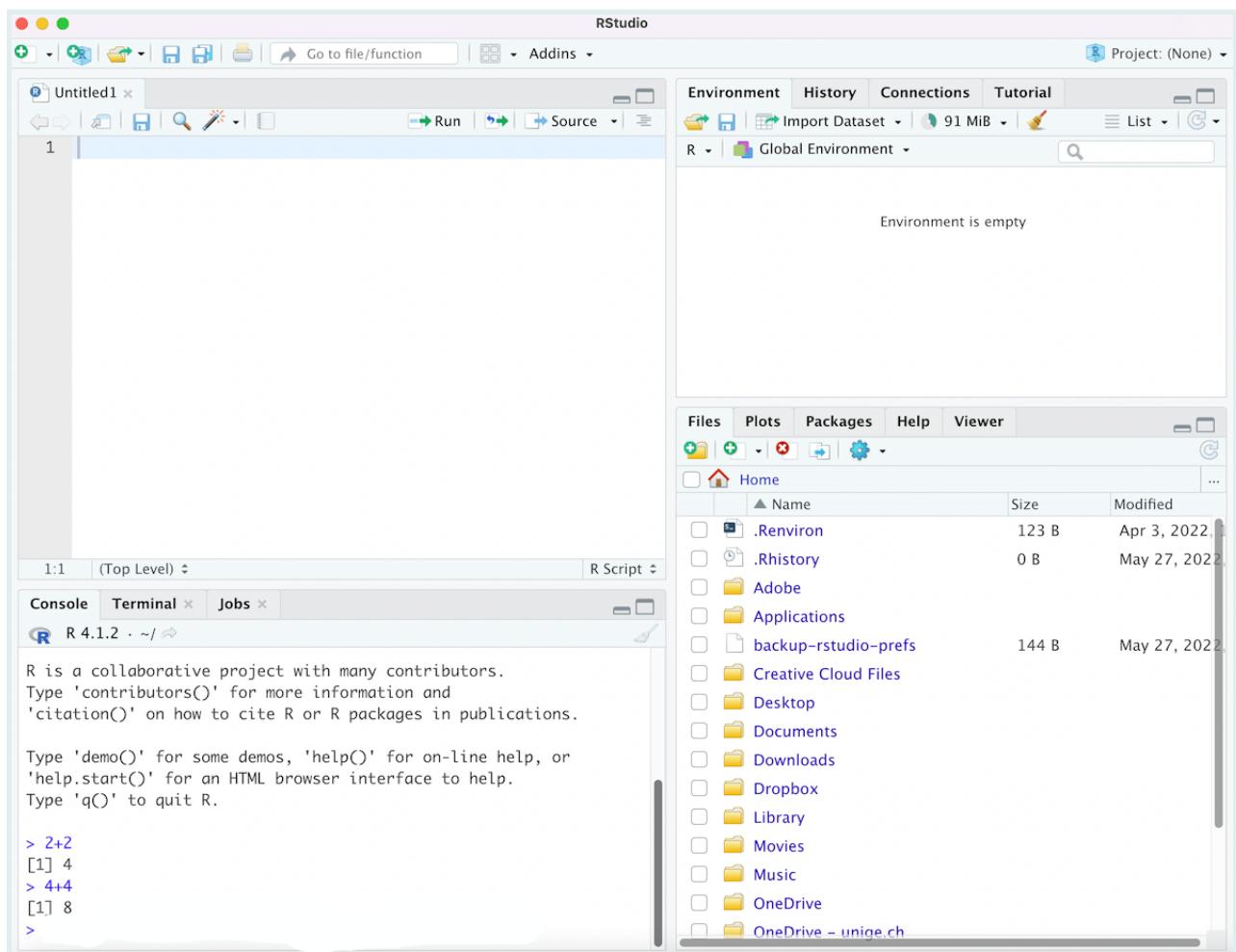
2022.02.0+443 | 217.18MB

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:

The screenshot shows the RStudio interface with the 'Console' tab selected. The R version is 4.1.2. The console output shows the following interaction:

```
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.  
  
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.  
  
> 2+2  
[1] 4  
> 4+4  
[1] 8
```

An orange arrow points from the text "Write code here" to the first input line in the console.

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

Wrap up

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

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



LAMECK AGASA

Statistician/Data Scientist



MICHAL SHRESTHA

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



ELTON MUKONDA

Data analyst, the GRAPH Network

A data enthusiast with a passion for population health research



OLIVIA KEISER

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

References

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

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

This work is licensed under the Creative Commons Attribution Share Alike license.



Lesson notes | Using RStudio

Created by the GRAPH Courses team

January 2023

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

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

Learning objectives
Introduction
The RStudio panes
Source/Editor
Console
Environment
History
Files
Plots
Packages
Viewer
Help
RStudio options
Command palette
Wrapping up
Further resources
References

Learning objectives

1. You can identify and use the following tabs in RStudio: Source, Console, Environment, History, Files, Plots, Packages, Help and Viewer.
2. You can modify RStudio's interface options to suit your needs.

Introduction

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

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

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

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

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

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

The RStudio panes

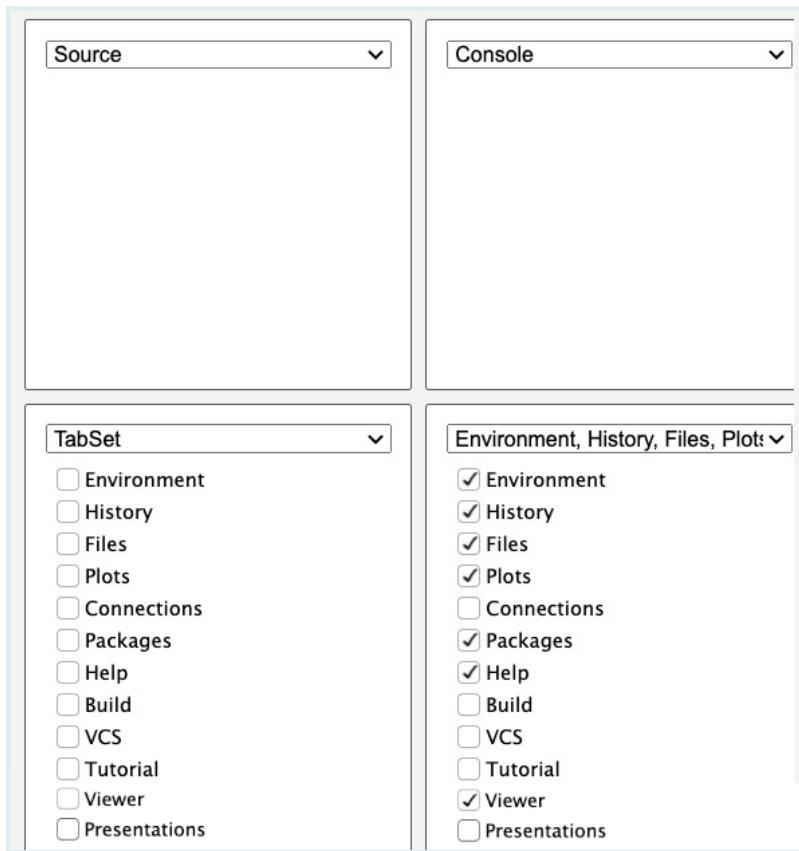
By default, RStudio is arranged into four window panes.

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



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

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

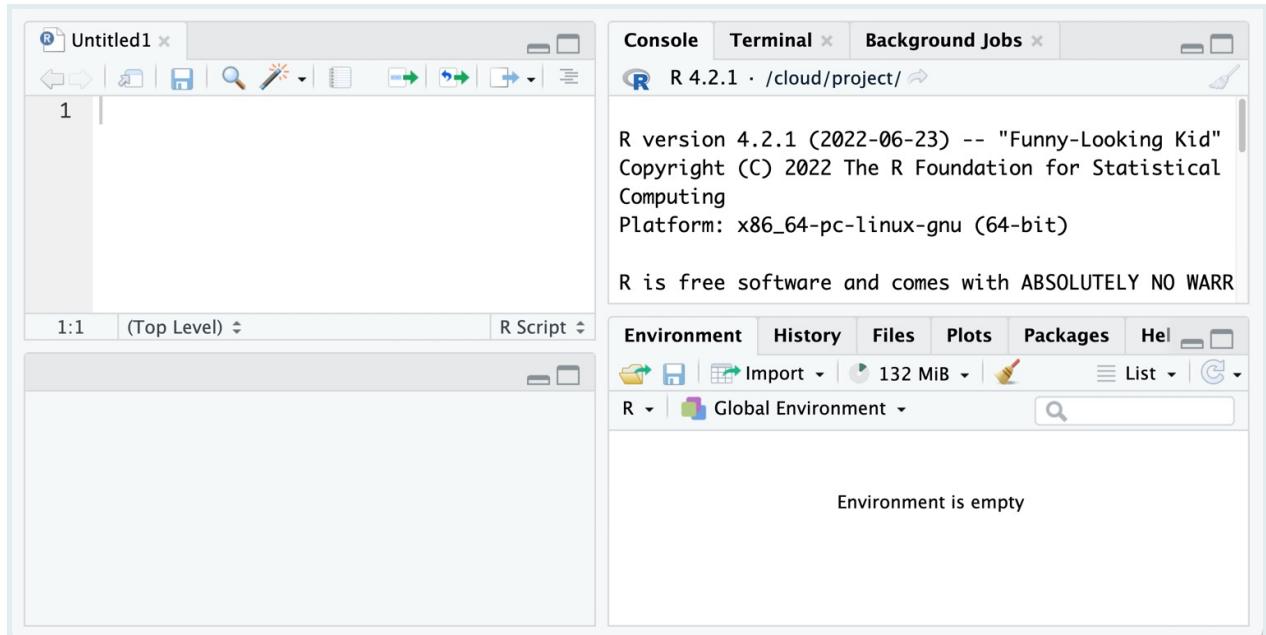


At the top left pane is the Source tab, and at the top right pane, you should have the Console tab.

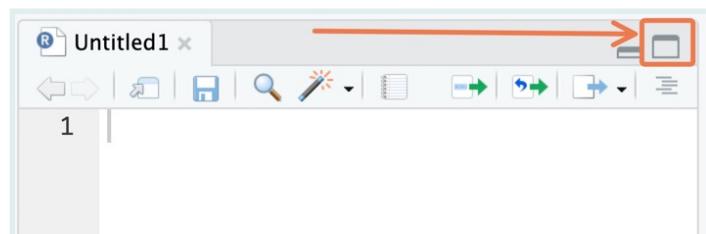
Then at the bottom left pane, no tab options should checked—this section should be left empty, with the drop-down saying just “TabSet”.

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

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



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:

The screenshot shows the RStudio interface with the following panes:

- Editor**: Shows code in `rstudio_intro.R`:

```

1 print("excited for R")
2 print("and RStudio")
3
4 View(women)
5
6 ebola_data <- read.csv("https://tinyurl.com/ebola-data-sample")
7
8 plot(women)
9

```
- Console**: Shows the output of the code run in the R 4.2.1 session:

```

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

```
- Environment**: Shows the global environment with objects:

Object	Type	Value
ebola_data	data frame	200 obs. of 7 variables
\$ id	integer	167 129 2...
\$ age	numeric	55 41 12 ...
- History**: Shows previous commands run in the console:

```

print("excited for R")
print("and RStudio")
View(women)
2 + 2
3 + 3
print("excited for R")

```
- Plots**: Shows a scatter plot of weight vs age.
- Packages**: Shows the system library with installed packages:

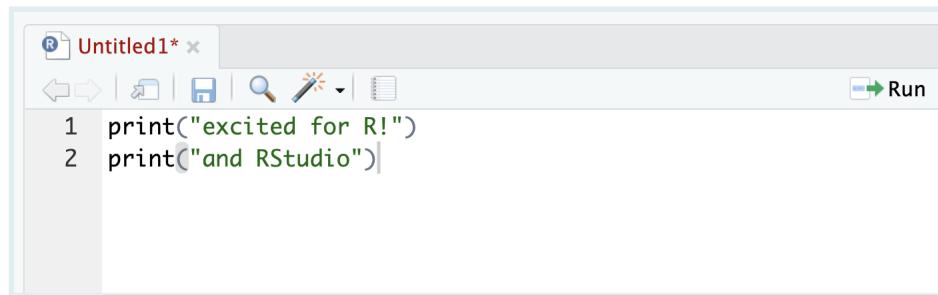
Name	Description	Version
base	The R Base Package	4.2.1
boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-28
class	Functions for Classification	7.3-20
- Files**: Shows the project file system:

Name	Size
..	0 B
.Rhistory	0 B
project.Rproj	205 B
rstudio_intro.R	135 B
- Help**: Shows documentation for the `women` dataset:

Average Heights and Weights for American Women

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

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!")
```

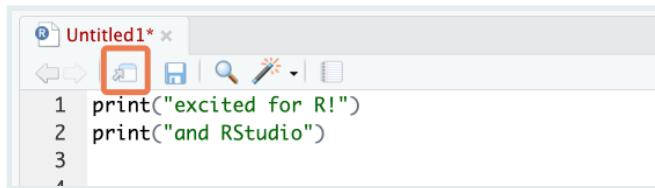
Now drag your cursor to highlight both lines and press Command/Control + Enter.

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.

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’.

The screenshot shows the RStudio interface with the 'Console' tab selected. The title bar indicates 'R 4.1.3 · /cloud/project/'. The console window displays the following text:

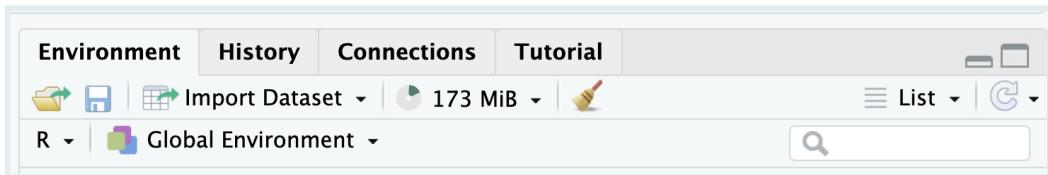
```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

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

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

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

Environment



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

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

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

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

SIDE NOTE



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

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

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

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

The screenshot shows the RStudio interface with the Environment tab selected. A red arrow points to the blue circular dropdown icon next to the dataset name 'ebola_data'. Below the dropdown, the dataset summary is displayed:

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

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

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

```
rm(ebola_data)
```

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

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



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

History

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

The screenshot shows the RStudio interface with the 'Environment' tab selected. The top bar has tabs for Environment, History, Connections, and Tutorial. Below the tabs is a toolbar with icons for file operations like Open, Save, and Print, along with 'To Console' and 'To Source' buttons. The main area displays a history of R commands:

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

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

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

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

Files

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

The screenshot shows the RStudio interface with the 'Files' tab selected. The top bar has tabs for Files, Plots, Packages, Help, Viewer, and Presentation. Below the tabs is a toolbar with icons for New Folder, New Blank File, Upload, Delete, Rename, and More. The main area shows a file tree under 'Cloud > project > chapter_01_getting_started > scripts'. A table lists files with columns for Name, Size, and Modified:

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

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

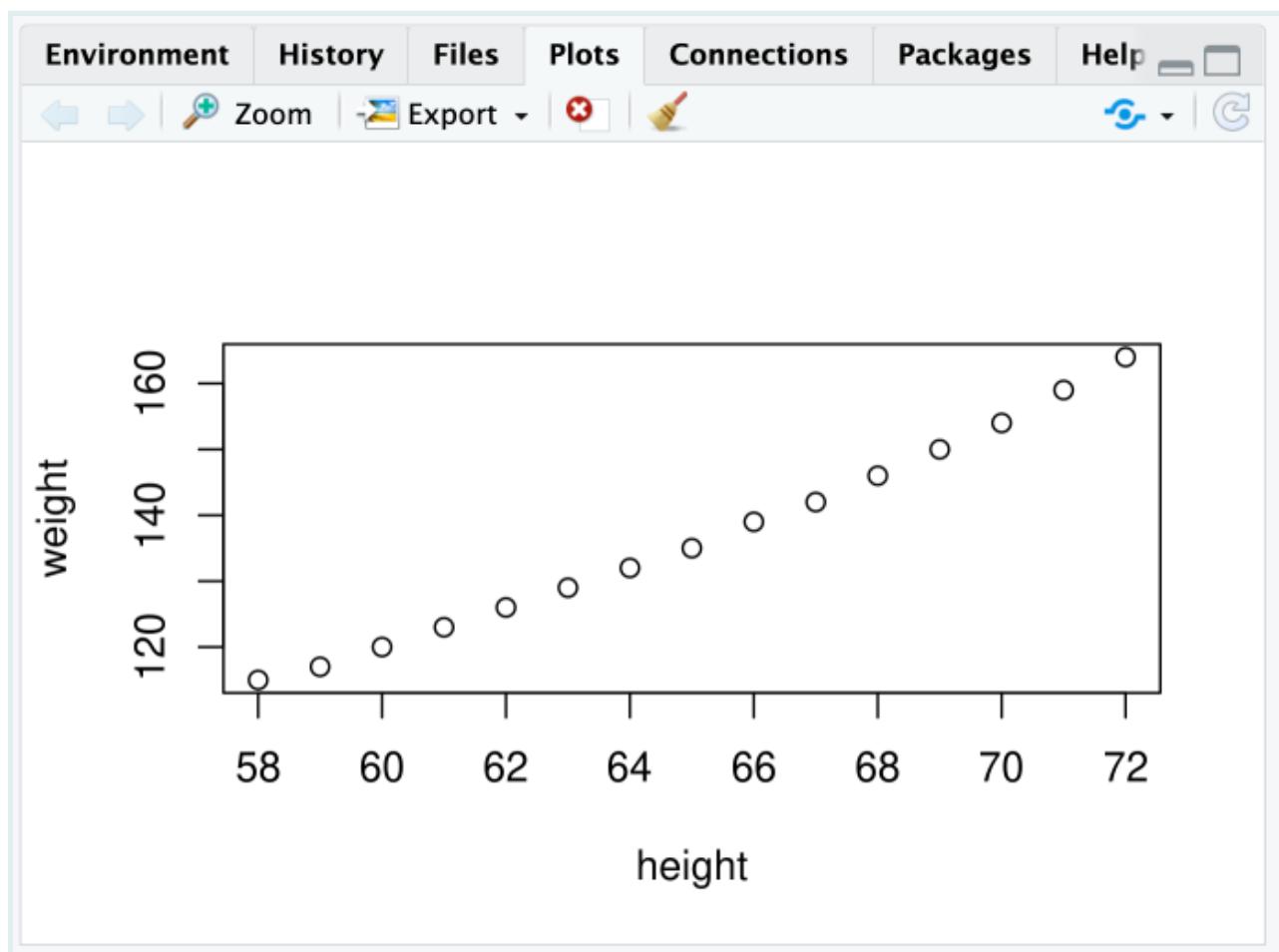
Try playing with some of the buttons here, to see what they do. You should try at least the following:

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

Plots

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

```
plot(women)
```

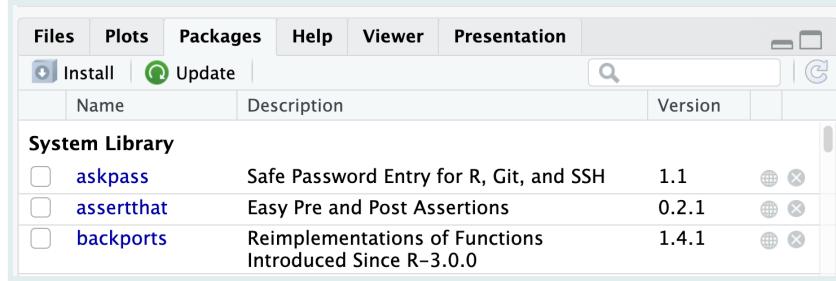


That code creates a plot of the two variables in the `women` dataset. You should see this figure in the Plots tab.

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

Packages

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

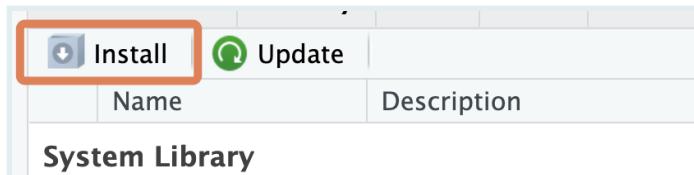


Packages are collections of R code that extend the functionality of R. We will discuss packages in detail in a future lesson.

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)
```

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

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

Viewer

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

Help

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

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

The screenshot shows the RStudio interface with the 'Help' tab selected. The top menu bar includes 'Files', 'Plots', 'Packages', 'Help', 'Viewer', and 'Presentation'. Below the menu is a toolbar with icons for back, forward, search, and refresh. A search bar contains the text 'R: Create a highchart object from a particular data type'. To the right of the search bar is a 'Find in Topic' button. The main content area displays the documentation for the `hchart` function. The title is 'Create a highchart object from a particular data type'. The 'Description' section states: 'hchart uses highchart to draw a particular plot for an object of a particular class in a single command. This defines the S3 generic that other classes and packages can extend.' The page is identified as 'R Documentation'.

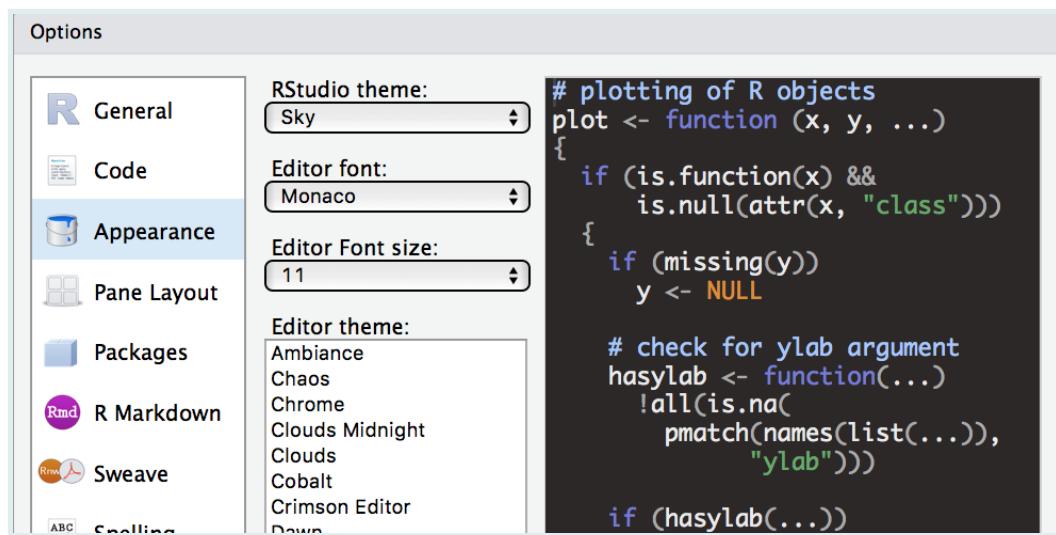
Help files are not always very easy to understand for beginners, but with time they will become more useful.

RStudio options

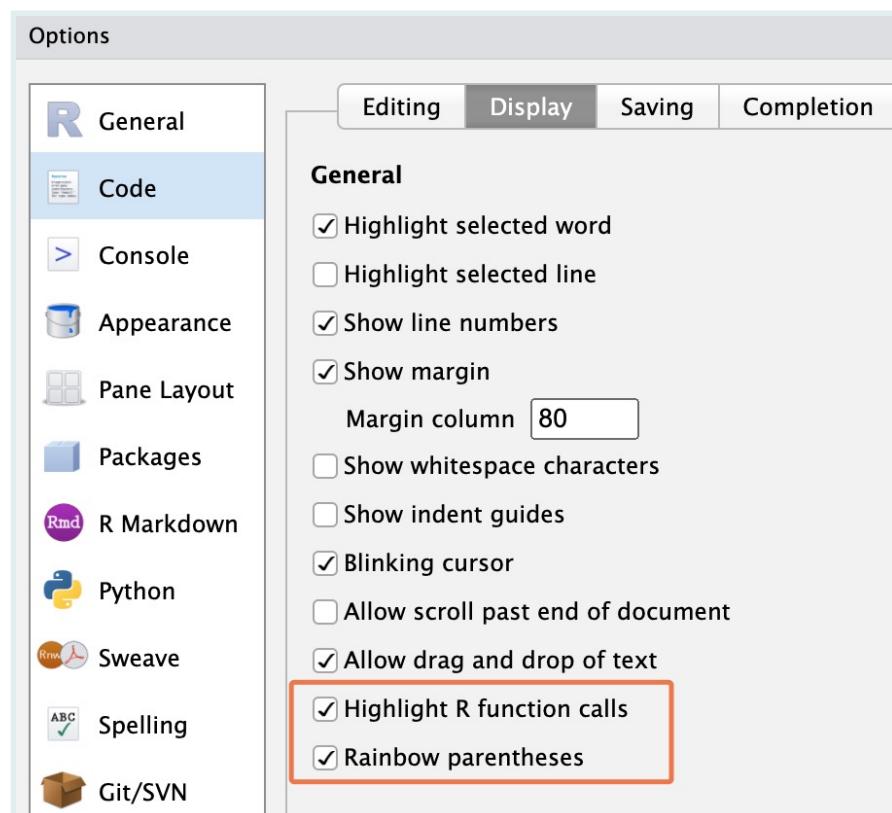
RStudio has a number of useful options for changing its look and functionality. Let's try these. You may not understand all the changes made for now. That's fine.

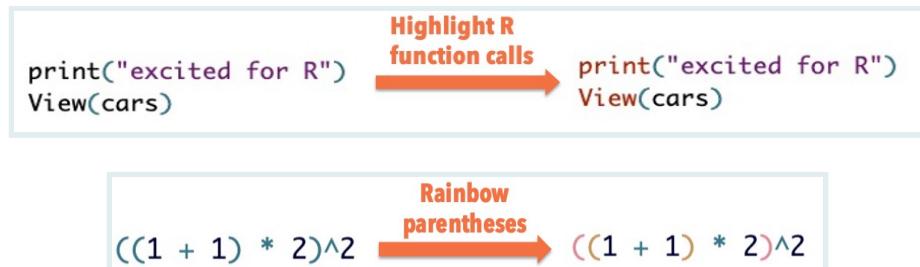
In the RStudio menu at the top of the screen, select Tools > Global Options to bring up RStudio's options.

- 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.





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

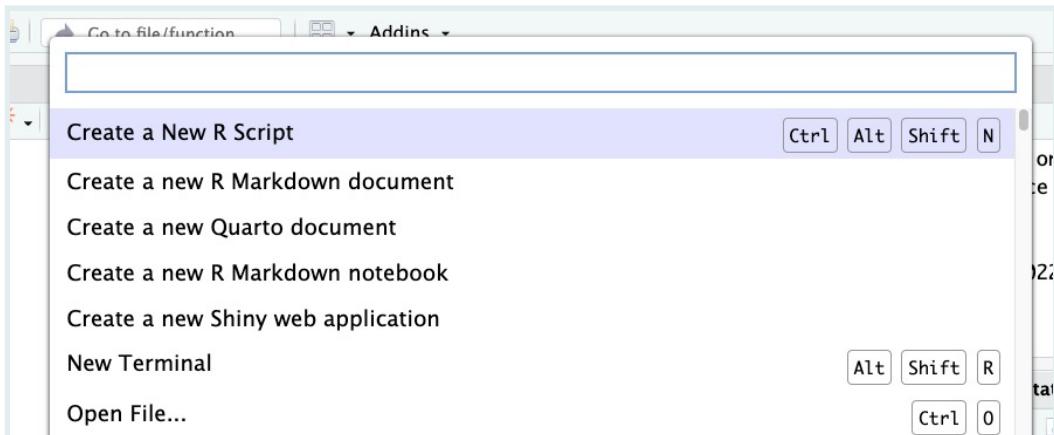
This also means that you never want to “**save your workspace to .RData on exit**”, so set this to **Never**.

Command palette

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

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

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



Try using it to:

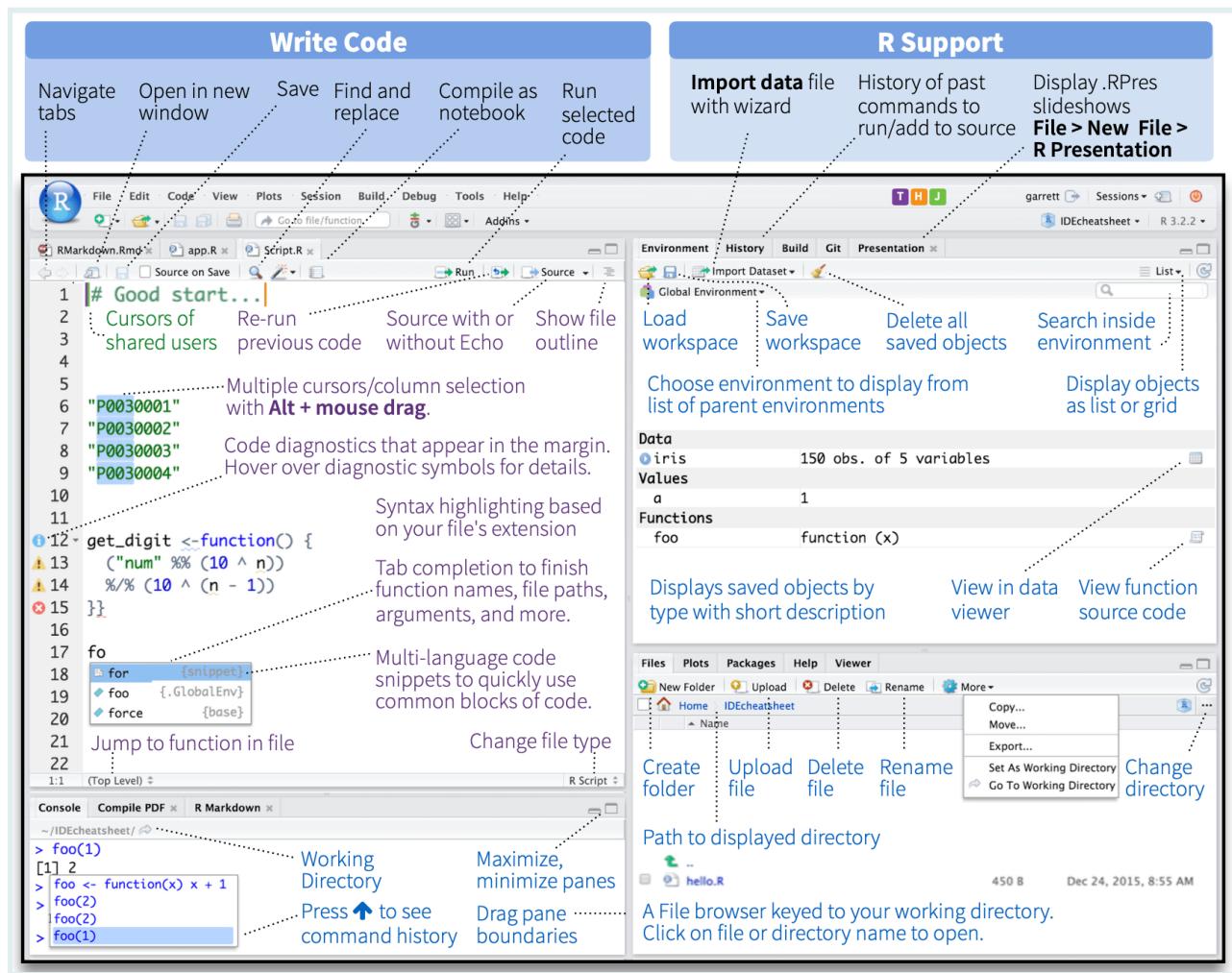
- Create a new script (Search “new script” and click on the relevant option)
- Rename a script (Search “rename” and click on the relevant option)

Wrapping up

Congratulations! You are now a new citizen of RStudio.

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

Below is one section of that sheet:



See you in the next lesson!

Further resources

1. 23 RStudio Tips, Tricks, and Shortcuts

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



LAMECK AGASA

Statistician/Data Scientist

References

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

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

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Lesson notes | Coding basics

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Comments
R s a calculator
Formatting code
Objects in R
Create an object
What is an object?
Datasets are objects too
Rename an object
Overwrite an object
Working with objects
Some errors with objects
Naming objects
Functions
Basic function syntax
Nesting functions
Packages
A first example: the {tableone} package
Full signifiers
pacman::p_load()
Wrapping up

Learning objectives

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

Introduction

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

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



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

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

Comments

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

A comment is text that is ignored by the computer.

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

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

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

Question 1

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

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

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

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

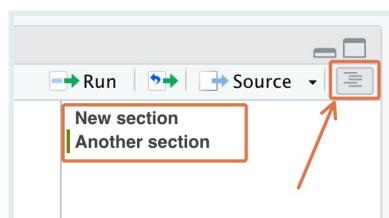
A fantastic use of comments is to separate your scripts into sections. If you put four dashes after a comment, RStudio will create a new section in your code:

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

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



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



R as a calculator

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

```
2 + 2
```

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

```
2 - 2
```

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

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

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

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

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

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

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

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

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

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

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

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

REMINDER



We hope you remember the shortcut to run code!

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

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

Question 2

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

```
2 - 2 / 2
```

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

Formatting code

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

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

```
2+2
```

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

```
2 + 2
```

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

```
2           +           2
```

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

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

```
sqrt(100)
```

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

```
sqrt(    100      )
```

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

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

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

But of course, you should try to space out your code in sensible ways. What exactly is “sensible”? Well, it may be hard for you to know at the moment. Over time, as you read

other people's code, you will learn that there are certain R *conventions* for code spacing and formatting.

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

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
```

WATCH OUT



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

```
> sqrt(100  
+ |
```

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

```
)
```

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

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

Objects in R

Create an object

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

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

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

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

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

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

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

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

SIDE NOTE



```
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 `<-`.

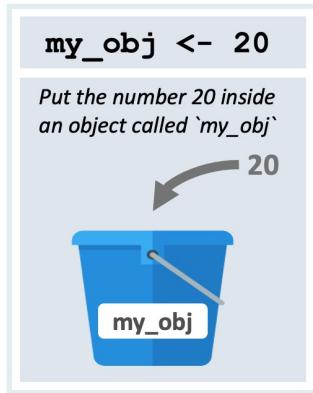
Now that you've created the object `my_obj`, R knows all about it and will keep track of it during this R session. You can view any created objects in the *Environment* tab of RStudio.

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’”.

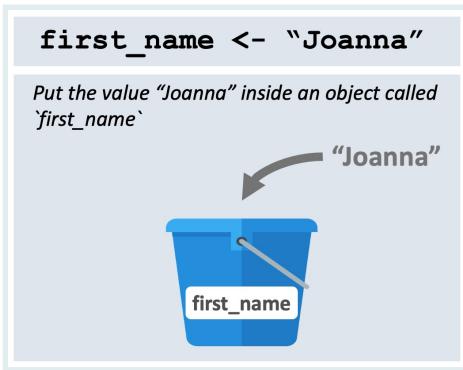


Once the code is run, we would say, in R terms, that “the value of object called `my_obj` is 20”.

And if you run this code:

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

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



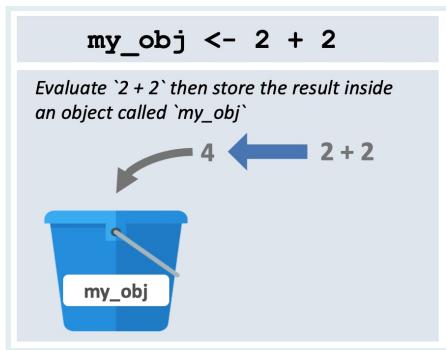
Once the code is run, we would say, in R terms, that “the value of the `first_name` object is Joanna”.

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.



Question 3

Consider the code chunk below:

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

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

- A. `2 + 2 + 2`
- B. numeric
- C. 6

Datasets are objects too

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

Be patient.

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

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

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

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

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

Because you can store datasets as objects, it's 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 there. We will cover this in a future lesson.

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

Rename an object

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

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

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

```
ebola_data <- ebola_sierra_leone_data
```

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

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

```
rm(ebola_sierra_leone_data)
```

Overwrite an object

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

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

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

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

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

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

Working with objects

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

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

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

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

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

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

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

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



SIDE NOTE

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

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

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

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

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

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

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

Question 4

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

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

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

Some errors with objects

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

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

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

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

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

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

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

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

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

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

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

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

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



Question 5

a. The code below returns an error. Why?

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

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

```
my_1st_name <- "Kene"  
my_last_name <- "Nwosu"  
  
paste(my_Ist_name, my_last_name)
```

Naming objects

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

– Phil Karlton.

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

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

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

```
sample_of_the_ebola_outbreak_dataset_from_sierra_leone_in_2014
```

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

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

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

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

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

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

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

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

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

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

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

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

Question 6

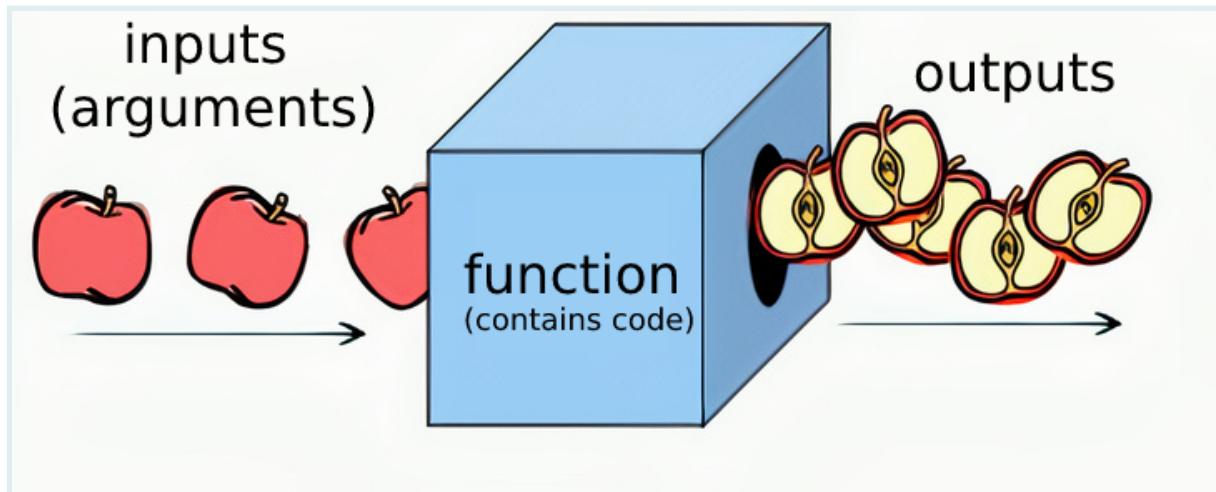
In the code chunk below, we are attempting to take the top 20 rows of the `ebola_data` table. All but one of these lines has an error. Which line will run properly?

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

Functions

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

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



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

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

Basic function syntax

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

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

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

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

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

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

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

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

We can also swap the order of the arguments:

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

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

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

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

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

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

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

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

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

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

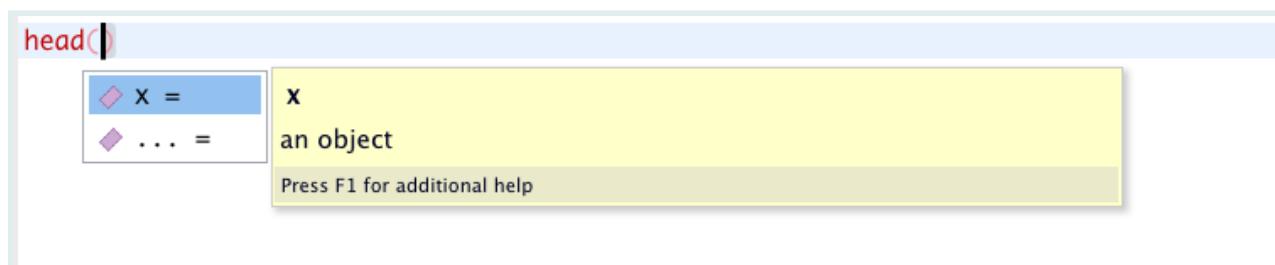
Some function arguments can be skipped altogether, because they have *default* values.

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

```
head(ebola_data)
```

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

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



Question 7

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

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

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

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

So you could have two arguments:

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

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

Or four arguments:

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

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

And so on up to infinity.

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

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

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

PRO TIP



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

Arguments

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

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

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

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

Nesting functions

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

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

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

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

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

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

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

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

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

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

Function nesting will come in very handy soon.

Question 8

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

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

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

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

Packages

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

these packages.

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

A first example: the `{tableone}` package

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

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

```
library(tableone)
```

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

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

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

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

```
##                                     Overall
##   n                               200
##   id (mean (SD))      146.00 (82.28)
##   age (mean (SD))      33.12 (17.85)
##   sex = M (%)           76 (38.0)
##   status = suspected (%) 18 ( 9.0)
##   date_of_onset (%)    1 ( 0.5)
##   2014-05-18             1 ( 0.5)
##   2014-05-20             1 ( 0.5)
##   2014-05-21             1 ( 0.5)
##   2014-05-22             2 ( 1.0)
##   2014-05-23             1 ( 0.5)
##   2014-05-24             2 ( 1.0)
##   2014-05-26             8 ( 4.0)
##   2014-05-27             7 ( 3.5)
##   2014-05-28             1 ( 0.5)
##   2014-05-29             9 ( 4.5)
##   2014-05-30             4 ( 2.0)
##   2014-05-31             2 ( 1.0)
##   2014-06-01             2 ( 1.0)
##   2014-06-02             1 ( 0.5)
##   2014-06-03             1 ( 0.5)
##   2014-06-05             1 ( 0.5)
```

```

##      2014-06-06      5 ( 2.5)
##      2014-06-07      3 ( 1.5)
##      2014-06-08      4 ( 2.0)
##      2014-06-09      1 ( 0.5)
##      2014-06-10     22 (11.0)
##      2014-06-11      1 ( 0.5)
##      2014-06-12      7 ( 3.5)
##      2014-06-13     15 ( 7.5)
##      2014-06-14      8 ( 4.0)
##      2014-06-15      3 ( 1.5)
##      2014-06-16      1 ( 0.5)
##      2014-06-17      4 ( 2.0)
##      2014-06-18      5 ( 2.5)
##      2014-06-19      8 ( 4.0)
##      2014-06-20      7 ( 3.5)
##      2014-06-21      2 ( 1.0)
##      2014-06-22      1 ( 0.5)
##      2014-06-23      2 ( 1.0)
##      2014-06-24      8 ( 4.0)
##      2014-06-25      6 ( 3.0)
##      2014-06-26     10 ( 5.0)
##      2014-06-27      9 ( 4.5)
##      2014-06-28     17 ( 8.5)
##      2014-06-29      7 ( 3.5)
##      date_of_sample (%) 
##      2014-05-23      1 ( 0.5)
##      2014-05-25      1 ( 0.5)
##      2014-05-26      1 ( 0.5)
##      2014-05-27      2 ( 1.0)
##      2014-05-28      1 ( 0.5)
##      2014-05-29      2 ( 1.0)
##      2014-05-31      9 ( 4.5)
##      2014-06-01      6 ( 3.0)
##      2014-06-02      1 ( 0.5)
##      2014-06-03      9 ( 4.5)
##      2014-06-04      4 ( 2.0)
##      2014-06-05      1 ( 0.5)
##      2014-06-06      2 ( 1.0)
##      2014-06-07      2 ( 1.0)
##      2014-06-10      2 ( 1.0)
##      2014-06-11      4 ( 2.0)
##      2014-06-12      3 ( 1.5)
##      2014-06-13      3 ( 1.5)
##      2014-06-14      1 ( 0.5)
##      2014-06-15     21 (10.5)
##      2014-06-16      1 ( 0.5)
##      2014-06-17      5 ( 2.5)
##      2014-06-18     13 ( 6.5)
##      2014-06-19      9 ( 4.5)
##      2014-06-21      8 ( 4.0)
##      2014-06-22      7 ( 3.5)
##      2014-06-23      6 ( 3.0)
##      2014-06-24      6 ( 3.0)
##      2014-06-25      3 ( 1.5)
##      2014-06-27      5 ( 2.5)
##      2014-06-28      2 ( 1.0)

```

```

##      2014-06-29          8 ( 4.0)
##      2014-06-30          6 ( 3.0)
##      2014-07-01          4 ( 2.0)
##      2014-07-02         16 ( 8.0)
##      2014-07-03         13 ( 6.5)
##      2014-07-04          2 ( 1.0)
##      2014-07-05          2 ( 1.0)
##      2014-07-06          1 ( 0.5)
##      2014-07-08          3 ( 1.5)
##      2014-07-12          1 ( 0.5)
##      2014-07-14          1 ( 0.5)
##      2014-07-17          1 ( 0.5)
##      2014-07-21          1 ( 0.5)
##    district (%)
##      Bo                  4 ( 2.0)
##      Kailahun           146 (73.0)
##      Kenema              41 (20.5)
##      Kono                2 ( 1.0)
##      Port Loko            2 ( 1.0)
##      Western Urban        5 ( 2.5)

```

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

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

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

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

SIDE NOTE



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

Full signifiers

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

So for example, instead of writing:

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

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

```
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)
```

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.

`pacman::p_load()`

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

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

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

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

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

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

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

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

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

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

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

We can do this with the following code:

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

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

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

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

Phew! Hope your head is still intact.

Question 10

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

A.

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

B.

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

C.

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

Wrapping up

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

Answers

1. True.
2. The division sign is evaluated first.
3. The answer is C. The code `2 + 2 + 2` gets evaluated before it is stored in the object.
4. a. The value is 1. The code evaluates to `9-8`.
b. `table(ebola_data$district)`
5. a. You cannot add two character strings. Adding only works for numbers.
b. `my_1st_name` is typed with the number 1 initially, but in the `paste()` command, it is typed with the letter "l".

6. The third line is the only line with a valid object name: `top_20_rows`
7. The last line, `head(6, women)`, is invalid because the arguments are in the wrong order and they are not named.
8. The third code chunk has a problem. It attempts to find the square root of a character, which is impossible.
9. The first line, A, is the correct interpretation.
10. The first code chunk is the recommended way to install and load the package `{janitor}`

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



LAMECK AGASA

Statistician/Data Scientist



OLIVIA KEISER

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

References

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

- “File:Apple slicing function.png.” *Wikimedia Commons, the free media repository*. 1 Oct 2021, 04:26 UTC. 20 Mar 2022, 17:27 <https://commons.wikimedia.org/w/index.php?title=File:Apple_slicing_function.png&oldid=594767630>.
- “PsyteachR | Data Skills for Reproducible Research.” 2021. Github.io. 2021. <https://psyteachr.github.io/reprores-v2/index.html>.
- Douglas, Alex, Deon Roos, Francesca Mancini, Ana Couto, and David Lusseau. 2022. “An Introduction to R.” Intro2r.com. January 27, 2022. <https://intro2r.com/>.

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Lesson notes | Data dive: Ebola in Sierra Leone

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Script setup
Header
Packages
Importing data into R
Intro to reproducibility
Quick data exploration
vis_dat()
inspect_cat() and inspect_num()
Analyzing a single numeric variable
Extract a column vector with \$
Basic operations on a numeric variable
Visualizing a numeric variable
Analyzing a single categorical variable
Frequency tables
Visualizing a categorical variable
Answering questions about the outbreak
Haven't had enough?
Wrapping up

Learning objectives

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

Introduction

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

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

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

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

Script setup

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



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

Empty your environment at the start of the analysis

SIDE NOTE



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

Header

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

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

Packages

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

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

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 to use `p_load()` from `{pacman}` without formally loading the `{pacman}` package, so we type `pacman::p_load()`.

REMINDER



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()`.)

Importing data into R

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

About the Ebola dataset

SIDE NOTE



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

SIDE NOTE

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

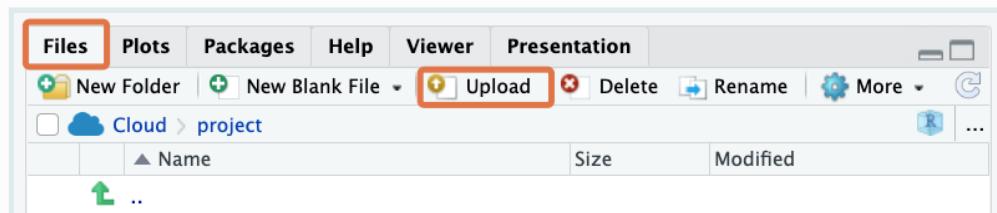
Go to bit.ly/view-ebola-data to view the dataset you will be working on. Then click the download icon at the top to download it to your computer.

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

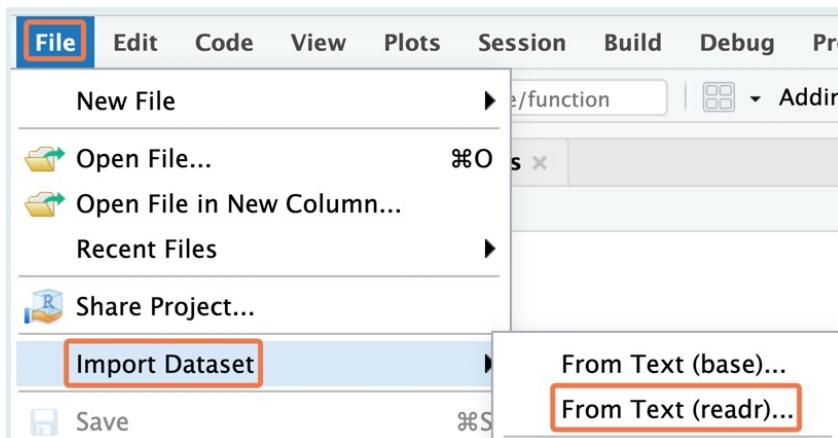
You can leave the dataset in your downloads folder, or move it to somewhere more respectable; the upcoming steps will work independent of where the data is stored. In the next lesson, you will learn how to organize your data analysis projects properly, and we will think about the ideal folder setup for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

RSTUDIO CLOUD

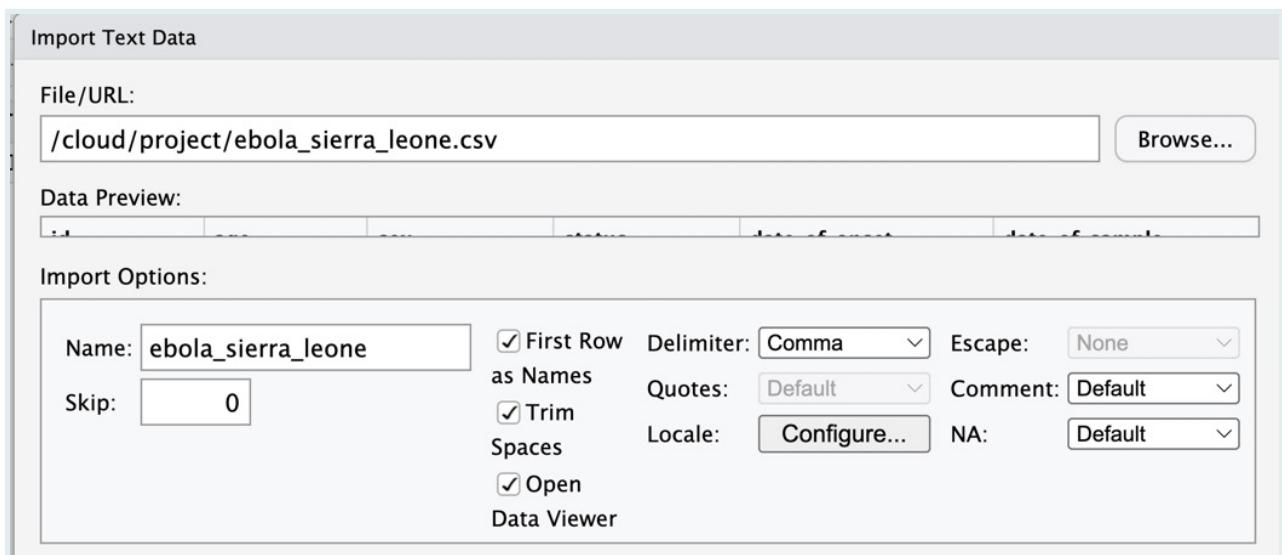
NOTE: If you are using RStudio Cloud, you need to upload your dataset to the cloud. Do this in the “Files” tab by clicking on the “Upload” button.



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



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



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:

	<code>id</code>	<code>age</code>	<code>sex</code>	<code>status</code>	<code>date_of_onset</code>	<code>date_of_sample</code>	<code>district</code>
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")
```

Nice work so far!

Your R script should look similar to this:

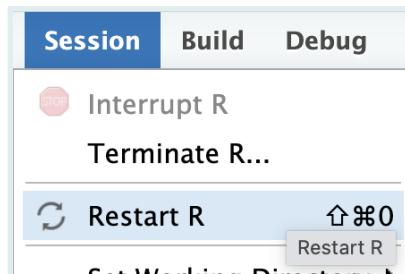
```
# Ebola Sierra Leone analysis  
# John Sample-Name Doe  
# 2024-01-01  
  
# Load packages ----  
if(!require(pacman)) install.packages("pacman")  
pacman::p_load(  
  tidyverse,  
  inspectdf,  
  plotly,  
  janitor,  
  visdat  
)  
  
# Load data ----  
ebola_sierra_leone <-  
  read_csv("~/Downloads/ebola_sierra_leone.csv")
```



Intro to reproducibility

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

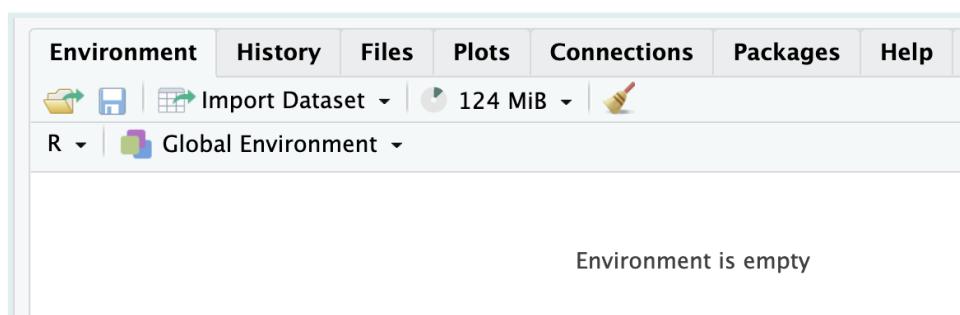
Try restarting R and rerunning the script now. Save your script with Control/Command + s, then *restart* R with the RStudio Menu, at Session > Restart R. On RStudio Cloud, the menu option looks like this:



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

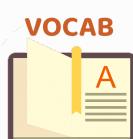
```
Restarting R session...  
> |
```

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



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

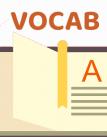
If this worked, congratulations; you have the beginnings of your first “reproducible” analysis script!



What does “reproducible” mean?

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

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

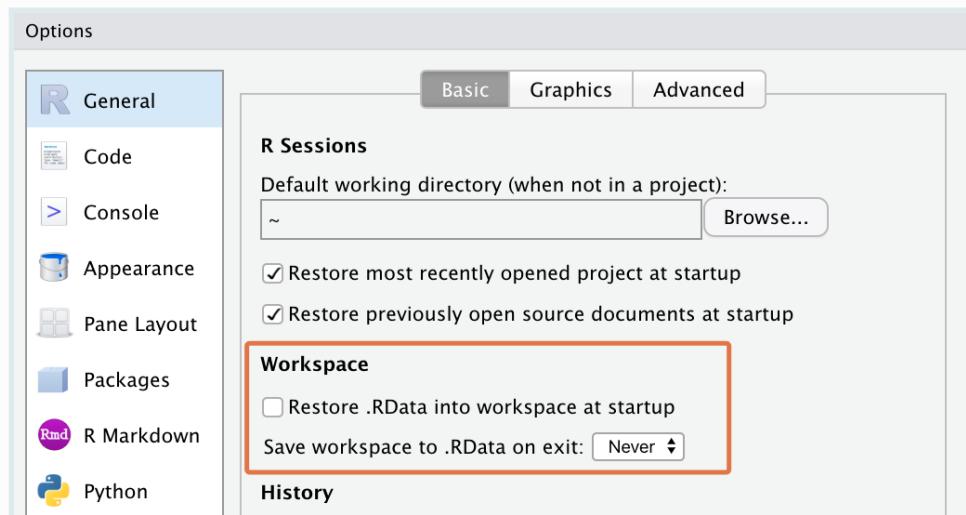


Of course, your script so far is not yet *entirely* reproducible, because the file path for the dataset (the one that looks like this: "...intro-to-data-analysis-with-r/ch01_getting_started/data...") is specific to just your computer. Later on we will see how to use relative file paths, so that the code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

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".

WATCH OUT



Quick data exploration

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

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

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

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

```
tail(ebola_sierra_leone)
```

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

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

```
View(ebola_sierra_leone)
```

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

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

You can close this tab and return to your script.

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

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

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

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

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

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

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

REMINDER



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

```
?ncol
```

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

```
summary(ebola_sierra_leone)
```

```
##           id                  age                  sex                  status
date_of_onset
##  Min.   : 1.00   Min.   : 1.80   Length:200               Length:200
##  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_sample          district
##  Min.   :2014-05-23  Length:200
##  1st Qu.:2014-06-07  Class  :character
##  Median :2014-06-18  Mode   :character
```

```
##   Mean     :2014-06-17  
##   3rd Qu.:2014-06-29  
##   Max.    :2014-07-17  
##
```

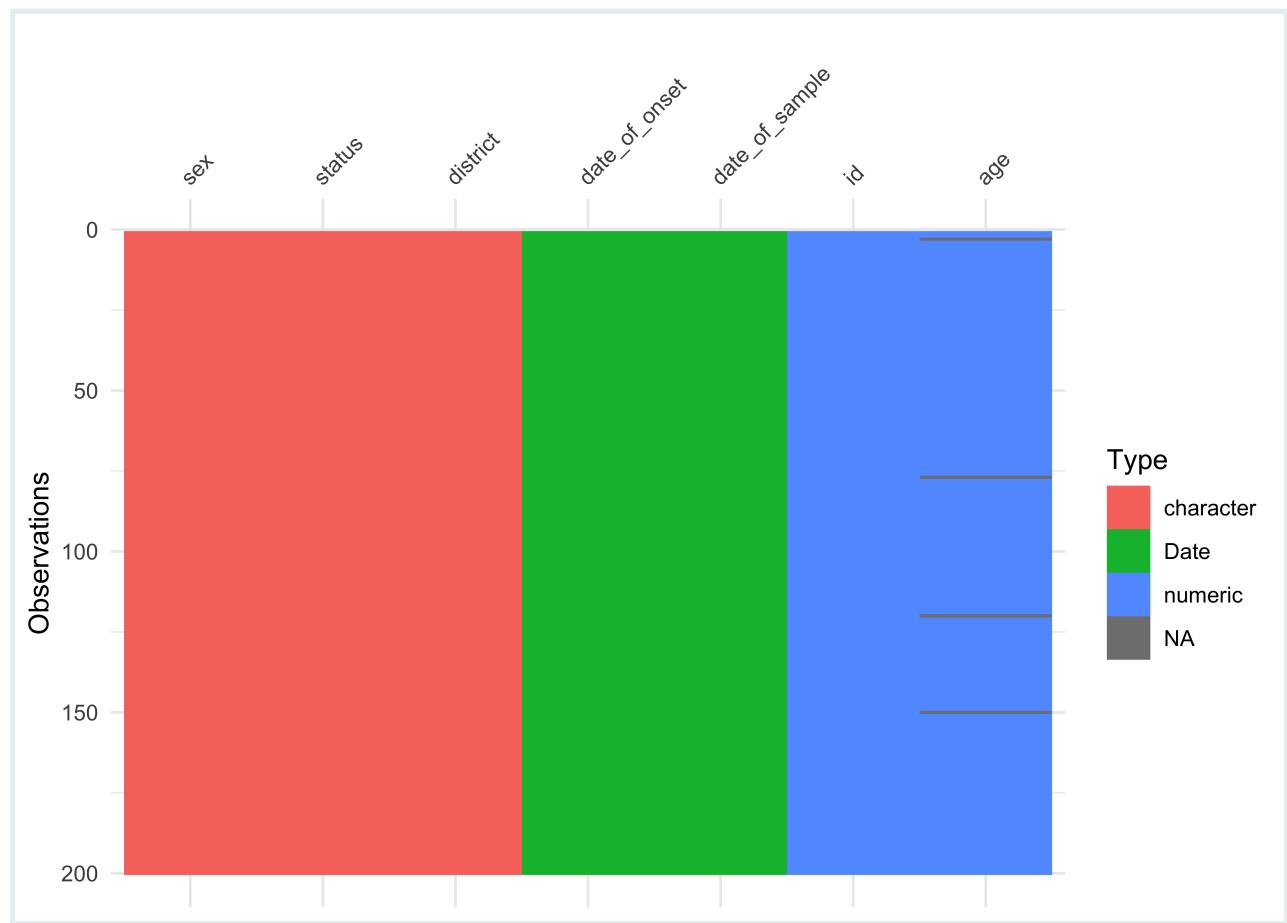
As you can see, for numeric columns in your dataset, `summary()` gives you the minimum value, the maximum value, the mean, median and the 1st and 3rd quartiles.

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

```
vis_dat()
```

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

```
vis_dat(ebola_sierra_leone)
```



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

```
inspect_cat() and inspect_num()
```

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

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

```
inspect_cat(ebola_sierra_leone)
```

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

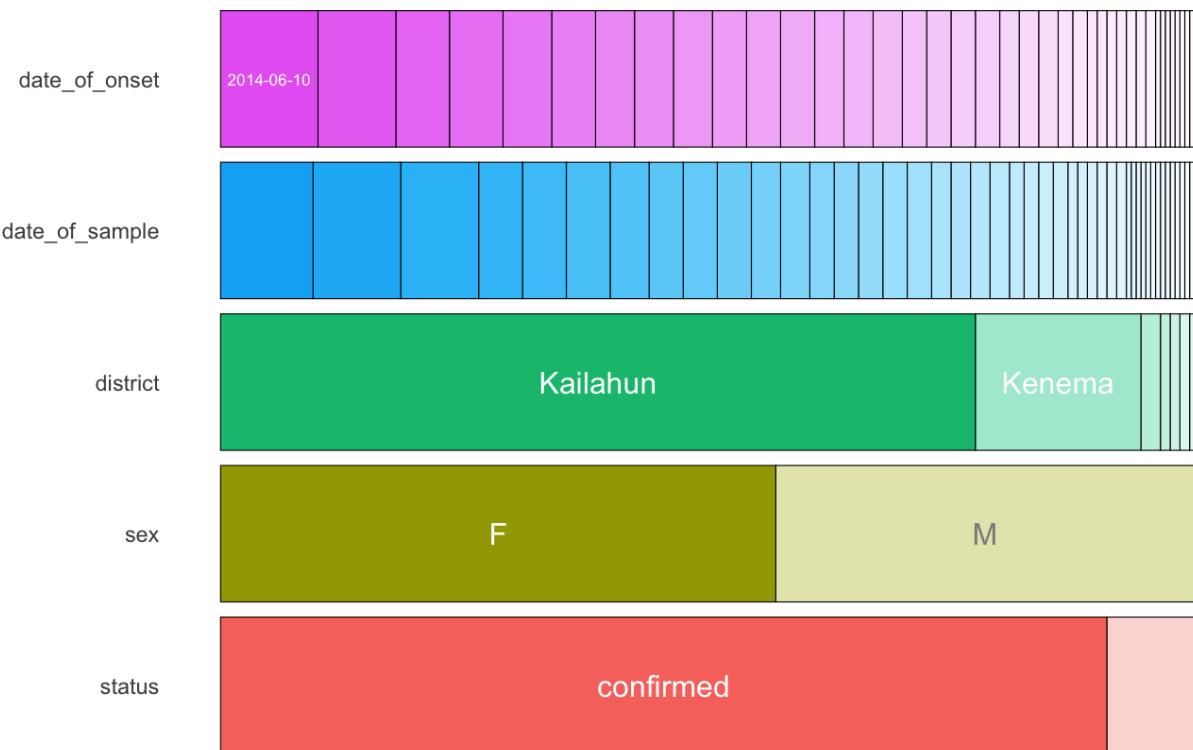
But the magic happens when you run `show_plot()` on the result from `inspect_cat()`:

```
# 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)
```

Frequency of categorical levels in df::ebola_sierra_leone

Gray segments are missing values



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

SIDE NOTE

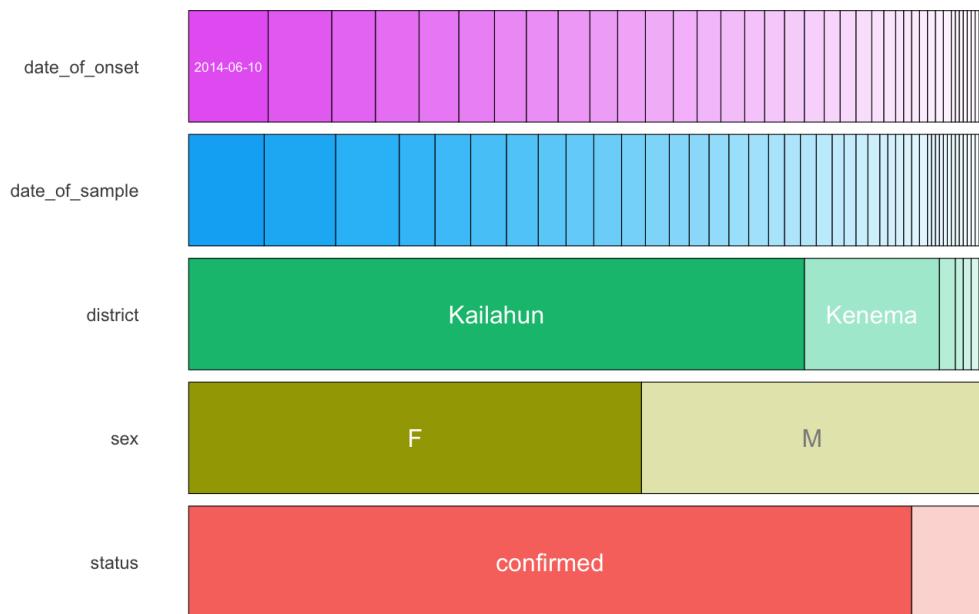


You could also run:

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

SIDE NOTE

Frequency of categorical levels in df::ebola_sierra_leone
Gray segments are missing values



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:

status

col_name: status
prop: 0.090
new_level_key: suspected-status-ebola_sierra_leone

REMINDER

REMINDER

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

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

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

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

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

Analyzing a single numeric variable

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

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

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

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

Extract a column vector with \$

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

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

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

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

VOCAB



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

Basic operations on a numeric variable

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

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

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

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

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

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

Great! This need to remove the `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
```

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

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

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

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

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

Now run these functions on age_vec and observe their outputs:

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

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

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

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

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

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

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

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

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

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

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

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

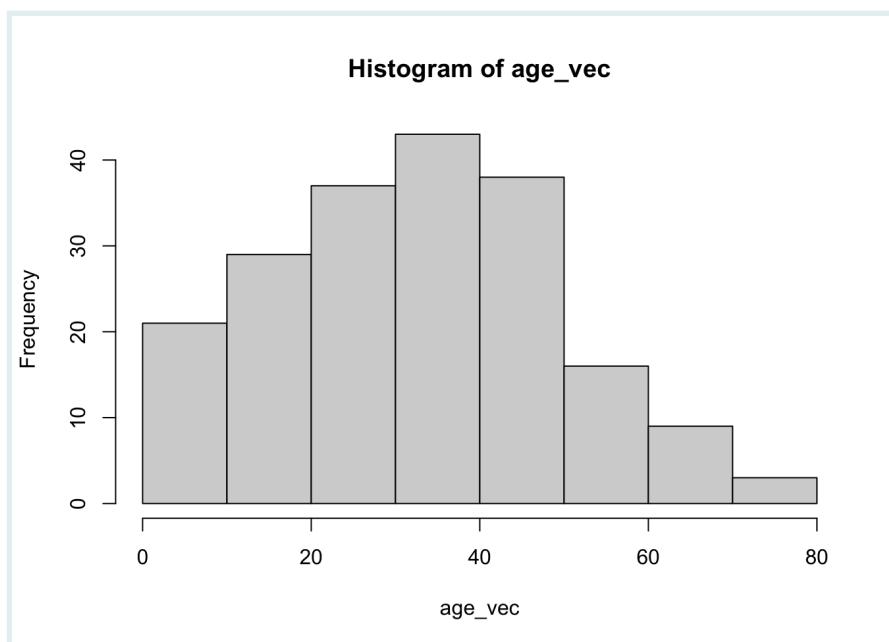
Do not feel intimidated by the long list of functions! You should not have to memorize them; rather you should feel free to Google the function for whatever operation you want to carry out. You might search something like “what is the function for standard deviation in R”. One of the first results should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

Visualizing a numeric variable

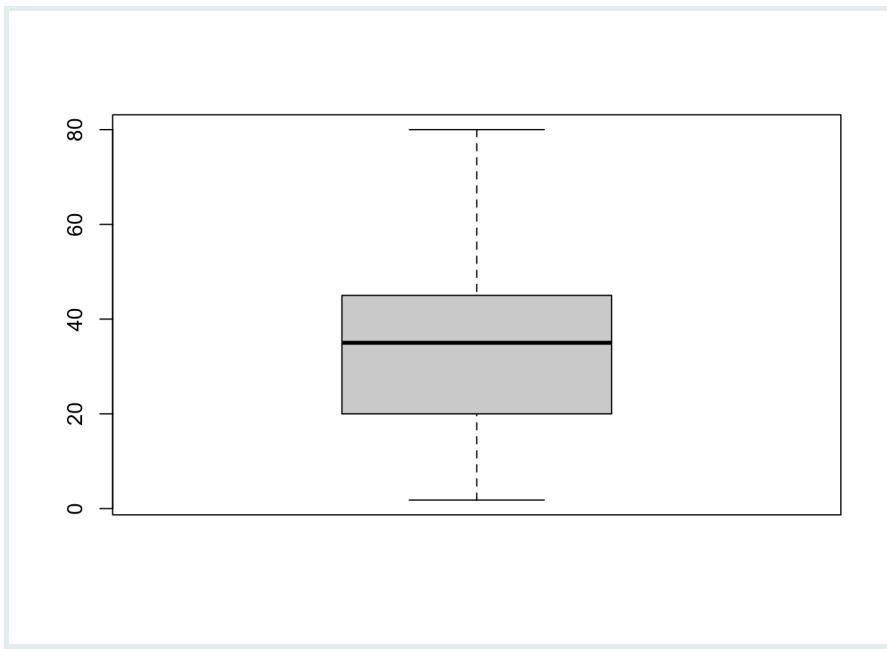
Now let's create a graph to visualize the age variable. The two most common graphics for inspecting the distribution of numerical variables are [histograms](#) (like the output of the `inspect_num()` function you saw earlier) and [boxplots](#).

R has built-in functions for these:

```
hist(age_vec)
```



```
boxplot(age_vec)
```



Nice and easy!

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

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

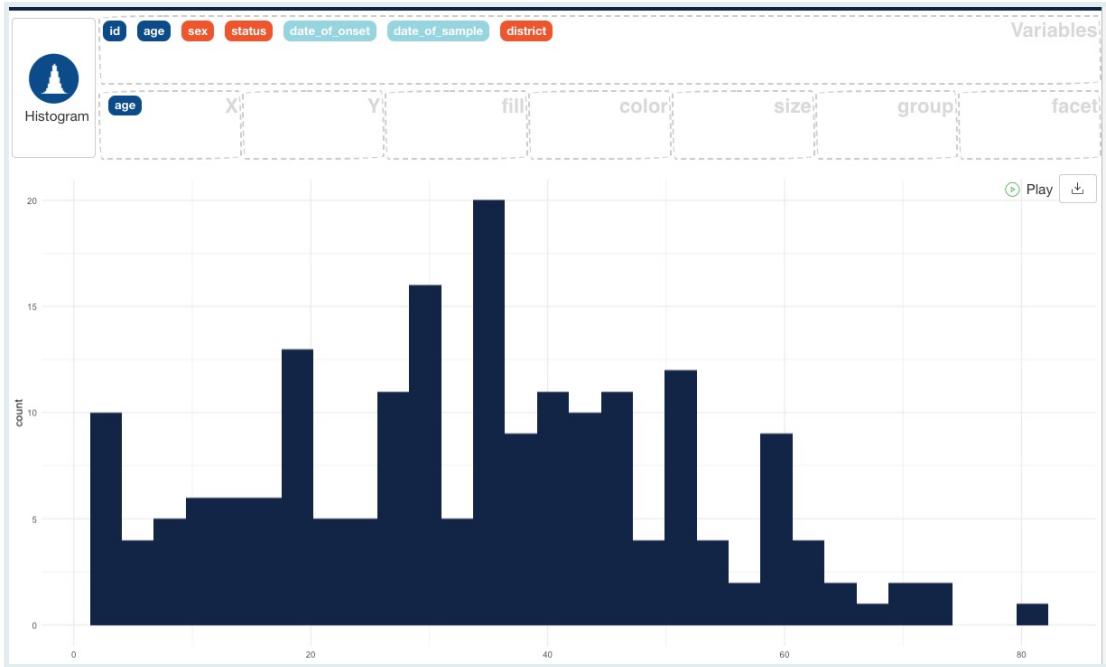
The workhorse function of the `{esquisse}` package is `esquisser()`, and this function takes a single argument—the dataset you want to visualize. So we can run:

```
esquisser(ebola_sierra_leone)
```

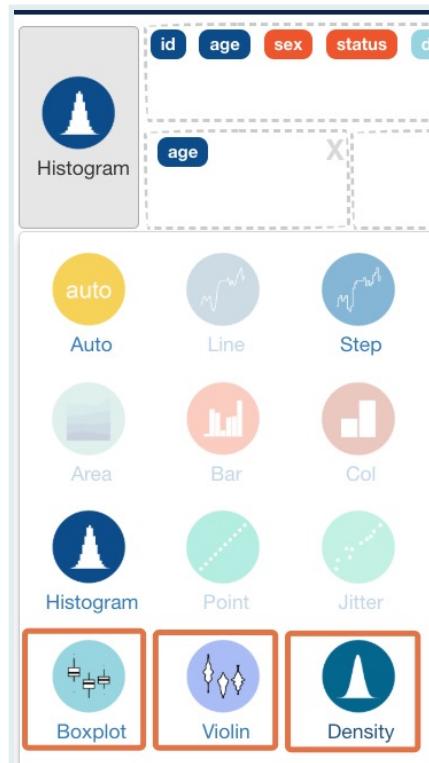
This should bring a graphic user interface that you can use to plot different variables. To visualize the age variable, simply drag `age` from the list of variables into the x axis box:



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



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



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

The screenshot shows the Esquisse interface with a histogram visualization. The top right corner has a large orange number '2.' and a 'Copy to clipboard' button. Below it is a 'Code:' section containing R code for creating a histogram. At the bottom left is a 'Data' tab and at the bottom right is a 'Code' tab, both with orange outlines. A blue button labeled 'Insert code in script' is also visible.

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

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

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

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



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

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

Easy bivariate and multivariate plots



CHALLENGE 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.

Analyzing a single categorical variable

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

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

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

```

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

```

Sorry for printing that very long vector!

Frequency tables

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

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

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

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

`table()` is 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.

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

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



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

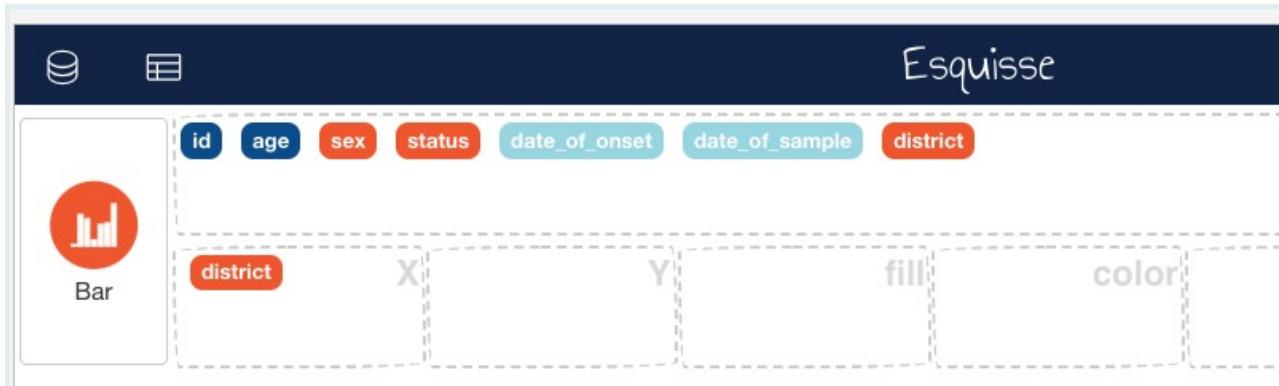
The output shows us that there were 0 women in the Bo district, 2 men in the Bo district, 91 women in the Kailahun district, and so on.

Visualizing a categorical variable

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

```
esquisser(ebola_sierra_leone)
```

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



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

Answering questions about the outbreak

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

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

Solutions

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

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

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

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

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

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

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

The median age of cases was 35.

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

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

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

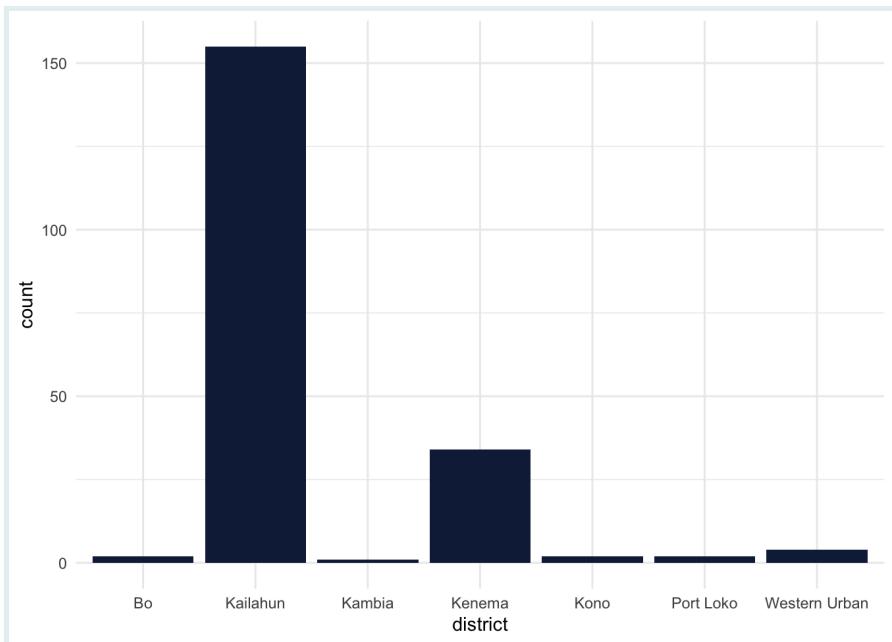
As seen in the table, there were more cases in women. Specifically, 57% of cases are of women.

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

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

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

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

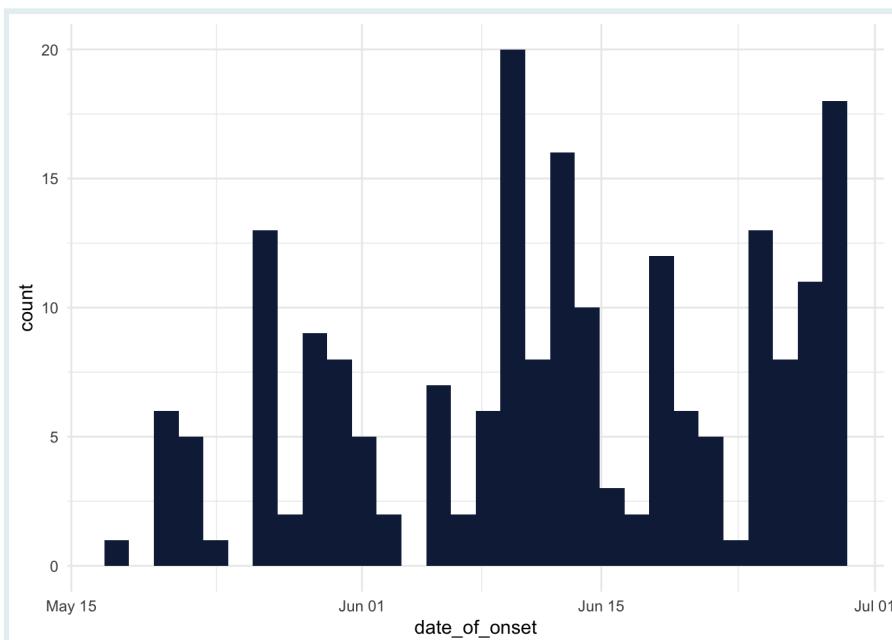


As seen, the Kailahun district had the majority of cases.

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

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

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



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

Haven't had enough?

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

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

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

Wrapping up

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

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

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

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

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Lesson notes | RStudio projects

Created by the GRAPH Courses team

January 2023

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

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

Getting started: RStudio projects
Learning objectives
Introduction
Creating a new RStudio Project
Creating Project subfolders
Adding a dataset to the “data” folder
Creating a script in the “scripts” folder
Importing data from the “data” folder
Exporting data to the “outputs” folder
Exporting plots to the “outputs” folder
Sharing a Project
Wrapping up

Getting started: RStudio projects

Learning objectives

1. You can set up an RStudio Project and create sub-directories for input data, scripts and analytic outputs.
2. You can import and export data within an RStudio Project.
3. You understand the difference between relative and absolute file paths.
4. You recognize the value of Projects for organizing and sharing your analyses.

Introduction

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

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

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

Let's get started.

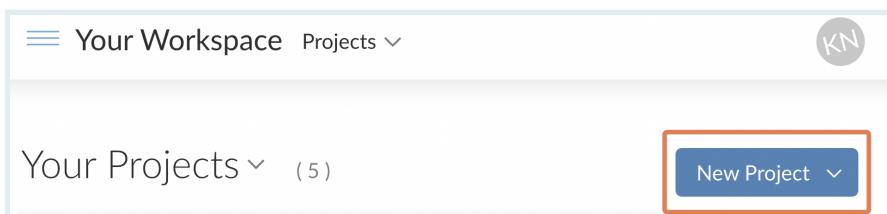
Creating a new RStudio Project

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

On RStudio Cloud

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

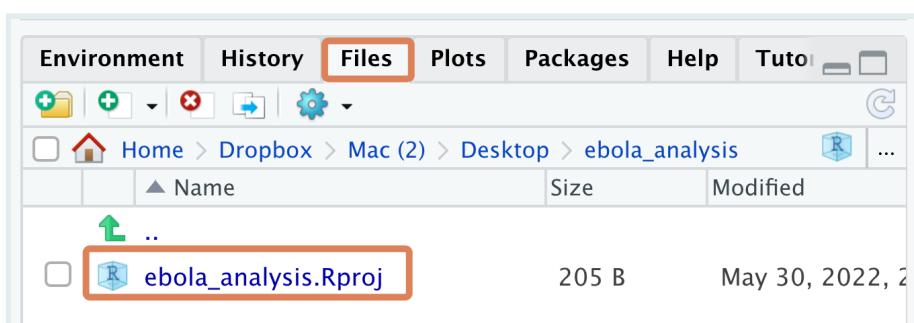
The steps are pretty simple: go to your Cloud homepage, rstudio.cloud, and click on the "New Project" button.



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



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:



KEY POINT



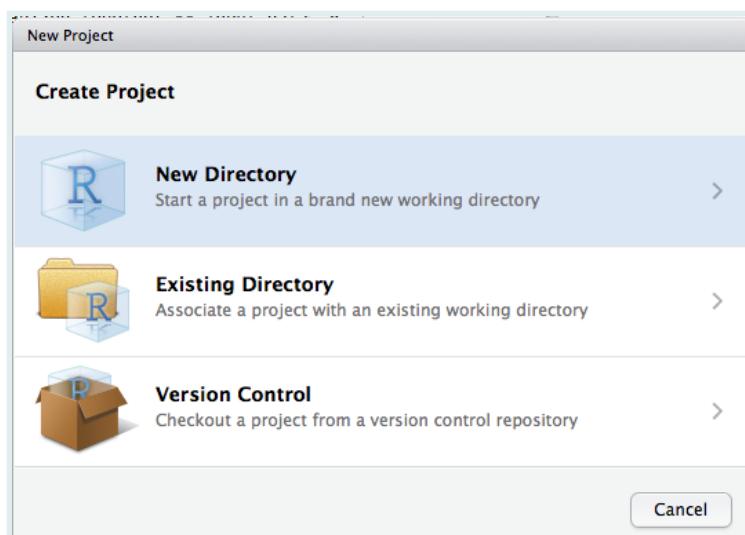
The `.RProj` file is what turns a regular computer folder into an "RStudio Project".

On a local computer

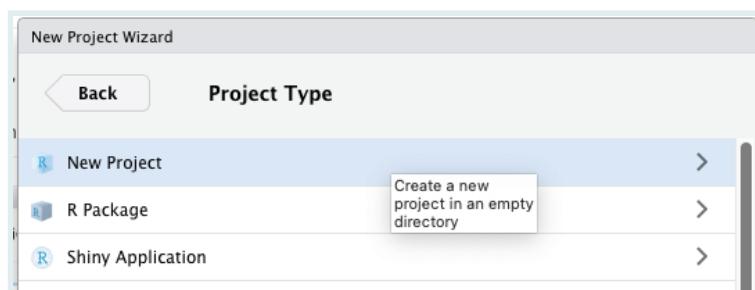
If you are on a local computer, open RStudio, then on the RStudio menu, go to `File > New Project`. Your options may look a little different from the screenshots below depending on your operating system.



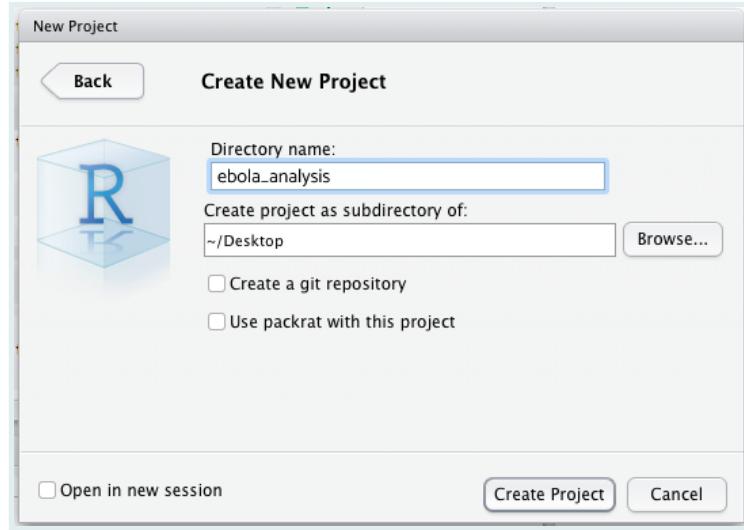
Choose “New directory”



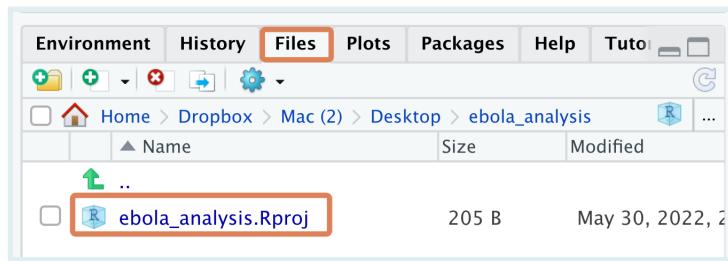
Then choose “New Project”:



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:



Click on the .Rproj file to open your project

KEY POINT



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

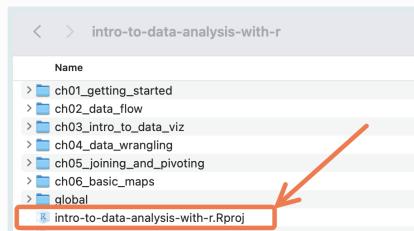
From now on, to open your project, you should double click on this .RProj file from your computer’s Finder/File Explorer.

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

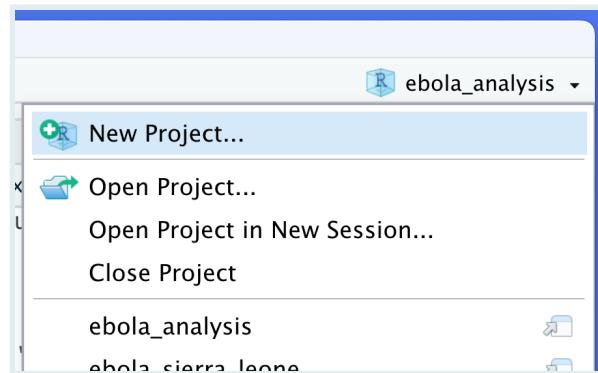
KEY POINT



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

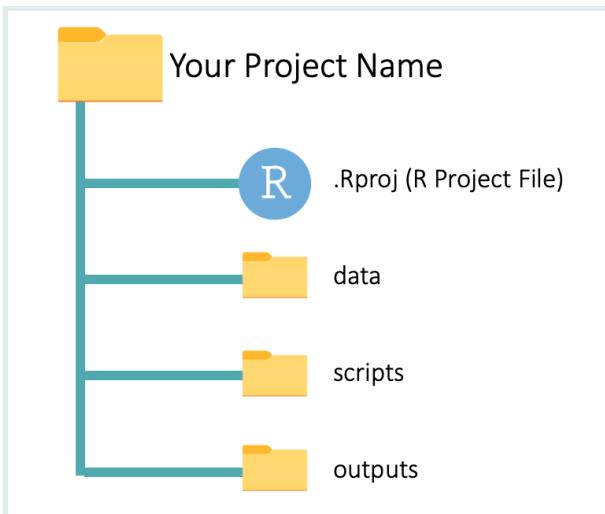


Note also that there is a header at the top right of RStudio window that tells you which Project you currently have open. Clicking on this gives you some additional Project options. You can create a new project, close a project and open recent projects, among other options.



Creating Project subfolders

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



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

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

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



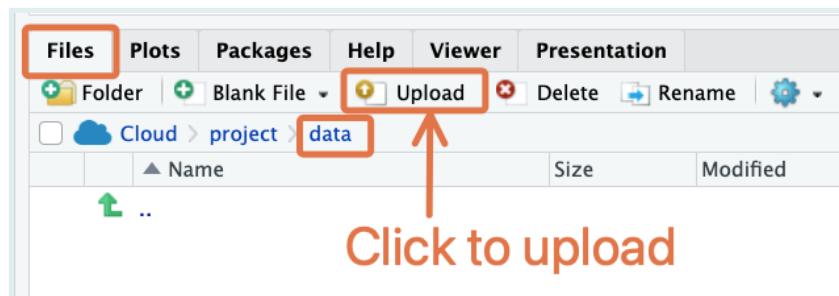
Adding a dataset to the “data” folder

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

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

On RStudio Cloud

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

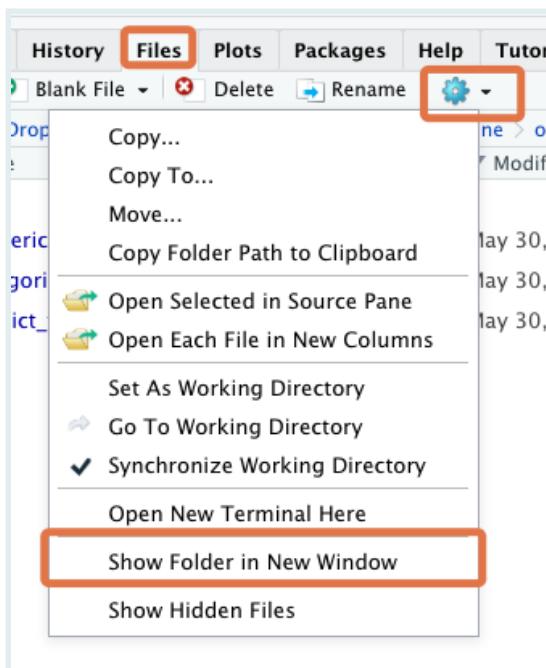


This will bring up a dialog box where you can select the file for upload.

On a local computer

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

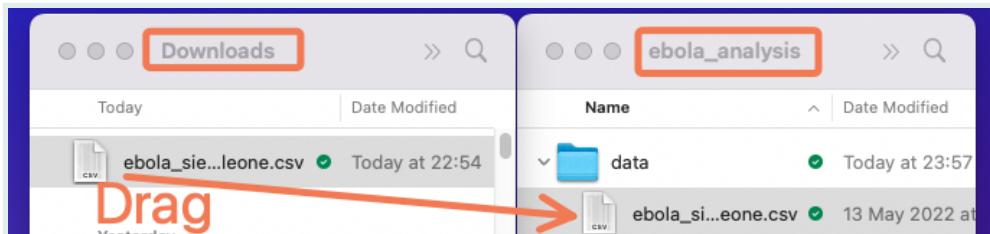
- First, locate the Project folder with your computer’s File Explorer/Finder. If you’re having trouble locating this, RStudio can help: go to the “Files” tab, click on “More” (the gear icon), then click “Show Folder in New Window”.



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

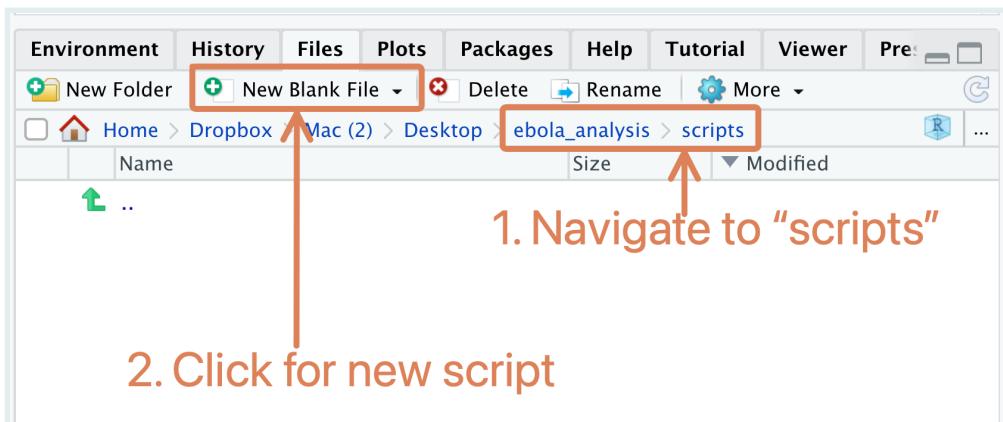
- Now, move the Ebola dataset you downloaded in the previous lesson to the newly-created “data” sub-folder.

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



Creating a script in the “scripts” folder

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

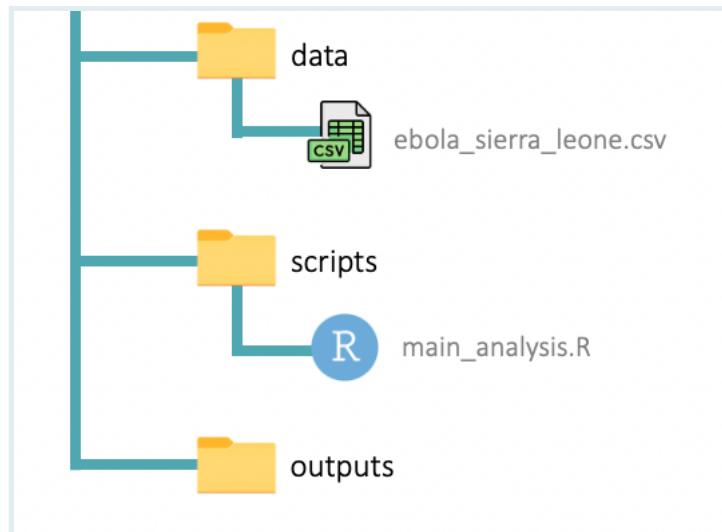


SIDE NOTE



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

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



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

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

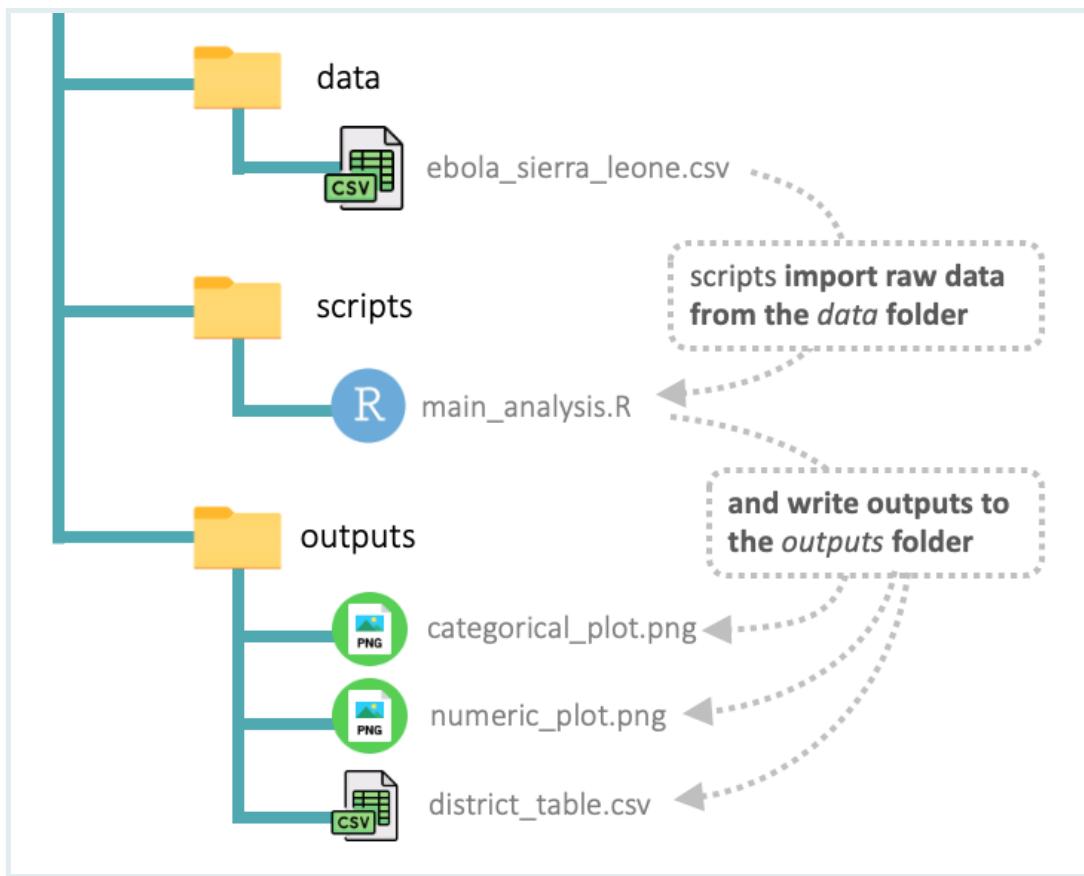


Figure: Data flow in an R project. Scripts in the “scripts” folder import data from “data” folder and export data and plots to the “outputs” folder

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

Importing data from the “data” folder

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

```

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

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

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

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

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

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

```

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

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

```

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

```

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

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

```

# Load data ----
ebola_sierra_leone <- read_csv("") # DATA PENDING! WE WILL UPDATE THIS
# Which districts have the most
district_vec <- ebola_sierra_leone
tabyl(district_vec)

```

Click on the “data” folder, then press Tab again. Since you only have one file in the “data” folder, RStudio should automatically fill in its 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:

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.

KEY POINT



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

KEYPOINT

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

RSTUDIO CLOUD

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

Using `here::here()`

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

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

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

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

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

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

Exporting data to the “outputs” folder

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

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

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

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

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

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

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

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

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

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

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

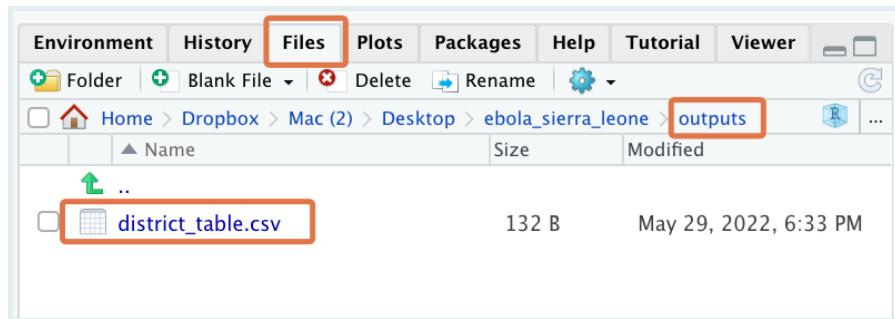
SIDE NOTE



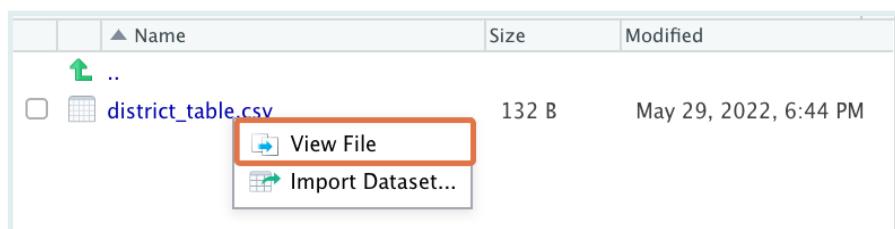
You can replace "district_table.csv" with any other appropriate name, for example "freq table across districts.csv":

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

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



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



This should bring up an RStudio viewer window:

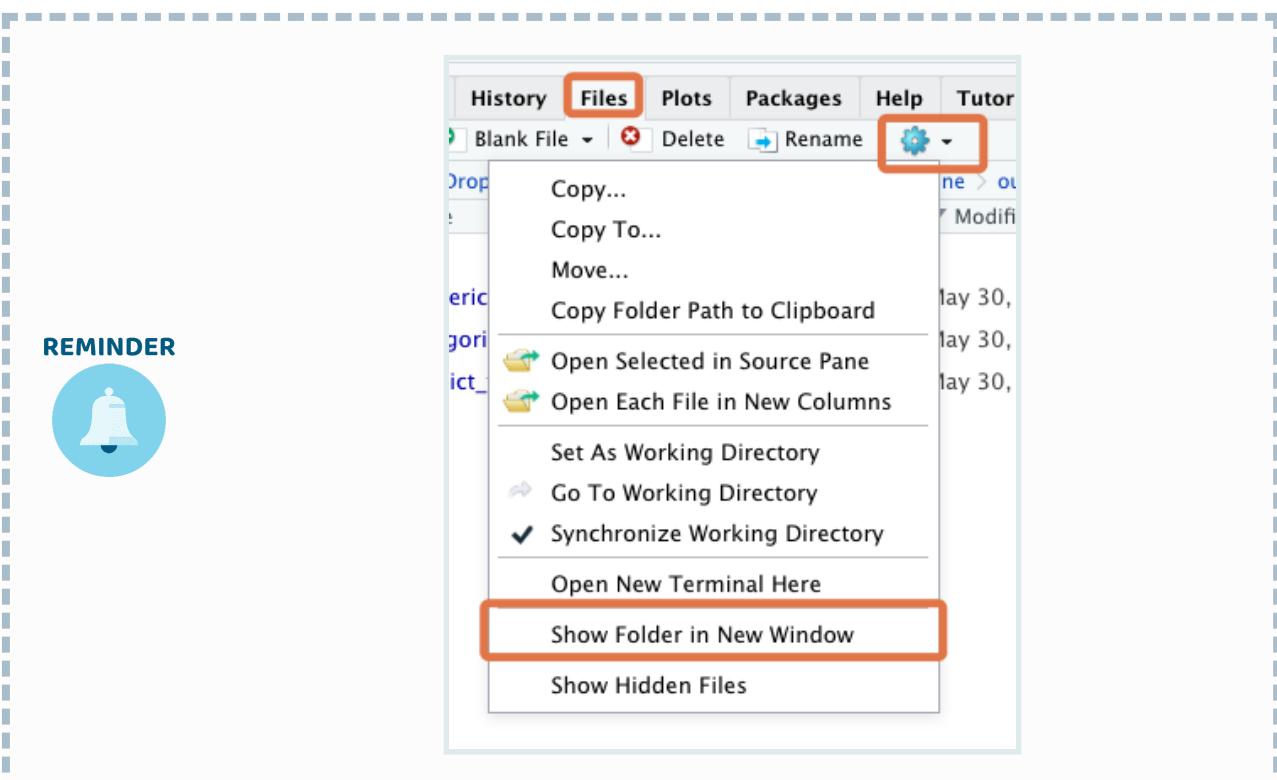
```
ola_analysis.R x district_table.csv x
district_vec,n,percent
Bo,2,0.01
Kailahun,155,0.775
Kambia,1,0.005
Kenema,34,0.17
Kono,2,0.01
Port Loko,2,0.01
Western Urban,4,0.02
```

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

REMINDER



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



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

Overwriting data

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

To test this, replace the "Cases by district" section of your script with the following code. It uses the `arrange()` function to arrange the table in order of the number of cases, `n`:

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

(`-n` means "sort in descending order of the `n` variable"; we will introduce you to the `arrange` function properly later on.)

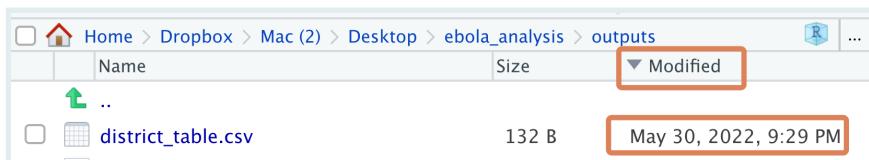
The output should be:

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

You can now overwrite the old “district_table.csv” file by re-running the write_csv function with the `district_tab` object:

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

To verify that the dataset was actually updated, observe the “Modified” time stamp in the RStudio Files pane:



Exporting plots to the “outputs” folder

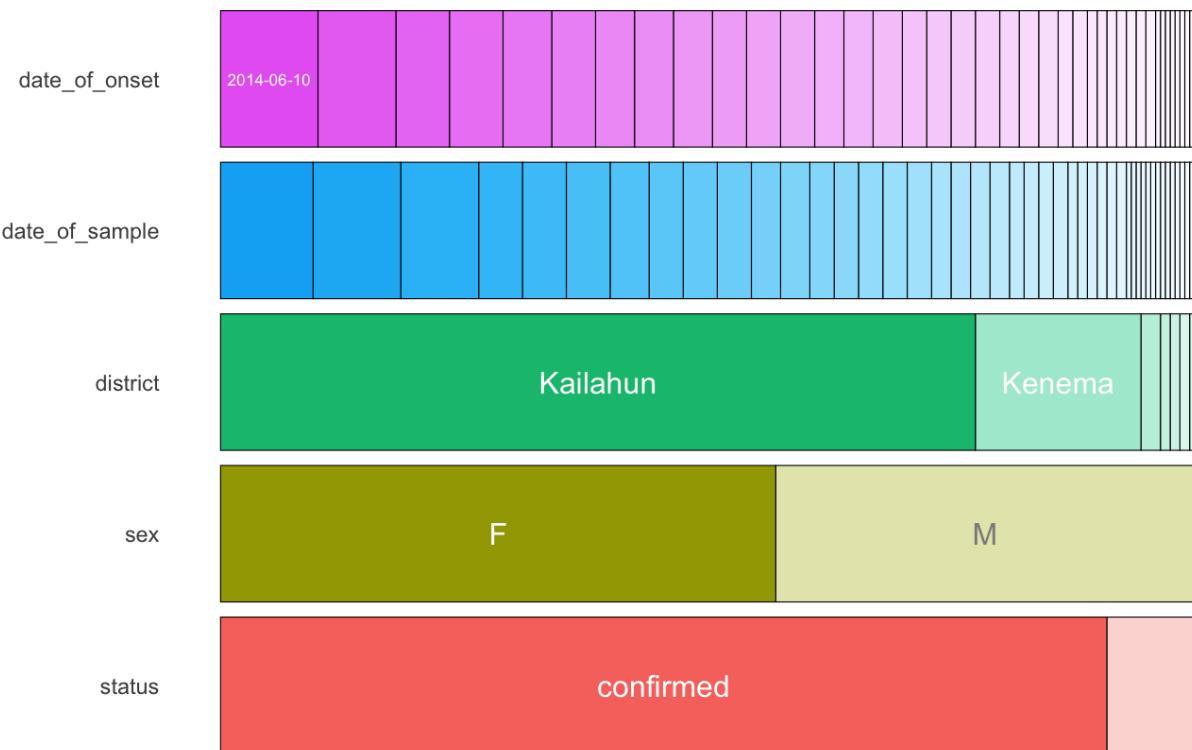
Finally, let's look at plot exporting in the context of an RStudio Project.

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:

Frequency of categorical levels in df::ebola_sierra_leone
Gray segments are missing values



Below these lines, type up the `ggsave()` command below (but don't run it yet):

```
ggsave(filename = "", plot = categ_vars_plot)
```

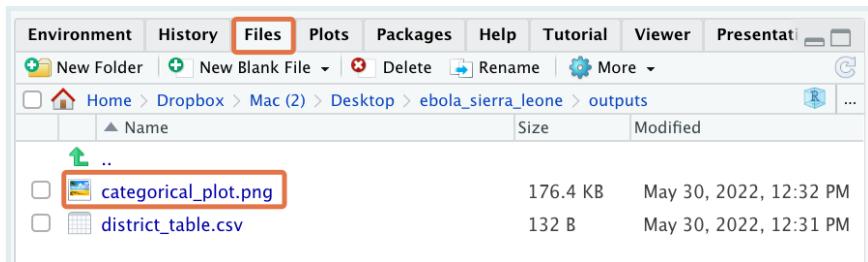
This command uses the `ggsave()` function to export the `categ_vars_plot` figure. The `plot` argument of `ggsave()` takes in the object to be saved (in this case `categ_vars_plot`), and the `filename` argument takes in the target file path for the plot.

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:

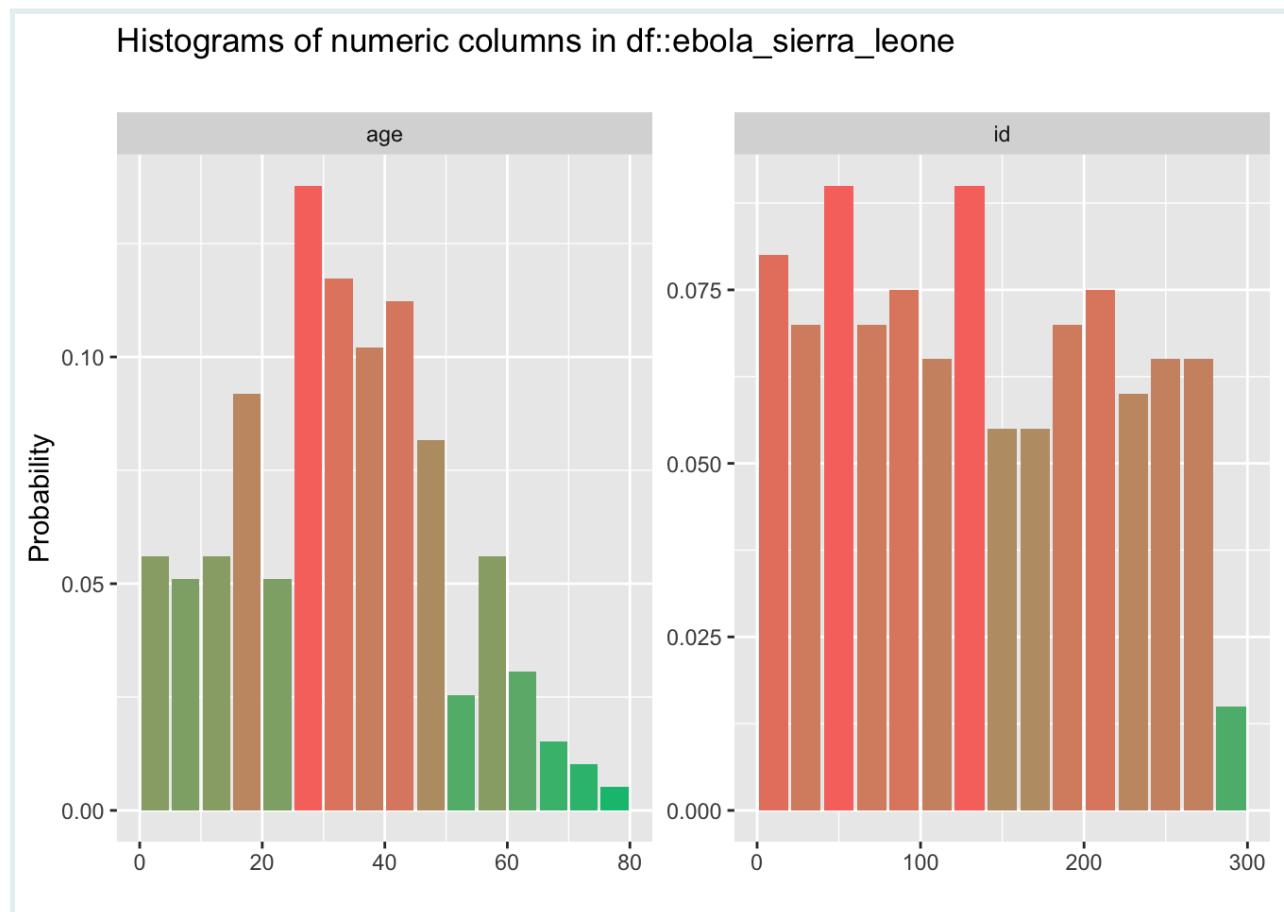
```
ggsave(filename = "outputs/categorical_plot.pdf", plot = categ_vars_plot)
```

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!

Sharing a Project

Projects are also great for sharing your analysis with collaborators.

You can zip up your Project folder and send it to a colleague through email or through a file sharing service like Dropbox. The colleague can then unzip the folder, click on the .Rproj file to open the Project in RStudio, and re-do and edit all your analysis steps.

This is a decent setup, but sending projects back and forth may not be ideal for long-term collaboration. So experienced analysts use a technology called *git* to collaborate on projects. But this topic is a bit too advanced for this course; we will cover it in detail in a future course. If you are impatient, you can check out this book chapter: https://intro2r.com/github_r.html

Wrapping up

Congratulations! You now know how to set up and use RStudio Projects!

Hopefully you see the value of organizing your analysis scripts, data and outputs in this way. Projects are a coherent way to structure your analyses, and make it easy to revisit, revise and share your work. They will be the foundation for much of your work as a data analyst going forward.

That's it for now. See you in the next lesson.

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

- Wickham, H., & Grolemund, G. (n.d.). *R for data science*. 8 Workflow: projects | R for Data Science. Retrieved May 31, 2022, from <https://r4ds.had.co.nz/workflow-projects.html>

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Lesson notes | Data classes and structures

Created by the GRAPH Courses team

January 2023

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

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

Intro
Learning objectives
Packages
What is a data class?
Numeric
Character
Quotation marks
Logical
Logical values and relational operators
Date
Converting to the YYYY-MM-DD format with {lubridate}
Introducing vectors
Creating vectors
Manipulating vectors
A common mistake: missing vector notation
Shorthand functions for numerical vectors
From vectors to data frames
Tibbles
read_csv() creates tibbles
Factors
Wrap-up

Intro

So far, we have focused on some of the important tools for working with data in R: the RStudio IDE, Rstudio projects and R Markdown. In this lesson, we will start to take a closer look at the ways that R stores data.

This lesson introduces many new concepts. Make sure you type along with the tutorial, so that you can develop a strong recall of these. Open a script in RStudio and type each code section out yourself.

Learning objectives

1. You can identify and create objects of the following classes: numeric, character, logical, date, and factor.
2. You can use the `class()` and `is.xxxx()` family of functions (e.g. `is.numeric()`) to check data classes.
3. You can use the `as.xxxx()` family of functions (e.g. `as.character()`) to convert between data classes.

4. You know what relational operators (e.g. ==) are and can combine relational conditions with & and |
5. You can use the ymd() -like functions from {lubridate} to parse date data.
6. You can create vectors with the c() function.
7. You can combine vectors into data frames.
8. You understand the difference between a tibble and a data frame.

Packages

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

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

What is a data class?

A data class is a way of categorizing data based on the type of values that it can take. In R, there are five main data classes that you need to be aware of: numeric, character, logical, date and factor. Each is described in more detail below.

Numeric

Let's start with the numeric class, which is used for data that contains, well, numbers. This could be an integer or a decimal, like 25 or 23.4.

You can verify the class of numbers (or any other data type) with the built-in `class()` function:

```
class(4)
```

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

```
class(0.1)
```

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

The function `is.numeric()` is also used to verify that an object is numeric:

```
is.numeric(4)
```

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

```
is.numeric("Bob") # Not numeric
```

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

(Apart from `is.numeric()`, there is a whole family of other `is.XXX` functions, such as `is.character()`. You will see these below.)

Numeric data can sometimes be represented with scientific notation, where “e” refers to “10 to the power of”. For example, we can write the number 2000 as 2 times 10 to the power of 3, `2e3`:

```
2e3
```

```
## [1] 2000
```

“Integers” (numbers without any decimal) are a special class of numbers, represented with an “L” after the number:

```
4L
```

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

```
class(4L)
```

```
## [1] "integer"
```

However, you usually should not have to use the `L` notation; to write a whole number like 4, just write `4`, not `4L`.

PRO TIP



You may also see the terms “real” or “double” (abbreviated as “dbl”) used to describe numeric data. The differences between these are only relevant for advanced users. You can ignore them for now.

Each line of code below tries to define a numeric object (the numbers 1 to 4). But all lines have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment:

```
numeric_obj1 <- `1L` # define the number 1  
numeric_obj2 <- two # define the number 2  
numeric_obj3 <- "3" # define the number 3  
numeric_obj4 <- 4,0 # define the number 4.0  
numeric_obj5 <- 5E0 # define the number 5
```

Character

The character class is used for data that contains text. In R, we write text by putting it in quotation marks, like this:

```
"A piece of text."
```

We can check this object’s class like so:

```
class("A piece of text.")
```

```
## [1] "character"
```

```
is.character("A piece of text.")
```

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

Note that if you wrap a number in quotes, it automatically becomes a character, according to R:

```
class(4) # numeric
```

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

```
class("4") # Now a character
```

```
## [1] "character"
```

VOCAB



Character values are sometimes referred to as “strings” or “character strings”. You can use these terms interchangeably.

Quotation marks

You can use either single, ' , or double quotation marks, " , to create a character string. The [tidyverse style guide](#), which we use, recommends that you use double quotes in most cases.

Notably, if you start a string with a single quote, you must also close it with a single quote; the same goes for double quotes. So a string like "Hello World' is not properly defined and will cause an error, because it opens with double quotes, but closes with a single quote.

If quotation marks are used to define character strings, what should you do when your string already has a quotation mark within it?

For example, how would you create the following string Obi said "Hello World".

Here, you have two options: you can double quotes internally, with single quotes to wrap the whole string:

```
obi_string <- 'Obi says "Hello World"'  
obi_string
```

```
## [1] "Obi says \"Hello World\""
```

or you can use single quotes internally, with double quotes to wrap the whole string:

```
obi_string_2 <- "Obi says 'Hello World'"  
obi_string_2
```

```
## [1] "Obi says 'Hello World'"
```

If you try to use double quotes both for the internal quote, and externally to wrap the string, you will get an error:

```
obi_string_3 <- "Obi says \"Hello World\""
```

```
Error: object 'obi_string_3' not found
```

The same thing will happen if you use single quotes internally and externally.

Each line of code below tries to define a character object. But all lines EXCEPT ONE have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment:

```
char_object1 <- Hello World  
char_object2 <- 'My name is Ike'  
char_object3 <- 'I am 24' years old  
char_object4 <- 'I'm learning R"  
char_object5 <- `for data science`
```

Note that you cannot perform math operations on character values. For example, the code below gives an error:

```
sqrt("100") # square root. Does not work
```

WATCH OUT



Error in sqrt("100") : non-numeric argument to mathematical function

But we can convert that character into a numeric class with the function `as.numeric()`, and then the function will run:

```
sqrt(as.numeric("100"))
```

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

The `as.numeric()` function seen above is part of a family of functions for converting between classes. There are many others. We could for example, convert a number into a character:

4

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

```
as.character(4)
```

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

Above, you can observe that the `as.character(4)` line prints its output with quotes, indicating that it is a character.

Logical

Logical data contains two values, `TRUE` or `FALSE`, which are written in all capital letters. These can also be written in short as `T` and `F`. Logical data can be thought of like a light switch: it's either on (`TRUE`) or off (`FALSE`).

Logical values are often used as the arguments to functions, for example:

```
mean(airquality$Ozone)
```

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

```
mean(airquality$Ozone, na.rm = TRUE) # remove NAs to calculate mean
```

```
## [1] 42.12931
```

The second code line, with `na.rm = TRUE` is an example of the use of the logical `TRUE` value as the argument to a function.

Each line of code below tries to define a logical object. But all lines, EXCEPT ONE, have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment.

```
logical_obj1 <- f
logical_obj2 <- False
logical_obj3 <- "True"
logical_obj4 <- F
logical_obj5 <- `TRUE`
```

Logical values and relational operators

Logical values are returned when you apply *relational operators* in R.

A relational operator (sometimes called comparison operators) tests the relationship between two values. You will consider them in detail in a future lesson, but here we'll give a few examples:

The greater-than, `>`, operator checks whether the left-hand-side (LHS) object is greater than the right-hand-side (RHS) object:

```
3 > 4 # is 3 greater than 4? Answer is FALSE
```

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

The `==` comparator checks whether two values are equal:

```
3 == 3 # is 3 equal to 3? Answer is TRUE. Note the double equals sign here.
```

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

The `<=` operator checks whether the LHS is less than or equal to the RHS object:

```
3 <= 3 # is 3 less than or equal to 3? Answer is TRUE
```

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

Logical values can be combined using the ampersand, “`&`”, which means “AND”, or the vertical bar, “`|`”, which means “OR”.

`&` checks whether ALL values are true:

```
TRUE & TRUE # All values are true. Returns TRUE
```

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

```
TRUE & FALSE # ALL values are not true. Returns FALSE
```

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

`|` checks whether AT LEAST ONE value is true:

```
TRUE | FALSE # At least one value is true. Returns TRUE
```

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

```
FALSE | FALSE # No value is true. Returns FALSE
```

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

It will become clearer how & and | work when you use these in the context of real datasets. So if it feels unclear now, feel free to ignore it.

Predict whether each line below will evaluate to TRUE or FALSE. Then use your R code console to check these.

```
is.numeric(5)
5 == 5
5 > 6
is.numeric(5) | 5 > 6
is.numeric(5) & 5 > 6
```

Date

The `Date` class contains dates, which must be formatted in YYYY-MM-DD format (e.g. "2022-12-31"), with four digits for the year, two digits for the month, and two digits for the day of the month.

Of course if you just put in such a date string, R will initially consider this to be a character:

```
class("2022-12-31")
```

```
## [1] "character"
```

In order for R to recognize a data value as a date, you use the `as.Date()` function:

```
my_date <- as.Date("2022-12-31")
class(my_date)
```

```
## [1] "Date"
```



WATCH OUT
Note the capital "D" in the `as.Date()` function!

With the date format, you can now do things like find the difference between two dates:

```
as.Date("2022-12-31") - as.Date("2022-12-20")
```

```
## Time difference of 11 days
```

This would of course not be possible if you had bare characters:

```
"2022-12-31" - "2022-12-20"
```

```
Error in "2022-12-31" - "2022-12-20" :  
non-numeric argument to binary operator
```

Note that if you use any other date format than YYYY-MM-DD, R's `as.Date()` function will not work:

```
as.Date("12/31/2022") # Common America date format MM/DD/YYYY  
as.Date("Dec 31, 2022") # Common America date format MM/DD/YYYY
```

```
Error in charToDate(x) :  
character string is not in a standard unambiguous format
```

Each line of code below tries to define a date object. But all lines, EXCEPT ONE, have a mistake, and do not properly define the object. Try to find the four mistakes, fix them and perform the assignment.

```
date_obj1 <- as.date("2022-12-31")  
date_obj2 <- as.date(2022-12-30)  
date_obj3 <- as.Date("2022-12-29")  
date_obj4 <- as.Date("12/28/2022")  
date_obj5 <- "2022-12-27"
```

Converting to the YYYY-MM-DD format with {lubridate}

Because R only recognizes the “YYYY-MM-DD” format as a date, you will often have to convert from other date formats into this format. The `{lubridate}` package makes this very easy. It has a family of intelligent date-parsing functions, which are named in terms of the relative arrangement of year, month and date.

So you have the `mdy()` function (which stands for month, day, year), `dmy()` (day, month, year), `ymd()` (year, month, day), and so on.

Run the lines of code below to observe how these `lubridate` date parsing functions work.

```
mdy("December 31 2001")
```

```
## [1] "2001-12-31"
```

```
mdy("Dec 31 2001")
```

```
## [1] "2001-12-31"
```

```
mdy("Dec-31-01")
```

```
## [1] "2001-12-31"
```

```
mdy("01/31/2001")
```

```
## [1] "2001-01-31"
```

```
ymd("2001 December 31st")
```

```
## [1] "2001-12-31"
```

```
ymd("2001-Dec-31")
```

```
## [1] "2001-12-31"
```

Do you see the beauty of this? You do not need to worry about whether a hyphen or a slash or a space was used to separate the dates, or whether the months were spelled out in full or abbreviated. All you need to know is the intended order of the date components (day, month and year) and voila!

Note that the lubridate functions automatically do the `as.Date()` conversion to convert the values from characters to dates:

```
class("December 31 2001") # recognized only as a character
```

```
## [1] "character"
```

```
class(mdy("December 31 2001")) # after applying lubridate, R recognizes value  
as date
```

```
## [1] "Date"
```

Convert the following to R dates with the `ymd` family of functions from `lubridate`:

```
"March 29, 2022"  
"2022 March 29"  
"2022 Mar 29"  
"2022/03/29"
```

R stores dates internally as the number of days since January 1, 1970. This means that the date January 1, 1970 is represented as 0, while January 2, 1970 is represented as 1, and so on. You can see this by running converting those dates to numbers:

```
as.numeric(as.Date("1970-01-01"))
```

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

PRO TIP



```
as.numeric(as.Date("1970-01-02"))
```

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

```
as.numeric(as.Date("2022-12-31"))
```

```
## [1] 19357
```

This information is sometimes useful when importing date data.

Introducing vectors

So far, we have been looking at data that contains just a single value. But of course, most data comes in some kind of collection—a data *structure*. The data structure you are most familiar with is a data table (sometimes called a spreadsheet, but typically represented as a data frame in R).

But the most fundamental data structures in R are actually *vectors*. Let's spend some time thinking about these.

A vector is a collection of values that all have the same class (for example, all numeric or all character). It may be helpful to think of a vector as a column or row in an Excel spreadsheet.

Creating vectors

Vectors can be created using the `c()` function, with the components of the vector separated by commas. For example, the code `c(1, 2, 3)` defines a vector with the elements 1, 2 and 3.

In your script, define the following vectors:

```
my_numeric_vec <- c(0, 1, 1, 2, 3)
my_numeric_vec # print the vector
```

```
## [1] 0 1 1 2 3
```

```
my_numeric_vec2 <- c(4, 5, 3, 4, 1)
my_numeric_vec2 # print the vector
```

```
## [1] 4 5 3 4 1
```

```
my_character_vec <- c("Bob", "Jane", "Joe", "Obi", "Aka")
my_character_vec # print the vector
```

```
## [1] "Bob"  "Jane" "Joe"  "Obi"  "Aka"
```

```
my_logical_vec <- c(T, T, F, F, F)
my_logical_vec # print the vector
```

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

You can also check the classes of these vectors:

```
class(my_numeric_vec)
```

```
## [1] "numeric"

class(my_character_vec)
```

```
## [1] "character"
```

```
class(my_logical_vec)
```

```
## [1] "logical"
```

Each line of code below tries to define a vector with three elements. But all lines, EXCEPT ONE, have a mistake, and do not properly define the vector object. Try to find the four mistakes, fix them and perform the assignment.

```
my_vec_1 <- (1,2,3)
my_vec_2 <- 2,3,4
my_vec_3 <- "c(4,5,6)"
my_vec_4 <- c("Obi", "Chika" "Nonso")
my_vec_5 <- as.Date(c("2020-10-10", "2020-10-11", "2020-10-13"))
```



VOCAB

The individual values within a vector are called *components* or elements.
So the vector `c(1, 2, 3)` has three components/elements.

Manipulating vectors

Many of the functions and operations you have encountered so far in the course can be applied to vectors.

For example, we can multiply our `my_numeric_vec` object by 2:

```
my_numeric_vec
```

```
## [1] 0 1 1 2 3
```

```
my_numeric_vec * 2
```

```
## [1] 0 2 2 4 6
```

Notice that every element in the vector was multiplied by 2.

Or, below we take the square root of `my_numeric_vec2`:

```
my_numeric_vec2
```

```
## [1] 4 5 3 4 1
```

```
sqrt(my_numeric_vec2)
```

```
## [1] 2.000000 2.236068 1.732051 2.000000 1.000000
```

You can also add (numeric) vectors to each other:

```
my_numeric_vec + my_numeric_vec2
```

```
## [1] 4 6 4 6 4
```

Note that the first element of `my_numeric_vec` is added to the first element of `my_numeric_vec2` and the second element of `my_numeric_vec` is added to the second element of `my_numeric_vec2` and so on.

Below are some other functions you could run on vector. Type them out in your console and observe their outputs:

```
head(my_character_vec, 2) # first two elements
table(my_logical_vec)
length(my_logical_vec)
sort(my_numeric_vec2)
sort(my_character_vec) # sorts in alphabetical order
```

Consider the vector defined here, which holds the hand circumference, in inches, of children:

```
circum_inches <- c(4.0, 5.6, 3.4, 5.8, 5.6, 4.0)
```

Each line of code below tries to run a function on this vector. But all lines, EXCEPT ONE, have a mistake, and do not properly carry out the function. Try to find the four mistakes and fix them.

```
sum()circum_inches # find the sum of the vector
head(3, circum_inches) # take the first three elements of the vector
sort(circum_inches, decreasing = false) # sort the vector in increasing order
Table(circum_inches) # frequency table
janitor::tabyl(circum_inches) # frequency table
```

```
## Error: <text>:1:6: unexpected symbol
## 1: sum()circum_inches
##           ^
##
```

A common mistake: missing vector notation

An error that students frequently encounter involves failing to create a vector where one is needed. For example, consider the code line below:

```
mean(1, 2, 3, 4)
```

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

Hmmm. The mean of 1, 2, 3 and 4 is definitely not 1. What is going on. This is happening because the primary argument to `mean()` must be a vector:

```
mean(c(1, 2, 3, 4))
```

```
## [1] 2.5
```

Now all good!

Watch out for this/

Using the `median()` function, find the median of this collection of numbers: 1, 5, 2, 8, 9, 10, 11, 46, 23, 45, 2, 4, 5, 6. (Remember to put the number collection in a vector!)

Shorthand functions for numerical vectors

R has a number of shorthand functions for creating numerical vectors. The most commonly used of these is the colon operator, `:`, which creates a sequence of integers:

```
1:5 # 1 to 5
```

```
## [1] 1 2 3 4 5
```

```
100:113 # 100 to 113
```

```
## [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113
```

You can also use the `seq()` function to create a sequence of numbers as a vector:

For example, to create a sequence of numbers from 1 to 10, you can run:

```
seq(from = 1, to = 10)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

By default, the increment is set to 1. To use a different increment, just change the value of the `by` argument. For example, to create a sequence with an increment of two:

```
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

It's also possible to create descending sequences by using `:` or `seq()`:

```
10:1
```

```
## [1] 10 9 8 7 6 5 4 3 2 1
```

```
seq(from = 10, to = 1, by = -2)
```

```
## [1] 10 8 6 4 2
```



Although we defined a vector as a *collection* of objects, in R even single values are technically considered vectors! You can check this with the `is.vector()` function:

```
is.vector(c(1,2,3)) # obviously a vector
```

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



```
is.vector(1) # still a vector!
```

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

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.

In the below example, we first create vector variables (`age`, `sex` and `comorbidity`) for six individuals:

```
age <- c(18, 25, 46, 54, 60, 72)
sex <- c('M', 'F', 'F', 'M', 'M', 'F')
comorbidity <- c(T, T, F, F, F, T)
```

We can now use the `data.frame()` function to combine these into a single tabular structure:

```
data.epi <- data.frame(age, sex, comorbidity)
data.epi
```

```
##   age sex comorbidity
## 1 18   M      TRUE
## 2 25   F      TRUE
## 3 46   F     FALSE
## 4 54   M     FALSE
## 5 60   M     FALSE
## 6 72   F      TRUE
```

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 <- data.frame(age = c(18, 25, 46, 54, 60, 72),
                       sex = c('M', 'F', 'F', 'M', 'M', 'F'),
                       comorbidity = c(T, T, F, F, F, T))

data.epi
```

```
##   age sex comorbidity
## 1 18   M      TRUE
```

```
## 2 25 F TRUE
## 3 46 F FALSE
## 4 54 M FALSE
## 5 60 M FALSE
## 6 72 F TRUE
```

We can check the class of this data frame:

```
class(data_epi)
```

```
## [1] "data.frame"
```

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.

WATCH OUT



To create a data frame from vectors, all the vectors **must** have the same length. Otherwise you will get an error. For example:

```
mydf <- data.frame(age = c(5, 9, 8),
                     sex = c('M', 'F'))
```

```
Error in data.frame(age = c(5, 9, 8), sex = c("M", "F")) :
  arguments imply differing number of rows: 3, 2
```

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
```

Earlier, we defined a data frame with the following vectors below. Combine these into a data frame, with the following column names: “number_of_children” for the numeric vector, “name” for the character vector and “is_married” for the logical vector.

```
my_numeric_vec <- c(0, 1, 1, 2, 3)
my_character_vec <- c("Bob", "Jane", "Joe", "Obi", "Aka")
my_logical_vec <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
```

Use the `data.frame()` function to define a data frame in R that resembles the following table:

room	number_of_windows
dining	3
kitchen	2
bedroom	5

Tibbles

The default version of tabular data in R is called a data frame, but there is another representation of tabular data provided by the `tidyverse` package. It's called a `tibble`, and it is an improved version of `data.frame`.

You can create a tibble using the `tibble()` function. (Remember to import the `tidyverse` package to use its functions.)

```
tibble.epi <- tibble(  
  age = c(18, 25, 46, 54, 60, 72),  
  sex = c('M', 'F', 'F', 'M', 'M', 'F'),  
  comorbidity = c(T, T, F, F, F, T)  
)  
tibble.epi
```

```
## # A tibble: 6 × 3  
##   age   sex   comorbidity  
##   <dbl> <chr> <lgl>  
## 1    18 M     TRUE  
## 2    25 F     TRUE  
## 3    46 F     FALSE  
## 4    54 M     FALSE  
## 5    60 M     FALSE  
## 6    72 F     TRUE
```

Notice that the tibble gives the data dimensions in the first line:

```
👉 # A tibble: 6 × 3👉  
  age   sex   comorbidity  
  <dbl> <chr> <lgl>  
1    18 M     TRUE  
2    25 F     TRUE
```

And also tells you the data types, at the top of each column:

```
# A tibble: 6 × 3
  age sex   comorbidity
  <dbl> <chr> <lgl>
1     18 M    TRUE
2     25 F    TRUE
```

There, “dbl” stands for double (which is a kind of numeric class), “chr” stands for character, and “lgl” for logical.

You can convert a `data.frame` to `tibble` using the `as_tibble` function:

```
df <- data.frame(
  age = c(18, 25, 46, 54, 60, 72),
  sex = c('M', 'F', 'F', 'M', 'M', 'F'),
  comorbidity = c(T, T, F, F, F, T)
)

a_tibble <- as_tibble(df)

a_tibble
```

```
## # A tibble: 6 × 3
##   age sex   comorbidity
##   <dbl> <chr> <lgl>
## 1     18 M    TRUE
## 2     25 F    TRUE
## 3     46 F    FALSE
## 4     54 M    FALSE
## 5     60 M    FALSE
## 6     72 F    TRUE
```

And you can convert a `tibble` back to a `data frame` with `as.data.frame()`:

```
as.data.frame(a_tibble)
```

```
##   age sex comorbidity
## 1 18   M   TRUE
## 2 25   F   TRUE
## 3 46   F   FALSE
## 4 54   M   FALSE
## 5 60   M   FALSE
## 6 72   F   TRUE
```

For your most of your data analysis needs, you should prefer `tibbles` over regular `data frames`.

`read_csv()` creates tibbles

When you import data with the `read_csv()` function from `{readr}`, you get a tibble:

```
ebola_tib <- read_csv("https://tinyurl.com/ebola-data-sample") # Needs  
            internet to run  
class(ebola_tib)
```

```
## [1] "spec_tbl_df" "tbl_df"        "tbl"          "data.frame"
```

But when you import data with the base `read.csv()` function, you get a `data.frame`:

```
ebola_df <- read.csv("https://tinyurl.com/ebola-data-sample") # Needs internet  
            to run  
class(ebola_df)
```

```
## [1] "data.frame"
```

Try printing `ebola_tib` and `ebola_df` to your console to observe the different printing behavior of tibbles and data frames.

The `iris` data frame is one of R's built-in datasets. Convert it to a tibble with the `as_tibble()` function. Then print it to your console. How does the tibble output differ from the original `iris` data frame?

Factors

Finally, let's turn briefly to factors, which is another data class. We left this class until the end because understanding factors requires that you understand vectors.

A factor is a nominal (categorical) variable with a set of known possible values called levels.

Why might we use a factor class? The most common reason are:

- to force characters to sort in a custom order
- to show zero counts

What these mean will become clear by considering an example.

Imagine that you have a variable that records the month of birth for a number of infants:

```
birth_month <- c("Dec", "Apr", "Jan", "Mar", "Oct", "Nov", "Jan", "Apr")
```

And you want to count the number of births per month. You could use the base `table()` function:

```
table(birth_month)
```

```
## birth_month
## Apr Dec Jan Mar Nov Oct
## 2   1   2   1   1   1
```

We see that 2 babies were born in April, 1 in December, and so on.

You could also use the `tabyl()` function from `{janitor}` for this:

```
tabyl(birth_month)
```

```
##   birth_month n percent
##       Apr 2    0.250
##       Dec 1    0.125
##       Jan 2    0.250
##       Mar 1    0.125
##       Nov 1    0.125
##       Oct 1    0.125
```

But do you see a problem with these outputs? The months are sorted in alphabetical order(!) Indeed if you try to `sort()` the `birth_month` vector directly, you will note the same thing:

```
sort(birth_month)
```

```
## [1] "Apr" "Apr" "Dec" "Jan" "Jan" "Mar" "Nov" "Oct"
```

But this is not a sensible way to sort this variable. A chronological order, with January first, would be much better.

You can fix this problem with a factor. To create a factor you use the `factor()` function, with your original character vector and a list of valid **levels**, arranged in the correct order:

```
birth_month_factor <- factor(x = birth_month,
                               levels = c("Jan", "Feb", "Mar", "Apr",
                                         "May", "Jun", "Jul", "Aug",
                                         "Sep", "Oct", "Nov", "Dec"))
```

```
class(birth_month_factor) # check its class
```

```
## [1] "factor"
```

```
birth_month_factor # print it
```

```
## [1] Dec Apr Jan Mar Oct Nov Jan Apr  
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Notice that the levels are listed in the output.

```
[1] Dec Apr Jan Mar Oct Nov Jan Apr  
👉 Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec👉
```

Now, we can sort the vector properly:

```
sort(birth_month_factor)
```

```
## [1] Jan Jan Mar Apr Apr Oct Nov Dec  
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

And if we create the frequency count tables, we get the right order:

```
table(birth_month_factor)
```

```
## birth_month_factor  
## Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec  
##   2   0   1   2   0   0   0   0   0   1   1   1
```

```
tabyl(birth_month_factor)
```

```
##   birth_month_factor n percent  
##                 Jan 2    0.250  
##                 Feb 0    0.000  
##                 Mar 1    0.125  
##                 Apr 2    0.250  
##                 May 0    0.000  
##                 Jun 0    0.000  
##                 Jul 0    0.000  
##                 Aug 0    0.000  
##                 Sep 0    0.000  
##                 Oct 1    0.125  
##                 Nov 1    0.125  
##                 Dec 1    0.125
```

As you can see, months with zero counts are also included in the table outputs. This will often be useful!

If you would rather not see these zero count months, the `tabyl()` function allows you to drop them, by using the `show_missing_levels` argument:

```
tabyl(birth_month_factor, show_missing_levels = FALSE)
```

```
## birth_month_factor n percent
##                 Jan 2    0.250
##                 Mar 1    0.125
##                 Apr 2    0.250
##                 Oct 1    0.125
##                 Nov 1    0.125
##                 Dec 1    0.125
```

The variable `visit_day` below records the day of the week that a clinic was visited.

```
visit_day <- c("Mon", "Mon", "Tue", "Fri", "Thu", "Wed", "Sun", "Sat", "Tue")
```

Convert this into a factor with the `factor()` function. The levels should be in order of the days of the week, starting with “Sun” and ending with “Sat”.

Then create a frequency table of this variable using the `tabyl()` function. Does the table sort in the proper chronological order?

Wrap-up

You've learned a lot in this lesson! You now know about all of the basic R data classes (numeric, character, logical, date, factor) and how to create objects of each class. You also know how to check an object's class with `class()` and convert between classes with `as.xxx()`. Finally, you know how to create vectors and data frames.

With this knowledge, you are now ready to start doing some serious data analysis in R. In the coming lessons, you'll start to learn about the `dplyr` package, which will provide you with powerful tools for manipulating your data frames and tibbles.

Congratulations on making it this far! You have covered a lot and should be proud of yourself.

Contributors

The following team members contributed to this lesson:



DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health researcher at Fiocruz, Brazil

Passionate about lots of things, especially when it involves people leading lives with more equality and freedom



EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

- Wickham, H., & Grolemund, G. (n.d.). *R for data science*. 15 Factors | R for Data Science. Accessed October 26, 2022. <https://r4ds.had.co.nz/factors.html>.

This work is licensed under the Creative Commons Attribution Share Alike license.



Lesson notes | Selecting and renaming columns

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Learning objectives
The Yaounde COVID-19 dataset
Introducing <code>select()</code>
Selecting column ranges with :
Excluding columns with !
Helper functions for <code>select()</code>
<code>starts_with()</code> and <code>ends_with()</code>
<code>contains()</code>
<code>everything()</code>
Change column names with <code>rename()</code>
Rename within <code>select()</code>
Wrap Up !

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.



Fig: the `select()` function.

Let's go !

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.

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 × 53
##   id                  date_surveyed    age age_category
##   <chr>                <date>        <dbl> <chr>
## 1 BRIQUETERIE_000_0001 2020-10-22     45 45 - 64
## 2 BRIQUETERIE_000_0002 2020-10-24     55 45 - 64
## 3 BRIQUETERIE_000_0003 2020-10-24     23 15 - 29
## 4 BRIQUETERIE_002_0001 2020-10-22     20 15 - 29
## 5 BRIQUETERIE_002_0002 2020-10-22     55 45 - 64
## # ... with 49 more variables: age_category_3 <chr>,
## #   sex <chr>, highest_education <chr>, occupation <chr>, ...
```



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

Introducing `select()`

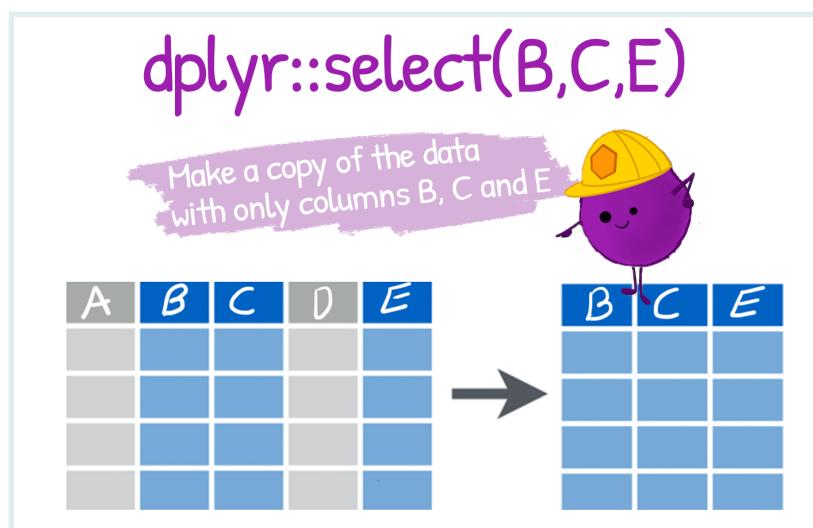


Fig: the `select()` function. (Drawing adapted from Allison Horst).

`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 × 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 × 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 × 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
## # ... with 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 × 8  
##   age   sex   highest_education occupation      is_smoker  
##   <dbl> <chr> <chr>           <chr>          <chr>  
## 1    45 Female Secondary       Informal worker Non-smoker  
## 2    55 Male   University     Salaried worker Ex-smoker  
## 3    23 Male   University     Student        Smoker  
## 4    20 Female Secondary       Student        Non-smoker  
## 5    55 Female Primary        Trader--Farmer Non-smoker  
## # ... with 3 more variables: is_pregnant <chr>,<br/>  
## #   igg_result <chr>, igm_result <chr>
```

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 × 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 × 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

```



- 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.)

Excluding columns with !

The **exclamation point** negates a selection:

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

```

## # A tibble: 5 × 7
##   sex   highest_education occupation     is_smoker
##   <chr> <chr>           <chr>          <chr>
## 1 Female Secondary       Informal worker Non-smoker
## 2 Male   University      Salaried worker Ex-smoker
## 3 Male   University      Student        Smoker
## 4 Female Secondary       Student        Non-smoker
## 5 Female Primary        Trader--Farmer Non-smoker
## # ... with 3 more variables: is_pregnant <chr>,
## #   igg_result <chr>, igm_result <chr>

```

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 × 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 × 5
##   highest_education occupation     is_smoker  is_pregnant
##   <chr>           <chr>          <chr>       <chr>
## 1 Secondary       Informal worker Non-smoker Non-pregnant
## 2 University      Salaried worker Ex-smoker  Non-pregnant
## 3 University      Student        Smoker     Non-pregnant
## 4 Secondary       Student        Non-smoker Non-pregnant
## 5 Primary         Trader--Farmer Non-smoker Non-pregnant

```

```

## <chr>          <chr>          <chr>          <chr>
## 1 Secondary    Informal worker Non-smoker No
## 2 University   Salaried worker Ex-smoker <NA>
## 3 University   Student        Smoker       <NA>
## 4 Secondary    Student        Non-smoker No
## 5 Primary      Trader--Farmer Non-smoker No
## # ... with 1 more variable: igm_result <chr>

```



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

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.

`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 × 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 × 2
##   igg_result igm_result
##   <chr>      <chr>
## 1 Negative   Negative
## 2 Positive   Negative
## 3 Negative   Negative
## 4 Positive   Negative
## 5 Positive   Negative

```

```
contains()
```

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

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

```
## # A tibble: 5 × 12
##   drugsource      is_drug_parac is_drug_antibio
##   <chr>            <dbl>          <dbl>
## 1 Self or familial     1              0
## 2 <NA>                NA             NA
## 3 <NA>                NA             NA
## 4 Self or familial     0              1
## 5 <NA>                NA             NA
## # ... with 9 more variables: is_drug_hydrocortisone <dbl>,
## #   is_drug_other_anti_inflam <dbl>, ...
```

```
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 × 8
##   is_pregnant    age sex   highest_education occupation
##   <chr>        <dbl> <chr> <chr>           <chr>
## 1 No            45   Female Secondary       Informal worker
## 2 <NA>          55   Male   University     Salaried worker
## 3 <NA>          23   Male   University     Student
## 4 No            20   Female Secondary       Student
## 5 No            55   Female Primary        Trader--Farmer
## # ... with 3 more variables: is_smoker <chr>,
## #   igg_result <chr>, 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 × 8
##   is_pregnant    age   sex highest_education occupation
##   <chr>        <dbl> <chr>      <chr>           <chr>
## 1 No            45  Female Secondary Informal worker
## 2 <NA>          55  Male   University Salaried worker
## 3 <NA>          23  Male   University Student
## 4 No            20  Female Secondary Student
## 5 No            55  Female Primary Trader--Farmer
## # ... with 3 more variables: is_smoker <chr>,
## #   igg_result <chr>, 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 × 53
##   is_pregnant id                  date_surveyed    age
##   <chr>        <chr>                <date>        <dbl>
## 1 No          BRIQUETERIE_000_0001 2020-10-22     45
## 2 <NA>        BRIQUETERIE_000_0002 2020-10-24     55
## 3 <NA>        BRIQUETERIE_000_0003 2020-10-24     23
## 4 No          BRIQUETERIE_002_0001 2020-10-22     20
## 5 No          BRIQUETERIE_002_0002 2020-10-22     55
## # ... with 49 more variables: age_category <chr>,
## #   age_category_3 <chr>, sex <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 × 53
##   igg_result igm_result id                  date_surveyed    age
##   <chr>        <chr>        <chr>                <date>        <dbl>
## 1 Negative   Negative   BRIQUETERIE_000... 2020-10-22     45
## 2 Positive    Negative   BRIQUETERIE_000... 2020-10-24     55
## 3 Negative   Negative   BRIQUETERIE_000... 2020-10-24     23
## 4 Positive    Negative   BRIQUETERIE_002... 2020-10-22     20
```

```
## 5 Positive    Negative    BRIQUETERIE_002... 2020-10-22      55
## # ... with 48 more variables: age_category <chr>,
## #   age_category_3 <chr>, sex <chr>, ...
```



- 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.

Change column names with `rename()`

RENAME COLUMNS

`dplyr::rename(enemies = species)`

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

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

`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 × 53  
##   id                  date_surveyed patient_age age_category  
##   <chr>                <date>           <dbl> <chr>  
## 1 BRIQUETERIE_000_00... 2020-10-22        45  45 - 64  
## 2 BRIQUETERIE_000_00... 2020-10-24        55  45 - 64
```

```
## 3 BRIQUETERIE_000_00... 2020-10-24      23 15 - 29
## 4 BRIQUETERIE_002_00... 2020-10-22      20 15 - 29
## 5 BRIQUETERIE_002_00... 2020-10-22      55 45 - 64
## # ... with 49 more variables: age_category_3 <chr>,
## #   patient_sex <chr>, highest_education <chr>, ...
```

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.

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 × 2
##   patient_age patient_sex
##       <dbl> <chr>
## 1        45 Female
## 2        55 Male
## 3        23 Male
## 4        20 Female
## 5        55 Female
```

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.

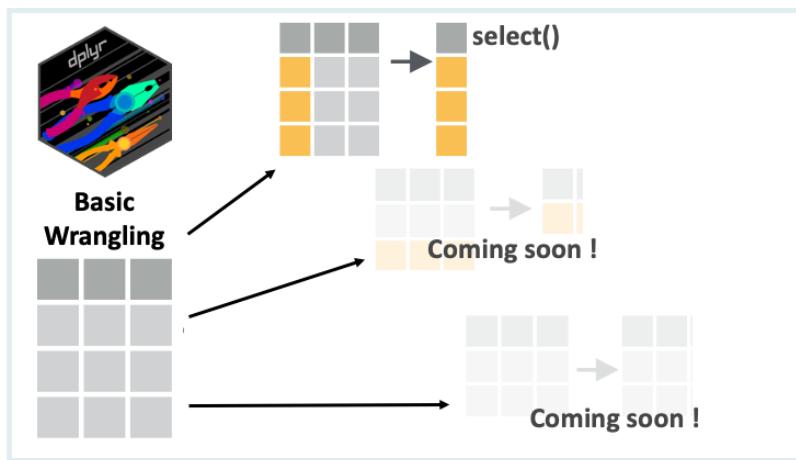


Fig: Basic Data Wrangling Dplyr Verbs.

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education



ANDREE VALLE CAMPOS

R Developer and Instructor, the GRAPH Network
Motivated by reproducible science and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

- 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)

Lesson notes | Filtering rows

Created by the GRAPH Courses team

January 2023

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

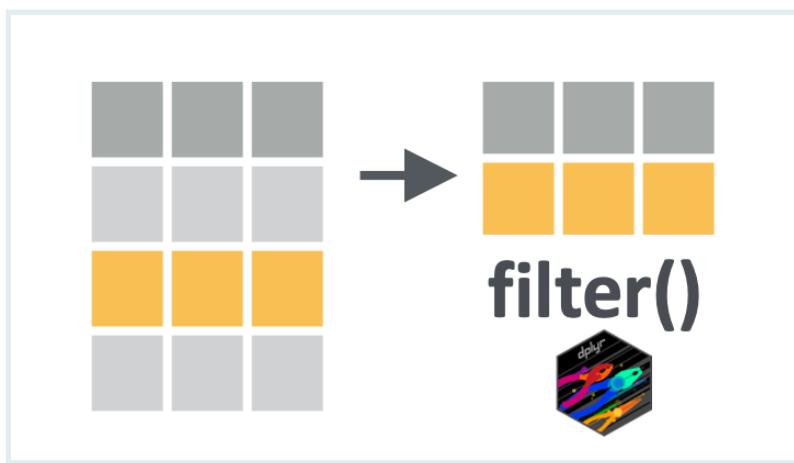
The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Intro
Learning objectives
The Yaounde COVID-19 dataset
Introducing filter()
Relational operators
Combining conditions with & and
Negating conditions with !
NA values
Wrap Up !

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 !



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.

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 × 10  
##   age   sex   weight_kg highest_education neighborhood  
##   <dbl> <chr>     <dbl> <chr>          <chr>  
## 1    45 Female      95 Secondary       Briqueterie  
## 2    55 Male        96 University      Briqueterie  
## 3    23 Male        74 University      Briqueterie  
## 4    20 Female      70 Secondary       Briqueterie  
## 5    55 Female      67 Primary        Briqueterie  
## # ... with 5 more variables: occupation <chr>,  
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

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 × 10  
##   age   sex   weight_kg highest_education neighborhood  
##   <dbl> <chr>     <dbl> <chr>          <chr>  
## 1    55 Male        96 University      Briqueterie  
## 2    23 Male        74 University      Briqueterie  
## 3    28 Male        62 Doctorate      Briqueterie  
## 4    30 Male        73 Secondary       Briqueterie  
## 5    42 Male        71 Secondary       Briqueterie
```

```
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_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



(in RMD)

Filter the `yao` data frame to respondents who were pregnant during the survey.

How many respondents were female? (Use `filter()` and `nrow()`)

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

`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

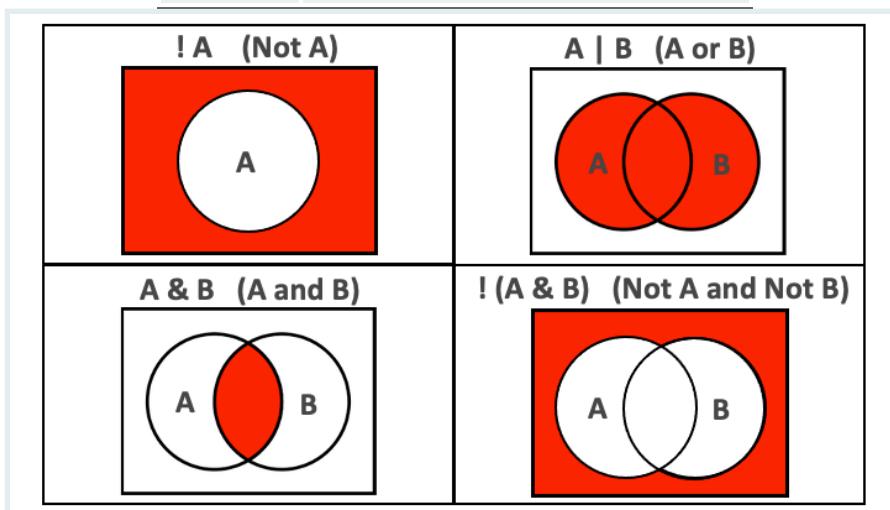


Fig: AND and OR operators visualized.

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

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

```
## # A tibble: 5 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>           <chr>
## 1    45 Female      95 Secondary        Briqueterie
## 2    20 Female      70 Secondary        Briqueterie
## 3    55 Female      67 Primary         Briqueterie
## 4    17 Female      65 Secondary        Briqueterie
## 5    13 Female      65 Secondary        Briqueterie
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

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

```
## # A tibble: 5 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>          <chr>
## 1     5 Female      19 Primary        Carriere
## 2     5 Female      26 Primary        Carriere
## 3     5 Male         16 Primary        Cité Verte
## 4     5 Female      21 Primary        Ekoudou
## 5     5 Male         15 Primary        Ekoudou
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

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

```
## # A tibble: 5 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>          <chr>
## 1    78 Male        95 Secondary     Briqueterie
## 2    79 Female      40 Primary       Briqueterie
## 3    78 Female      60 Primary       Briqueterie
## 4    75 Male         74 Primary       Briqueterie
## 5    72 Male         65 Secondary     Carriere
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

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

```
## # A tibble: 5 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>          <chr>
## 1    45 Female      95 Secondary     Briqueterie
## 2    20 Female      70 Secondary     Briqueterie
## 3    55 Female      67 Primary       Briqueterie
## 4    17 Female      65 Secondary     Briqueterie
## 5    13 Female      65 Secondary     Briqueterie
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_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.

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 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>          <chr>
## 1    25 Female      90 Secondary       Carriere
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_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 × 10
##   age sex    weight_kg highest_education neighborhood
##   <dbl> <chr>     <dbl> <chr>          <chr>
## 1    25 Female      90 Secondary       Carriere
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

Don't confuse:

SIDE NOTE



- 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 × 10
##   age sex  weight_kg highest_education neighborhood
##   <dbl> <chr>    <dbl> <chr>          <chr>
## 1 55  Male      96  University    Briqueterie
## 2 42  Male      71  Secondary     Briqueterie
## 3 38  Male      71  University    Briqueterie
## 4 69  Male      108 University   Briqueterie
## 5 65  Male      93  Secondary     Briqueterie
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_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.

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 × 10
##   age sex  weight_kg highest_education neighborhood
##   <dbl> <chr>    <dbl> <chr>          <chr>
## 1 45  Female    95  Secondary    Briqueterie
## 2 55  Male      96  University   Briqueterie
## 3 23  Male      74  University   Briqueterie
## 4 20  Female    70  Secondary    Briqueterie
## 5 55  Female    67  Primary     Briqueterie
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_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 × 10
##       age sex   weight_kg highest_education neighborhood
##     <dbl> <chr>    <dbl> <chr>           <chr>
## 1     55 Male      96 University Briqueterie
## 2     23 Male      74 University Briqueterie
## 3     28 Male      62 Doctorate Briqueterie
## 4     38 Male      71 University Briqueterie
## 5     54 Male      71 University Briqueterie
## # ... with 5 more variables: occupation <chr>,
## #   is_smoker <chr>, is_pregnant <chr>, igg_result <chr>, ...
```

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”.

KEY POINT



```
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.

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 × 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 × 2
## # ... with 2 variables: sex <chr>, is_pregnant <chr>
```

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

```
## # A tibble: 0 × 2
## # ... with 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 × 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 × 2  
##   sex    is_pregnant  
##   <chr>  <chr>  
## 1 Female No  
## 2 Female No response  
## 3 Female Yes
```

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

SIDE NOTE



```
# 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
```

A common error with `NA`

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



(in RMD)

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

PRACTICE



(in RMD)

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.

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!

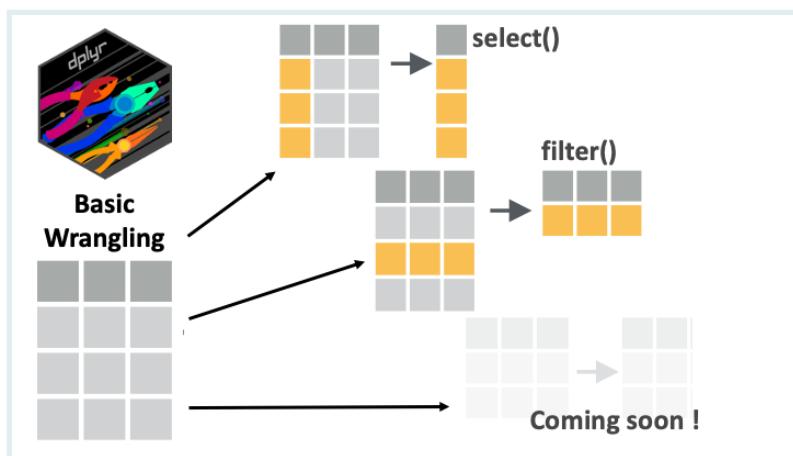


Fig: Basic Data Wrangling: `select()` and `filter()`.

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education



ANDREE VALLE CAMPOS

R Developer and Instructor, the GRAPH Network



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

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 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)

Lesson notes | Mutating columns

Created by the GRAPH Courses team

January 2023

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

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

Intro
Learning objectives
Packages
Datasets
Introducing <code>mutate()</code>
Creating a Boolean variable
Creating a numeric variable based on a formula
Changing a variable's type
Integer: <code>as.integer</code>
Wrap up

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!

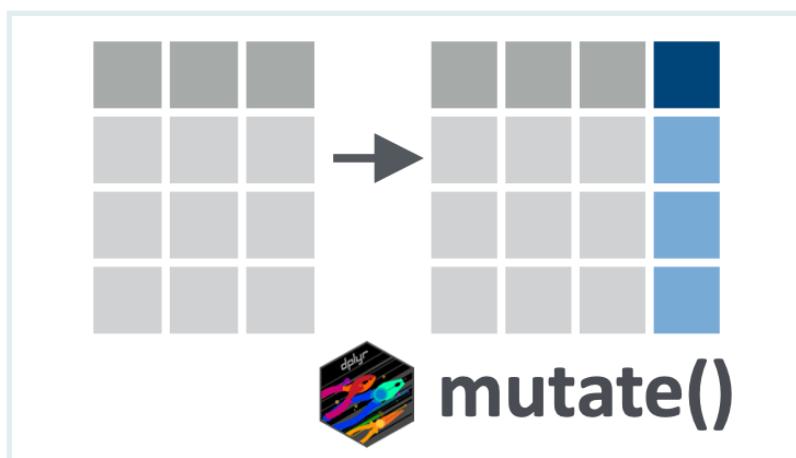


Fig: the `mutate()` verb.

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

Packages

This lesson will require the packages loaded below:

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

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
```

date_surv...	age	weight_kg	height_cm	symptoms	is_smoker
2020-10-22	45	95	169	Muscle pain	Non-smoker
2020-10-24	55	96	185	No sympto...	Ex-smoker
2020-10-24	23	74	180	No sympto...	Smoker
2020-10-22	20	70	164	Rhinitis--Sn...	Non-smoker
2020-10-22	55	67	147	No sympto...	Non-smoker
2020-10-25	17	65	162	Fever--Cou...	Non-smoker
2020-10-25	13	65	150	Sneezing	Non-smoker
2020-10-24	28	62	173	Headache	Non-smoker
2020-10-24	30	73	170	Fever--Rhin...	Non-smoker
2020-10-24	13	56	153	No sympto...	Non-smoker

1-10 of 971 rows

Previous **1** 2 3 4 5 ... 98 Next

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
```

number	age	age_group	sex_male...	marital_s...	height_m...	weig
7	60.8	Sixties	0	married	1.57	58
8	72.3	Seventies	1	married	1.65	72
9	62.6	Sixties	0	married	1.59	64
12	72	Seventies	0	widow	1.473	54.5
13	60.1	Sixties	0	married	1.55	47
19	60.6	Sixties	0	married	1.422	64
45	60.1	Sixties	1	widower	1.68	60
46	60.2	Sixties	0	married	1.8	64.6
51	63	Sixties	0	married	1.6	57.8
56	60.4	Sixties	0	married	1.6	71.8

1-10 of 239 rows

Previous 1 2 3 4 5 ... 24 Next

Introducing `mutate()`



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 × 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 × 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?

SIDE NOTE



Sometimes it is helpful to think of data manipulation functions in the context of familiar spreadsheet software. Here is what the R command `mutate(height_m = height_cm/100)` would be equivalent to in Google Sheets:

SIDE NOTE

	A	B
1	height_cm	
2	169	
3	185	
4	180	
5	164	
6	147	
7	162	
8	150	
9		

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 × 1
##   height_cm
##       <dbl>
## 1     174
## 2     190
## 3     185
## 4     169
## 5     152
```

Again, very easy to do!

PRACTICE

(in RMD)

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.

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 × 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 × 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 × 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 × 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



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
```

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 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 × 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)
```

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

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

PRACTICE



(in RMD)

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 # _____
  # _____
```

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.

Integer: `as.integer`

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

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

```
## # A tibble: 5 × 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



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 × 6
##   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
```

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

(in RMD)

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 # _____  
  # _____
```

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.

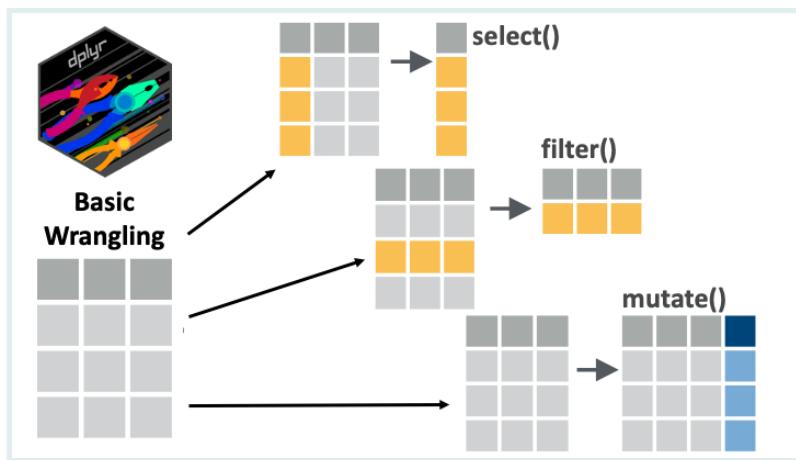


Fig: Basic Data Wrangling with `select()`, `filter()`, and `mutate()`.

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
 A firm believer in science for good, striving to ally programming, health and education



ANDREE VALLE CAMPOS

R Developer and Instructor, the GRAPH Network
 Motivated by reproducible science and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network
 Passionate about world improvement

References

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

- 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>.

Lesson notes | Conditional mutating

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Learning objectives
Packages
Datasets
Reminder: relational operators (comparators) in R
Introduction to <code>case_when()</code>
The TRUE default argument
Matching NA's with <code>is.na()</code>
Keeping default values of a variable
Multiple conditions on a single variable
Multiple conditions on multiple variables
Order of priority of conditions in <code>case_when()</code>
Overlapping conditions within <code>case_when()</code>
Binary conditions: <code>dplyr::if_else()</code>
Wrap up

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.



Fig: the `case_when()` conditions

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.

Packages

This lesson will require the tidyverse suite of packages:

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

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
```

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
```

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

<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

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
```

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"))
```

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’”.

Formulas, LHS and RHS

VOCAB



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.

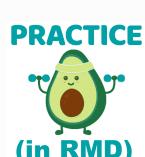
To inspect the variable, you can pipe your data frame into the `View()` function to view it in spreadsheet form:

```
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)
```



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 number without quotes:
Q_age_group_percentage <- YOUR_ANSWER_HERE
```

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"))
```

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'" .

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:

WATCH OUT



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

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"))
```



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 %>%
```



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

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

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 %>%
```

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
```

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"
    ))
```

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
    ))
```

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
    ))
```

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`'.



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
```

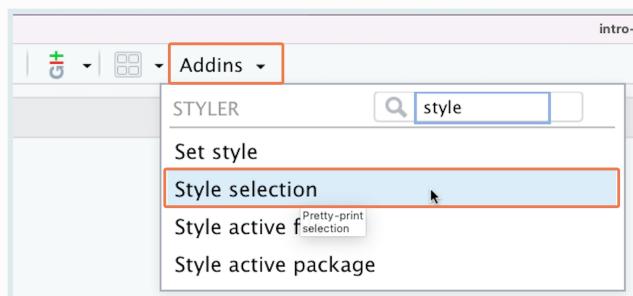
PRACTICE

(in RMD)

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

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.

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

```

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 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

```

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:

WATCH OUT



```
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



(in RMD)

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 %>%
```

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
```

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"
  ))
```

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)`.



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 = _____)
```

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:

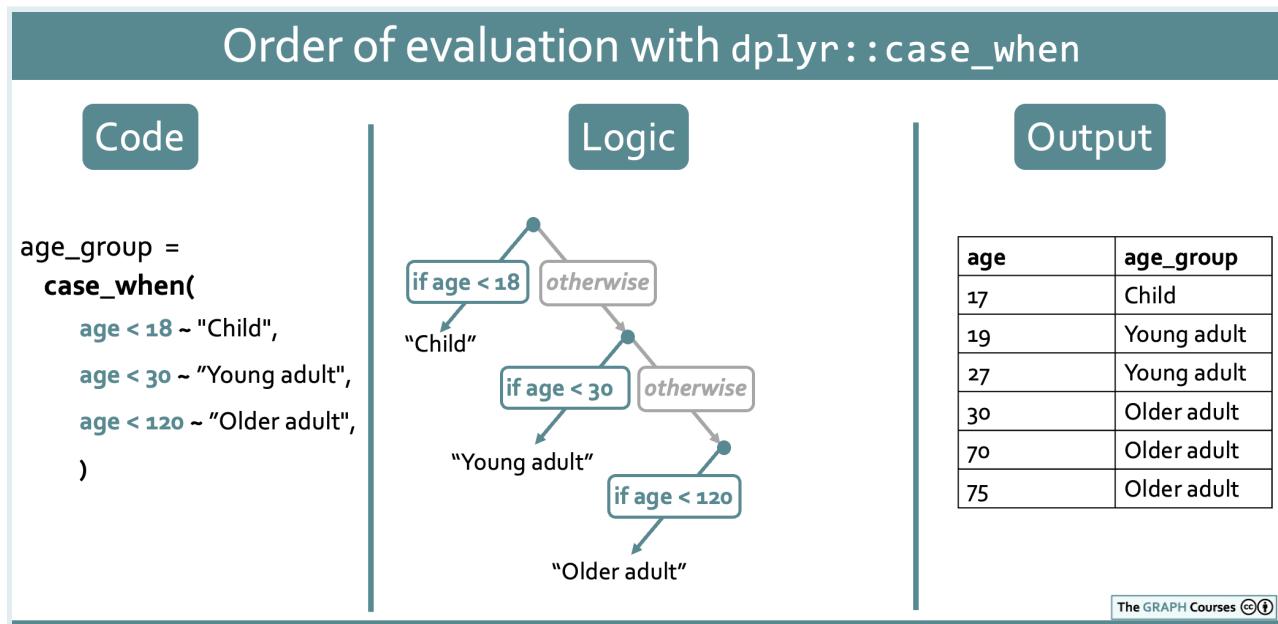
```
yaounde_age_sex %>%
  mutate(age_group = case_when(age_years < 18 ~ "Child",
                               age_years < 30 ~ "Young adult",
                               age_years < 120 ~ "Older adult"))
```

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:

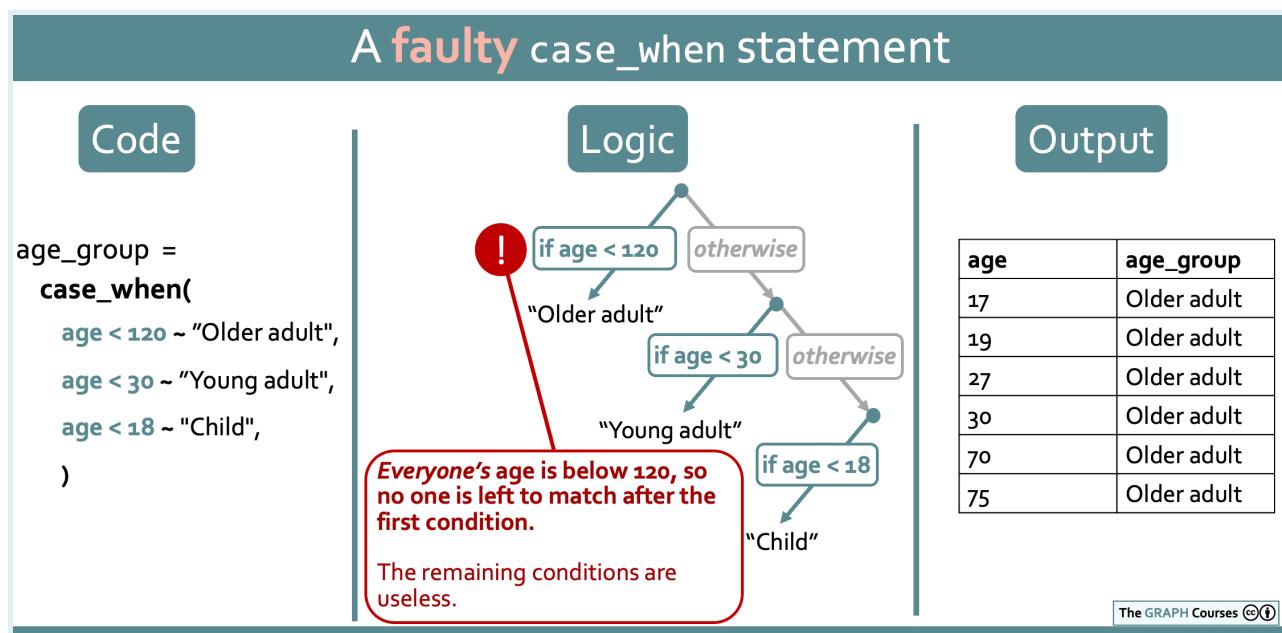


This means that if you swap the order of the conditions, you will end up with a faulty `case_when()` statement:

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

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"))
```

```
# 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"))
```

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.

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_"))
```

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
```

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"
  ))
```

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"
  ))
```

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.

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"))
```

PRACTICE (in RMD)



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 = _____
  )
```

Binary conditions: `dplyr::if_else()`

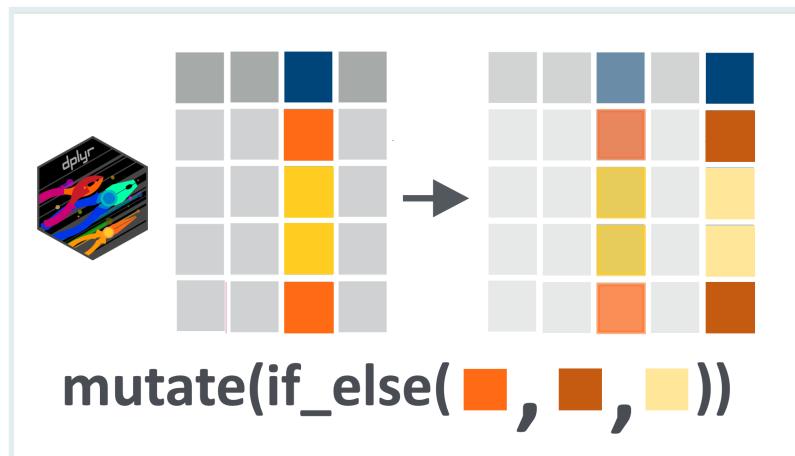


Fig: the `if_else()` conditions

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:

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

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
    ))
```

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

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.

PRACTICE



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(____))
```

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!



Fig: the conditional `mutate()` options

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement

References

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

- 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)

Lesson notes | Grouping and summarizing data

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Learning objectives
The Yaounde COVID-19 dataset
What are summary statistics?
Introducing <code>dplyr::summarize()</code>
Grouped summaries with <code>dplyr::group_by()</code>
Grouping by multiple variables (nested grouping)
Ungrouping with <code>dplyr::ungroup()</code> (why and how)
Counting rows
Counting rows that meet a condition
<code>dplyr::count()</code>
Including missing combinations in summaries
Wrap-up

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 !

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.
-

The Yaounde COVID-19 dataset

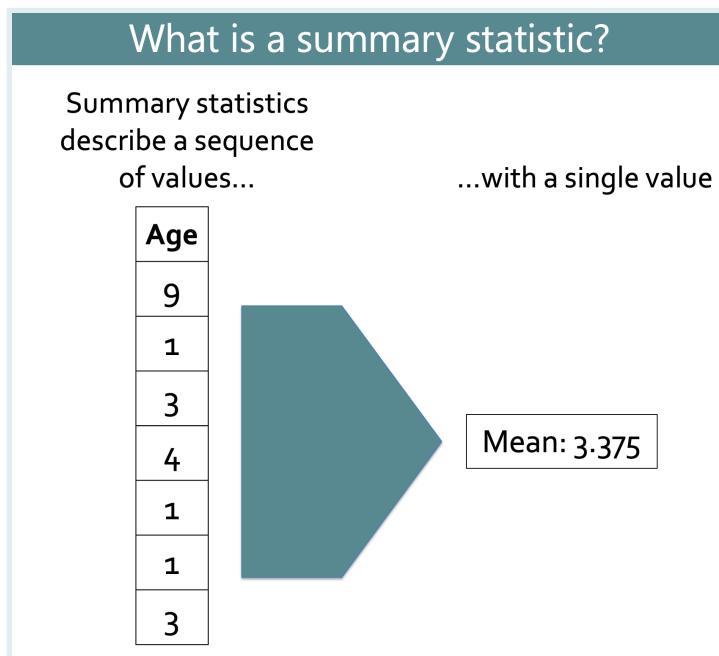
In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
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 × 15  
##       age age_category_3 sex    weight_kg height_cm  
##   <dbl> <chr>        <chr>      <dbl>      <dbl>  
## 1     45 Adult        Female      95       169  
## 2     55 Adult        Male       96       185  
## 3     23 Adult        Male       74       180  
## 4     20 Adult        Female      70       164  
## 5     55 Adult        Female      67       147  
## 6     17 Child        Female      65       162  
## 7     13 Child        Female      65       150  
## 8     28 Adult        Male       62       173  
## 9     30 Adult        Male       73       170  
## 10    13 Child        Female      56       153  
## # ... with 961 more rows, and 10 more variables:  
## #   neighborhood <chr>, is_smoker <chr>, ...
```

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

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).



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

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 × 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`).

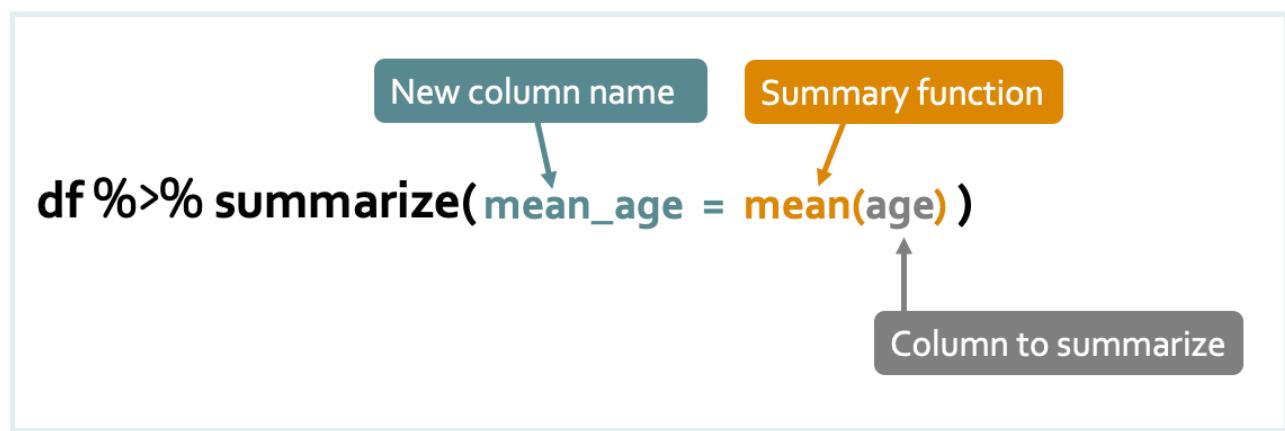


Fig. Basic syntax for the `summarize()` function.

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 × 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.



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 %>%
```



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()

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 × 15  
## # Groups:   sex [2]  
##       age age_category_3 sex     weight_kg height_cm  
##   <dbl> <chr>        <chr>      <dbl>      <dbl>  
## 1     45 Adult        Female     95       169  
## 2     55 Adult        Male      96       185  
## 3     23 Adult        Male      74       180  
## 4     20 Adult        Female     70       164  
## 5     55 Adult        Female     67       147  
## 6     17 Child         Female     65       162  
## 7     13 Child         Female     65       150  
## 8     28 Adult        Male      62       173  
## 9     30 Adult        Male      73       170  
## 10    13 Child         Female     56       153  
## # ... with 961 more rows, and 10 more variables:  
## #   neighborhood <chr>, is_smoker <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 × 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 × 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 × 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 × 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
## # ... with 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 × 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 × 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**(in RMD)**

is_smoker	weight_mean
Ex-smoker	
Non-smoker	
Smoker	
NA	

```
Q_weight_by_smoking_status <-
  yao %>%
```

PRACTICE**(in RMD)**

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 %>%
```



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 %>%
```

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) %>%
  summarise(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `groups` argument.
```

```
## # A tibble: 18 × 3
## # Groups:   sex [2]
```

```

## 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) %>%
  summarise(mean_age = mean(age))

```

```

## `summarise()` has grouped output by 'neighborhood'. You can override
## using the ` `.groups` argument.

```

```

## # A tibble: 18 × 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 Oliga Female  24.3
## 18 Tsinga Oliga 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.

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_iggresult <-
  yao %>%
```

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 × 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 × 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 × 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) %>%
  summarise(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 × 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 × 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 × 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
```

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 `summarise()`.

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 × 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 × 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

```



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 %>%
```

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 × 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 × 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

```

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.

CHALLENGE



```

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 × 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
## # ... with 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))
```

CHALLENGE



```
## # A tibble: 1 × 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 × 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>
## # ... with 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 × 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 × 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

```

Group your `yao` data frame by the respondents' symptoms (`symptoms`) and use the `sum()` function to count how many adults have each symptom combination.

PRACTICE



Your output should be a data frame with two columns named as shown below:

symptoms	sum_adults
----------	------------

```

Q_symptoms_adults <-
  yao %>%
  group_by(GROUPED VARIABLE HERE) %>%
  summarise(sum_adults = sum(HERE, INPUT A CONDITION TO MATCH
                             ADULTS))

```

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 × 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  
## # ... with 18 more rows
```

Note that this is the same output as:

```
yao %>%  
  group_by(occupation) %>%  
  summarize(n = n())
```

```
## # A tibble: 28 × 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  
## # ... with 18 more rows
```

You can also apply `dplyr::count()` in a nested fashion:

```
yao %>%  
  count(sex, occupation)
```

```
## # A tibble: 40 × 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
## # ... with 30 more rows

```

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 × 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 × 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

```

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 × 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 × 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 × 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 × 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 × 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
```

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
--------------	----------------	-----	------------

PRACTICE

(in RMD)

```
Q_median_age_by_neighborhood_agecategory_sex <-
  yao %>%
```

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:

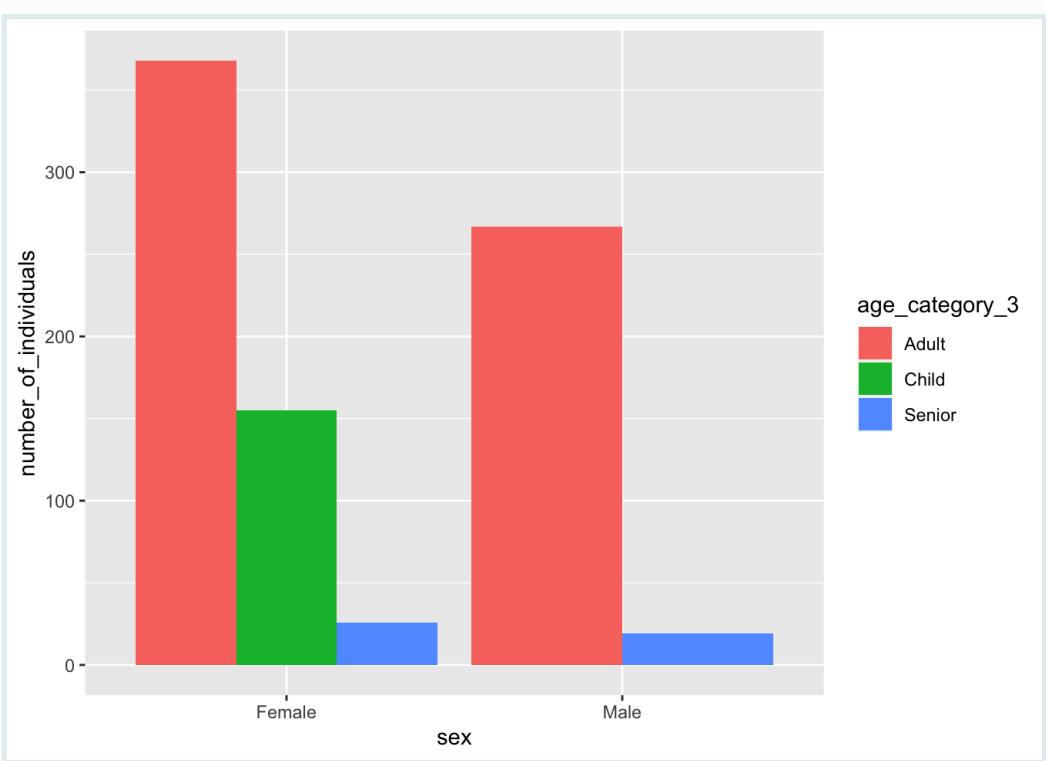
SIDE NOTE



```
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.
```



SIDE NOTE



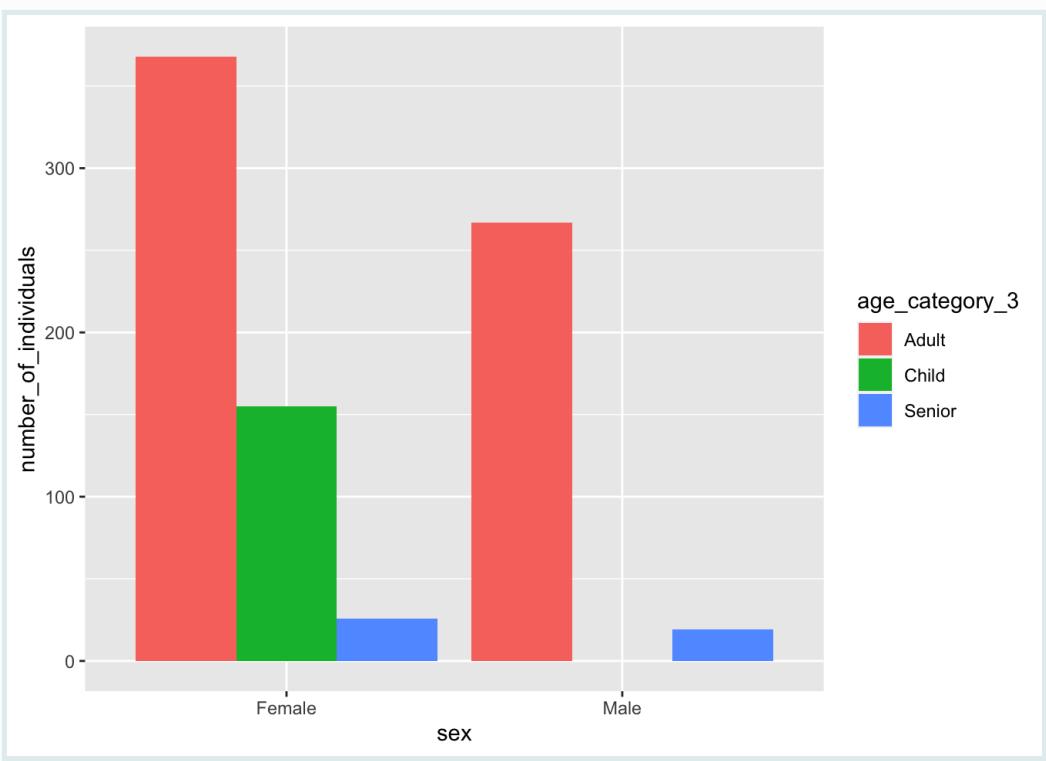
Not very elegant! Ideally there should be an empty space indicating 0 for the number of male children.

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.



SIDE NOTE Much better!

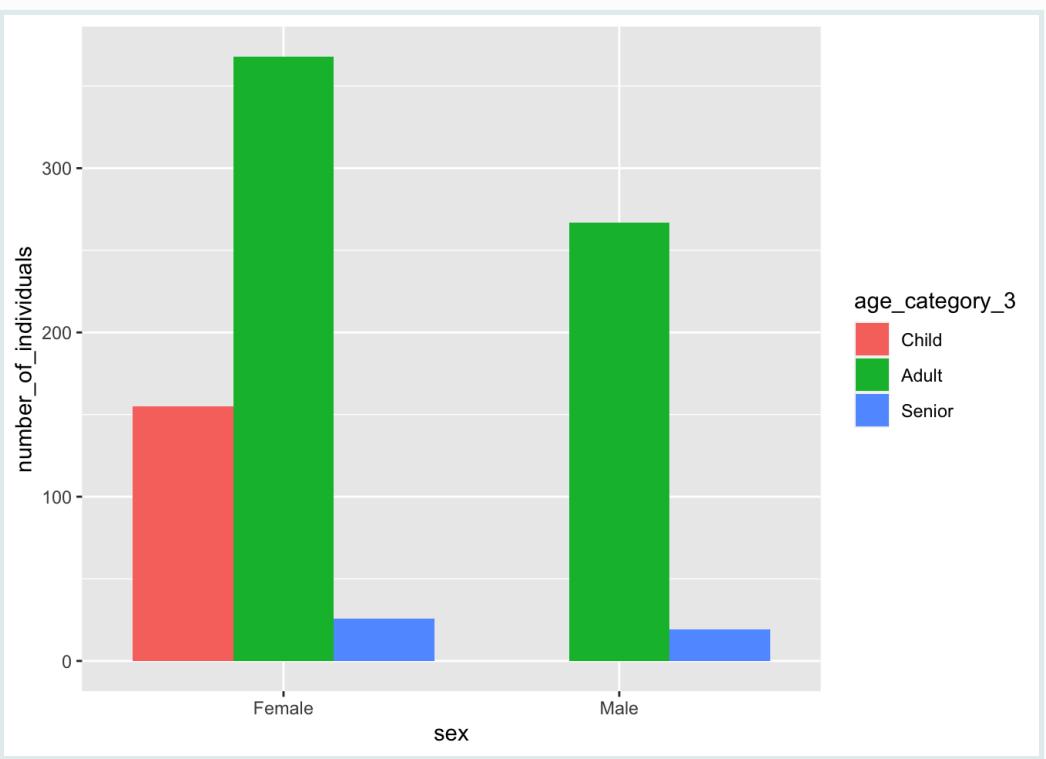


By the way, this output can be improved slightly by setting the factor levels for age to their proper ascending order: first "Child", then "Adult" then "Senior":

```
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.

SIDE NOTE

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 a one of the most common grouping manipulations.

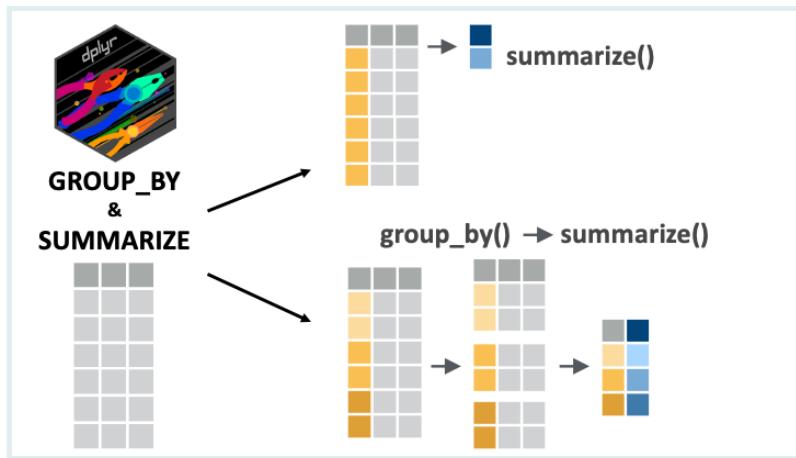


Fig: `summarize()` and `group_by()`

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 !

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education



ANDREE VALLE CAMPOS

R Developer and Instructor, the GRAPH Network
Motivated by reproducible science and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

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)

Lesson notes | Grouped filter, mutate and arrange

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Learning objectives
Packages
Datasets
Arranging by group
arrange() can group automatically
Filtering by group
Filtering with nested groupings
Mutating by group
Mutating with nested groupings
Wrap up

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.

Learning objectives

1. You can use `group_by()` with `arrange()`, `filter()`, and `mutate()` to conduct grouped operations on a data frame.

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)
```

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 × 7  
##   sex     age age_category weight_kg occupation  
##   <chr>    <dbl> <chr>          <dbl> <chr>  
## 1 Female    45 45 - 64           95 Informal worker  
## 2 Male      55 45 - 64           96 Salaried worker  
## 3 Male      23 15 - 29           74 Student  
## 4 Female    20 15 - 29           70 Student  
## 5 Female    55 45 - 64           67 Trader--Farmer  
## # ... with 2 more variables: igg_result <chr>,  
## #   igm_result <chr>
```

```
yao_sex_weight <-  
  yao %>%  
  select(sex, weight_kg)  
  
yao_sex_weight
```

```
## # A tibble: 5 × 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 × 9
##   number    age age_group sex_male_1_female_0 marital_status
##   <dbl> <dbl> <chr>          <dbl> <chr>
## 1     7 60.8 Sixties           0 married
## 2     8 72.3 Seventies         1 married
## 3     9 62.6 Sixties           0 married
## 4    12 72   Seventies         0 widow
## 5    13 60.1 Sixties           0 married
## # ... with 4 more variables: height_meters <dbl>,
## #   weight_kg <dbl>, grip_strength_kg <dbl>, ...
```

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 × 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 × 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 × 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 × 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 × 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 × 2
##   sex    weight_kg
##   <chr>     <dbl>
## 1 Female      162
## 2 Female      161
## 3 Female      158
## 4 Female      135
## 5 Female      129
```



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(_____)
```



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
```

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 × 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 × 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 × 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))
```

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)
```

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)
```

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
```

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 × 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 × 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 × 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 × 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 × 8
## # Groups:   sex, age_category [4]
##   sex    age age_category weight_kg occupation
##   <chr>  <dbl> <chr>          <dbl> <chr>
## 1 Female  45  45 - 64        95 Informal worker
## 2 Male    55  45 - 64        96 Salaried worker
## 3 Male    23  15 - 29        74 Student
## 4 Female  20  15 - 29        70 Student
## 5 Female  55  45 - 64        67 Trader--Farmer
## # ... with 3 more variables: igg_result <chr>,
## #   igm_result <chr>, weight_rank <dbl>

```

The output shows that the person in the first row is 20th heaviest *woman in the 45 to 64 age group*.



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 × 8
## # Groups:   sex, age_category [4]
##   sex      age age_category weight_kg occupation
##   <chr>    <dbl> <chr>          <dbl> <chr>
## 1 Female    45  45 - 64        95 Informal worker
## 2 Male      55  45 - 64        96 Salaried worker
## 3 Male      23  15 - 29        74 Student
## 4 Female    20  15 - 29        70 Student
## 5 Female    55  45 - 64        67 Trader--Farmer
## # ... with 3 more variables: igg_result <chr>,
## #   igm_result <chr>, weight_rank <dbl>
```

WATCH OUT



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 × 8
## # Groups:   sex, age_category [5]
##   sex      age age_category weight_kg occupation
##   <chr>    <dbl> <chr>          <dbl> <chr>
## 1 Male     65  45 - 64        93 Retired
## 2 Male     78  65 +          95 Retired--Informal
## 3 Male     14  5 - 14        44 Student
## 4 Female   44  30 - 44        67 Home-maker
## 5 Female   79  65 +          40 Retired
```

```
## # ... with 3 more variables: igg_result <chr>,
## #   igm_result <chr>, 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

WATCH OUT



```
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 × 8  
##   sex      age age_category weight_kg occupation igg_result  
##   <chr>    <dbl> <chr>           <dbl> <chr>     <chr>  
## 1 Female    79 65 +            40 Retired   Negative  
## 2 Female    79 65 +            81 Home-maker Negative  
## # ... with 2 more variables: igm_result <chr>,  
## #   weight_rank <dbl>
```

Wrap up

`group_by()` is a marvelous tool for arranging, mutating, filtering based on the groups within a single or multiple variables.

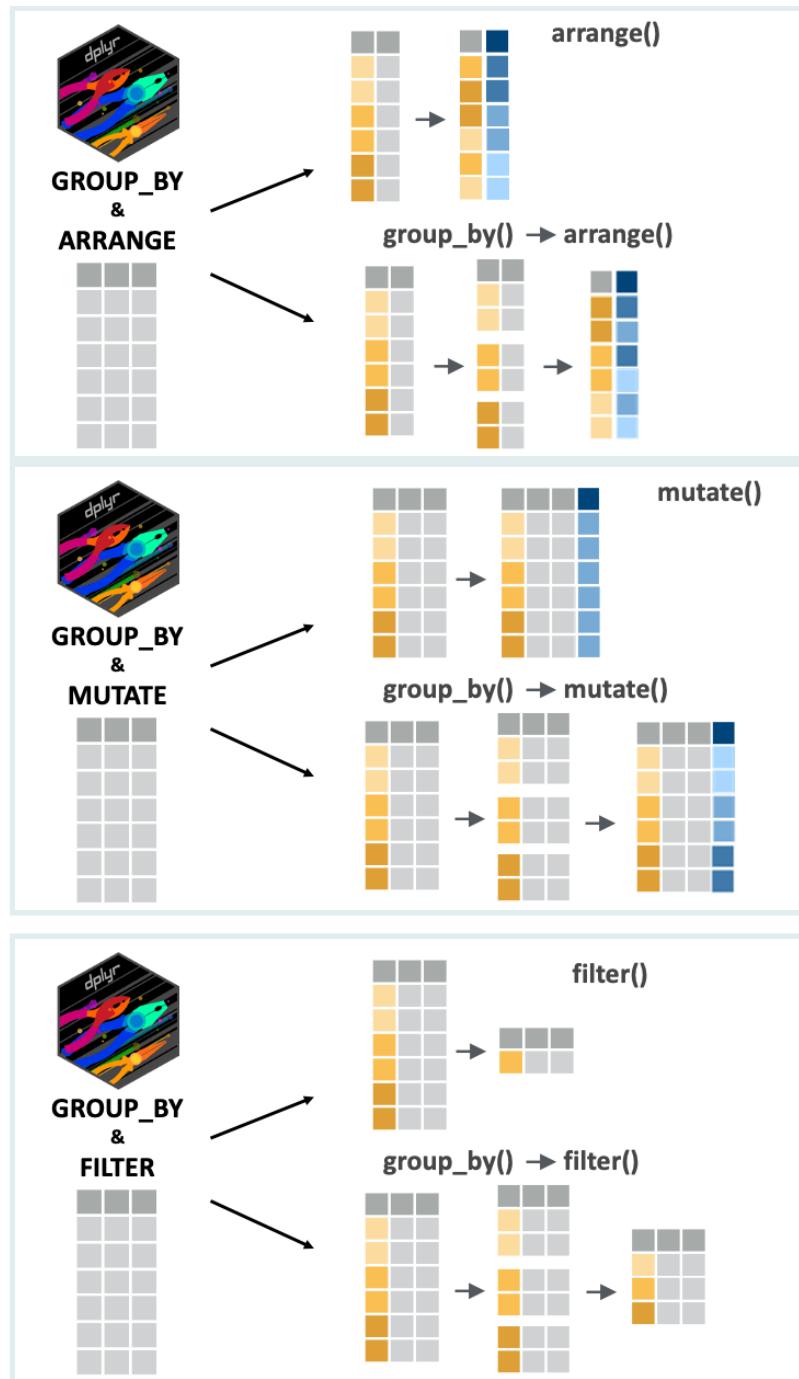


Fig: `filter()` and `group_by()`

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!

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement

References

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

- 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)

Lesson notes | The across function

Created by the GRAPH Courses team

January 2023

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

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

Intro
Learning objectives
Packages
Datasets
Using <code>across()</code> with <code>mutate()</code>
The <code>.cols</code> argument
The <code>.fns</code> argument
Custom (“anonymous”) functions
Creating new columns with the <code>.names</code> argument
Using <code>across()</code> with <code>summarize()</code>
Multiple summary statistics
Recap !
Wrap up !

Intro

In previous lessons, you learned how to perform a range of wrangling operations like filtering, mutating and summarizing. But so far, you only performed these operations *one column at a time*. Sometimes however, it will be useful (and efficient) to apply the same operation to *several columns at the same time*. For this, the `across()` function can be used.

Let's see how!

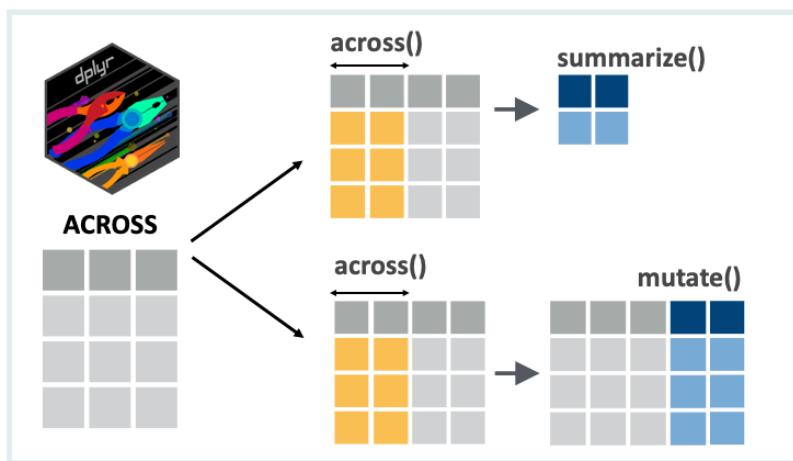


Fig: the `across()` verb.

Learning objectives

1. You can use `across()` with the `mutate()` and `summarize()` verbs to apply operations over multiple columns.

2. You can use the `.names` argument within `mutate(across())` to create new columns.
3. You can write anonymous (lambda) functions within `across()`

Packages

This lesson will require the packages loaded below:

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

Datasets

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

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

yaounde <- yaounde %>% rename(age_years = age)

yaounde

## # A tibble: 5 × 53
##   id                  date_surveyed age_years age_category
##   <chr>                <date>        <dbl> <chr>
## 1 BRIQUETERIE_000_0001 2020-10-22      45  45 - 64
## 2 BRIQUETERIE_000_0002 2020-10-24      55  45 - 64
## 3 BRIQUETERIE_000_0003 2020-10-24      23  15 - 29
## 4 BRIQUETERIE_002_0001 2020-10-22      20  15 - 29
## 5 BRIQUETERIE_002_0002 2020-10-22      55  45 - 64
## # ... with 49 more variables: age_category_3 <chr>,
## #     sex <chr>, highest_education <chr>, occupation <chr>, ...
```

We will also use data from a hospital [study](#) conducted in Burkina Faso, in which a range of clinical data was collected from patients with febrile (fever-causing) diseases, with the aim of predicting the cause of the fever.

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

```

## # A tibble: 5 × 36
##   age_category    sexe pretreatment_ma...¹ pretreatment_an...²
##   <chr>          <chr>  <chr>                <chr>
## 1 5 years or old... 2      do not know      currently taking
## 2 5 years or old... 2      no                  no
## 3 5 years or old... female currently taking no
## 4 5 years or old... female no                  currently taking
## 5 5 years or old... female no                  no
## # ... with 32 more variables: onset_fever <dbl>,
## #   abd_pain <chr>, diarrhoea <chr>, runny_nose <chr>, ...

```

Finally, we will use data from a dietary diversity [survey](#) conducted in Vietnam, in which women were asked to recall (one or several days) the foods and drinks they consumed the previous day.

```

diet <- read_csv(here("data/vietnam_diet_diversity.csv"))
diet <- diet %>% rename(household_id = hhid)

diet

```

```

## # A tibble: 5 × 45
##   household_id date_of_visit     age_y age_group
##   <dbl>        <dttm>       <dbl>  <chr>
## 1 278 2017-05-23 00:00:00 47  40-49
## 2 348 2017-06-17 00:00:00 34  30-39
## 3 354 2017-06-17 00:00:00 37  30-39
## 4 324 2017-06-17 00:00:00 35  30-39
## 5 209 2017-06-07 00:00:00 35  30-39
## # ... with 41 more variables: kilocalories_consumed <dbl>,
## #   water_consumed_grams <dbl>, ...

```

Using `across()` with `mutate()`

The `mutate()` function gives you an easy way to create new variables or modify in place variables.

But sometimes you have a large number of columns to operate on, and typing out `mutate()` statements line-by-line can become onerous. In such cases `across()` can radically simplify and shorten your code.

Let's see an example.

Consider the symptoms columns (from `symp_fever` to `symp_stomach_ache`) in the `yaounde` data frame:

```
yao_symptoms <-
  yaounde %>%
  select(age_years, sex, date_surveyed, symp_fever:symp_stomach_ache)

yao_symptoms
```

```
## # A tibble: 5 × 16
##   age_years   sex   date_surveyed   symp_fever   symp_headache
##       <dbl>   <chr>   <date>       <chr>       <chr>
## 1       45 Female 2020-10-22    No          No
## 2       55 Male   2020-10-24    No          No
## 3       23 Male   2020-10-24    No          No
## 4       20 Female 2020-10-22    No          No
## 5       55 Female 2020-10-22    No          No
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

The **13 columns** between `symp_fever` and `symp_stomach_ache` indicate whether or not each respondent had a specific COVID-compatible symptom.

Now, imagine you wanted to convert all these columns to upper case. (That is, “Yes” to “YES” and “No” to “NO”). How might you do this? Without `across()`, you would have to mutate the columns one by one, with the `toupper()` function:

```
yao_symptoms %>%
  mutate(symp_fever = toupper(symp_fever),
        symp_headache = toupper(symp_headache),
        symp_cough = toupper(symp_cough),
        symp_rhinitis = toupper(symp_rhinitis),
        symp_sneezing = toupper(symp_sneezing),
        symp_fatigue = toupper(symp_fatigue),
        symp_muscle_pain = toupper(symp_muscle_pain)
      #... And on and on and on and on and on
      )
```

```
## # A tibble: 5 × 16
##   age_years   sex   date_surveyed   symp_fever   symp_headache
##       <dbl>   <chr>   <date>       <chr>       <chr>
## 1       45 Female 2020-10-22    NO          NO
## 2       55 Male   2020-10-24    NO          NO
## 3       23 Male   2020-10-24    NO          NO
## 4       20 Female 2020-10-22    NO          NO
## 5       55 Female 2020-10-22    NO          NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

This is obviously not very time-efficient. An experienced data analyst who saw this code might scold you for not obeying the **DRY (“Don’t Repeat Yourself”) principle of programming**.

But with the `across()` function, you have the power to do this in all of two lines:

```
yao_symptoms %>%  
  mutate(across(.cols = symp_fever:symp_stomach_ache,  
              .fns = toupper))
```

```
## # A tibble: 5 × 16  
##   age_years sex    date_surveyed symp_fever symp_headache  
##       <dbl> <chr>   <date>      <chr>      <chr>  
## 1        45 Female 2020-10-22    NO         NO  
## 2        55 Male   2020-10-24    NO         NO  
## 3        23 Male   2020-10-24    NO         NO  
## 4        20 Female 2020-10-22    NO         NO  
## 5        55 Female 2020-10-22    NO         NO  
## # ... with 11 more variables: symp_cough <chr>,  
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

Amazing!

Let's break down the code above. We used `across()` inside of the `mutate()` function, and provided it with two main arguments:

- `.cols` defined the *columns* to be modified. The `symp_fever:symp_stomach_ache` code means “all columns between `symp_fever` and `symp_stomach_ache`”.
- `.fns` defined the *functions* to apply on the selected columns. In this case, the `toupper` function was applied.

And that's the basic gist of `across()`! But below we'll consider each of these arguments in a bit more detail.

Why follow the DRY (Don't Repeat Yourself) principle?

There are many reasons to avoid repetitive code. Here are just a few:

SIDE NOTE



1. You'll save time in writing the code (obviously).
2. You'll also save time in *maintaining* the code. This is because if you need to make a change (e.g. switch `toupper` to `tolower`), you won't need to make the same change in several places. You can fix it in a single place.
3. DRY code is usually easier to read and understand, both by yourself and by others.

The `.cols` argument

Now let's look at the arguments of `across()` in some more detail.

As mentioned above, the `.cols` argument of `across()` selects the columns to be modified.

Most the different methods you have learned for selecting columns can be used here.

One difference with the classical use of `select()` is that to list column names with `across()`, you must wrap them in `c()`:

```
yao_symptoms %>%
  mutate(across(.cols = c(symp_fever, symp_headache, symp_cough),
               .fns = toupper))
```

If, instead of `c(symp_fever, symp_headache, symp_cough)` you just put in `symp_fever, symp_headache, symp_cough`, you'll get an error:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever, symp_headache, symp_cough, # Don't do this
               .fns = toupper))
```

```
Error in `mutate()` :
! Problem while computing `..1 = across(.cols = symp_fever, symp_headache....
```

Other than that, the usual variable selection methods can be used here.

So you can use numeric ranges, like `4:16`:

```
yao_symptoms %>%
  mutate(across(.cols = 4:16,
               .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>      <chr>      <chr>
## 1        45 Female 2020-10-22  NO        NO
## 2        55 Male   2020-10-24  NO        NO
## 3        23 Male   2020-10-24  NO        NO
## 4        20 Female 2020-10-22  NO        NO
## 5        55 Female 2020-10-22  NO        NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

Or helper verbs like `starts_with()`:

```
yao_symptoms %>%
  mutate(across(.cols = starts_with("symp_"),
               .fns = toupper))

## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>        <chr>      <chr>
## 1       45 Female 2020-10-22    NO         NO
## 2       55 Male   2020-10-24    NO         NO
## 3       23 Male   2020-10-24    NO         NO
## 4       20 Female 2020-10-22    NO         NO
## 5       55 Female 2020-10-22    NO         NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

Or the function `where()` to select columns of a particular type:

```
yao_symptoms %>%
  mutate(across(.cols = where(is.character),
               .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>        <chr>      <chr>
## 1       45 FEMALE 2020-10-22    NO         NO
## 2       55 MALE   2020-10-24    NO         NO
## 3       23 MALE   2020-10-24    NO         NO
## 4       20 FEMALE 2020-10-22    NO         NO
## 5       55 FEMALE 2020-10-22    NO         NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

Or the catch-all `everything()`:

```
yao_symptoms %>%
  mutate(across(.cols = everything(),
               .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <chr>   <chr>   <chr>        <chr>      <chr>
## 1 45       FEMALE 2020-10-22    NO         NO
## 2 55       MALE   2020-10-24    NO         NO
## 3 23       MALE   2020-10-24    NO         NO
## 4 20       FEMALE 2020-10-22    NO         NO
## 5 55       FEMALE 2020-10-22    NO         NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

Note that `everything()` is the default value for the `.cols`. So the above code, is equivalent to simply running:

```
yao_symptoms %>%  
  mutate(across(.fns = toupper))
```

```
## # A tibble: 5 × 16  
##   age_years sex    date_surveyed symp_fever symp_headache  
##   <chr>     <chr>   <chr>        <chr>      <chr>  
## 1 45       FEMALE 2020-10-22  NO        NO  
## 2 55       MALE   2020-10-24  NO        NO  
## 3 23       MALE   2020-10-24  NO        NO  
## 4 20       FEMALE 2020-10-22  NO        NO  
## 5 55       FEMALE 2020-10-22  NO        NO  
## # ... with 11 more variables: symp_cough <chr>,  
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

In the `febrile_diseases` dataset, the columns from `abd_pain` to `splenomegaly` indicate whether a patient had a specified symptom, recorded as “yes” or “no”.

```
febrile_diseases %>%  
  select(abd_pain:splenomegaly)
```



```
## # A tibble: 5 × 18  
##   abd_pain diarrhoea runny_nose earpains throat_ache cough  
##   <chr>     <chr>   <chr>        <chr>      <chr>      <chr>  
## 1 yes       yes     no          no        no        no  
## 2 yes       yes     no          no        no        yes  
## 3 yes       yes     no          no        no        yes  
## 4 yes       no      yes         no        no        yes  
## 5 yes       no      no          no        no        yes  
## # ... with 12 more variables: productive_cough <chr>,  
## #   dyspnoea <chr>, dysuria <chr>, myalgia <chr>, ...
```

Use `mutate()` and `across()` to convert the variable levels to uppercase. (That is, “yes” to “YES” and “no” to “NO”)

```
Q_febrile_disease_symptoms <-  
  febrile_diseases %>%
```

The `.fns` argument

Now, on to the second argument in `across()`. As mentioned above, this argument takes in the function to be applied across columns.

You can provide any valid function here. We had previously used `toupper()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>      <chr>      <chr>
## 1        45 Female 2020-10-22  NO         NO
## 2        55 Male   2020-10-24  NO         NO
## 3        23 Male   2020-10-24  NO         NO
## 4        20 Female 2020-10-22  NO         NO
## 5        55 Female 2020-10-22  NO         NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

In a similar style, we can also use `tolower()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = tolower))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>      <chr>      <chr>
## 1        45 Female 2020-10-22  no         no
## 2        55 Male   2020-10-24  no         no
## 3        23 Male   2020-10-24  no         no
## 4        20 Female 2020-10-22  no         no
## 5        55 Female 2020-10-22  no         no
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...
```

Of course, the function we apply through `across()` needs to be **type-appropriate**: it should apply to the type (character, numeric, factor, etc) of the variables we are feeding in.

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = log))
```

```
Error in `mutate()`:  
! non-numeric argument to mathematical function
```

Here we get an error message because we tried to apply a function made for numeric variables to character type variables.

SIDE NOTE



It is a bit confusing to write a function without parentheses, as in `.fns = toupper`. There is a difference between `toupper()` and `toupper`.

`toupper()` calls the function, while `toupper` without parenthesis makes a reference to the function. With a reference to the function, `across()` will take care of calling it from its back-end code (the code that defines `across()`). We call it “back-end” because it’s “in the back” and you cannot see it unless you go looking into it explicitly.)

PRACTICE

(in RMD)

In the `febrile_diseases` dataset, ensure that all the columns from `abd_pain` to `splenomegaly`, indicating symptoms of patients, are in lower case. Apply `tolower()` across all these variables using `mutate()` and `across()`.

```
Q_febrile_disease_symptoms_to_lower <-  
  febrile_diseases %>%
```

Custom (“anonymous”) functions

Sometimes it is useful to use a custom function, called a “lambda function” or “anonymous function”. You will see more about functions in later lessons. The idea here is that you write your own operation which will be applied across your selected variables. The writing of these lambda functions has certain strict rules so pay attention to this as we go through several examples.

The `toupper` example we saw above can be rewritten with this syntax:

```
yao_symptoms %>%  
  mutate(across(.cols = symp_fever:symp_stomach_ache,  
              .fns = ~ toupper(.x)))
```

```
## # A tibble: 5 × 16  
##   age_years sex     date_surveyed symp_fever symp_headache
```

```

## 1      45 Female 2020-10-22    NO      NO
## 2      55 Male   2020-10-24    NO      NO
## 3      23 Male   2020-10-24    NO      NO
## 4      20 Female 2020-10-22    NO      NO
## 5      55 Female 2020-10-22    NO      NO
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

In the code `.fns = ~ toupper(.x)`, the tilda, `~`, introduces the lambda function, and the `.x` references each of the columns across which you are applying the function. The `.x` takes the columns one by one and “calls” the function on each one.

So overall, this code can be read as “apply `toupper()` to each of the symptom variables.”

Here is another example, but with `tolower()`:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ tolower(.x)))

```

```

## # A tibble: 5 × 16
##   age_years sex   date_surveyed symp_fever symp_headache
##       <dbl> <chr> <date>        <chr>        <chr>
## 1      45 Female 2020-10-22    no          no
## 2      55 Male   2020-10-24    no          no
## 3      23 Male   2020-10-24    no          no
## 4      20 Female 2020-10-22    no          no
## 5      55 Female 2020-10-22    no          no
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

The pattern is quite simple once you get used to it.

Now, with this anonymous function syntax, it becomes very intuitive to use functions that take in multiple arguments.

For example, we could explicit the “No” and “Yes” by pasting into the string what they are referring to, in other words, symptoms:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ paste0(.x, " symptoms")))

```

```

## # A tibble: 5 × 16
##   age_years sex   date_surveyed symp_fever symp_headache
##       <dbl> <chr> <date>        <chr>        <chr>
## 1      45 Female 2020-10-22  No symptoms No symptoms
## 2      55 Male   2020-10-24  No symptoms No symptoms
## 3      23 Male   2020-10-24  No symptoms No symptoms

```

```

## 4      20 Female 2020-10-22    No symptoms No symptoms
## 5      55 Female 2020-10-22    No symptoms No symptoms
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

Or we could use `str_sub()`, a function that allows to keep a subset of your string:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ str_sub(.x, start = 1, end = 1)))

```

```

## # A tibble: 5 × 16
##   age_years sex   date_surveyed symp_fever symp_headache
##       <dbl> <chr> <date>        <chr>        <chr>
## 1       45 Female 2020-10-22    N            N
## 2       55 Male   2020-10-24    N            N
## 3       23 Male   2020-10-24    N            N
## 4       20 Female 2020-10-22    N            N
## 5       55 Female 2020-10-22    N            N
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

In our case our string values are “No” and “Yes” so we will make a substring with just their first letters (“N” and “Y”) to have a **one letter encoding**.

Or we can recode the “Yes” and “No” entries in a different manner:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ if_else(.x == "Yes", "Has symptom", "Does not have
symptom")))

```

```

## # A tibble: 5 × 16
##   age_years sex   date_surveyed symp_fever
##       <dbl> <chr> <date>        <chr>
## 1       45 Female 2020-10-22  Does not have symptom
## 2       55 Male   2020-10-24  Does not have symptom
## 3       23 Male   2020-10-24  Does not have symptom
## 4       20 Female 2020-10-22  Does not have symptom
## 5       55 Female 2020-10-22  Does not have symptom
## # ... with 12 more variables: symp_headache <chr>,
## #   symp_cough <chr>, symp_rhinitis <chr>, ...

```

Now we have the “Yes” encoded as “Has symptom” and the “No” encoded as “Does not have symptom”. These strings are longer but they are clearer in their meaning than just “Yes” vs. “No”.

We could also recode the “Yes” and “No” to numeric values:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ if_else(.x == "Yes", 1, 2)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>           <dbl>          <dbl>
## 1        45 Female 2020-10-22      2              2
## 2        55 Male   2020-10-24      2              2
## 3        23 Male   2020-10-24      2              2
## 4        20 Female 2020-10-22      2              2
## 5        55 Female 2020-10-22      2              2
## # ... with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, ...
```

Now we have “Yes” encoded as choice 1 and “No” as 2.

Note that you can chain several `mutate()` calls together:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ if_else(.x == "Yes", 1, 2))) %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ case_when(.x == 1 ~ 1,
                                  .x == 2 ~ 0)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>   <date>           <dbl>          <dbl>
## 1        45 Female 2020-10-22      0              0
## 2        55 Male   2020-10-24      0              0
## 3        23 Male   2020-10-24      0              0
## 4        20 Female 2020-10-22      0              0
## 5        55 Female 2020-10-22      0              0
## # ... with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, ...
```

Above, we first convert from “Yes” & “No” to the numeric values 1 and 2, to follow R indexing (R counts from 1 onwards). However, other programming language start their index at 0, such as Python (Python counts from 0 onwards). For many machine learning algorithms, your encoding should be 0 or 1, so this could be a useful conversion of the encoding of your data. We use `case_when()` to define which numerical value should be switched to 1 (TRUE, has symptoms) and which numerical value should be switched to 0 (FALSE, does not have symptoms).

PRACTICE





had a specified symptom, recorded as “yes” or “no”.

Use `mutate()`, `across()` and an anonymous function to convert the variable levels to numbers, with “yes” as 1 and “no” as 0.

```
Q_febrile_disease_symptoms_to_numeric <-  
  febrile_diseases %>%
```



In the `diet` dataset, the columns from `retinol` to `zinc` give the number of milligrams of each nutrient consumed by the surveyed women in a day.

```
diet %>%  
  select(retinol:zinc)
```

```
## # A tibble: 5 × 15  
##   retinol alpha_catoren beta_catoren beta_cryptoxanthin  
##   <dbl>     <dbl>      <dbl>          <dbl>  
## 1 0.998    0.0273     1.58          0.141  
## 2 3.53     0.114      3.66          0.0804  
## 3 1.32     0.388      13.9         0.0117  
## 4 1.08     0.0305     10.9         0.509  
## 5 1.25     0.102      9.66         0.164  
## # ... with 11 more variables: vitamin_c <dbl>,  
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, ...
```

Use `mutate()`, `across()` and an anonymous function to convert these values to grams (divide by 1000).

```
Q_diet_to_grams <-  
  diet %>%
```

Creating new columns with the `.names` argument

The examples we have seen so far all involved replacing existing columns.

But what if you want to create new columns instead?

To illustrate this, let’s create a smaller subset of `yao_symptoms`:

```
yao_symptoms_mini <-
  yao_symptoms %>%
  select(symp_fever, symp_headache, symp_cough)

yao_symptoms_mini
```

```
## # A tibble: 5 × 3
##   symp_fever symp_headache symp_cough
##   <chr>      <chr>        <chr>
## 1 No        No            No
## 2 No        No            No
## 3 No        No            No
## 4 No        No            No
## 5 No        No            No
```

Now, to convert all columns to uppercase, we would usually run:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper))
```

```
## # A tibble: 5 × 3
##   symp_fever symp_headache symp_cough
##   <chr>      <chr>        <chr>
## 1 NO        NO            NO
## 2 NO        NO            NO
## 3 NO        NO            NO
## 4 NO        NO            NO
## 5 NO        NO            NO
```

This code modifies existing columns *in place*

(Remember that the default argument for `.cols` is `everything()`, so the above code modifies all columns in the dataset)

Now, if we instead want to make *new* columns that are uppercase, we can use the `.names` argument of `across()`

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
               .names = "{.col}_uppercase"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough  symp_fever_uppercase
##   <chr>      <chr>        <chr>        <chr>
## 1 No        No            No            NO
## 2 No        No            No            NO
## 3 No        No            No            NO
## 4 No        No            No            NO
## 5 No        No            No            NO
```

```
## # ... with 2 more variables: symp_headache_uppercase <chr>,
## #   symp_cough_uppercase <chr>
```

{.col} represents each of the old column names. The rest of the string, “_uppercase” is pasted together with the old column names. So the code “{.col}_uppercase” code can be read as “for each column, convert to uppercase and name it by pasting the existing lowercase column name with _uppercase.”

Of course, we can input any arbitrary string:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
    .names = "{.col}_BIG LETTERS"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough symp_fever_BIG LETTERS
##   <chr>      <chr>       <chr>      <chr>
## 1 No        No          No         NO
## 2 No        No          No         NO
## 3 No        No          No         NO
## 4 No        No          No         NO
## 5 No        No          No         NO
## # ... with 2 more variables: symp_headache_BIG LETTERS <chr>,
## #   symp_cough_BIG LETTERS <chr>
```

If we want the text to come before the old column name, we can also do this:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
    .names = "uppercase_{.col}"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough uppercase_symp_fever
##   <chr>      <chr>       <chr>      <chr>
## 1 No        No          No         NO
## 2 No        No          No         NO
## 3 No        No          No         NO
## 4 No        No          No         NO
## 5 No        No          No         NO
## # ... with 2 more variables: uppercase_symp_headache <chr>,
## #   uppercase_symp_cough <chr>
```

More usefully, we can create a numeric version of these symptoms variables:

```
yao_symptoms_mini %>%
  mutate(across(.fns = ~ if_else(.x == "Yes", 1, 0),
    .names = "numeric_{.col}"))
```

```

## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough numeric_symp_fever
##   <chr>      <chr>       <chr>           <dbl>
## 1 No        No          No             0
## 2 No        No          No             0
## 3 No        No          No             0
## 4 No        No          No             0
## 5 No        No          No             0
## # ... with 2 more variables: numeric_symp_headache <dbl>,
## #   numeric_symp_cough <dbl>

```



Now you will convert again the columns from `abd_pain` to `splenomegaly` in the `febrile_diseases` dataset, on patient symptoms, into numerical values. But, you will create new columns named `numeric_abd_pain` to `numeric_splenomegaly` using the `.names` argument within `across()`.

```
Q_febrile_disease_symptoms_to_numeric_new_variables <-  
  febrile_diseases %>%
```

Using `across()` with `summarize()`

To get summary statistics over multiple variables it is often helpful to use `across()`.

Consider again the columns from `retinol` to `zinc` in the `diet` dataset, which indicate the number of milligrams of each nutrient consumed by surveyed Vietnamese women in a day:

```
diet %>%
  select(retinol:zinc)
```

```

## # A tibble: 5 × 15
##   retinol alpha_catoren beta_catoren beta_cryptoxanthin
##   <dbl>      <dbl>       <dbl>           <dbl>
## 1 0.998     0.0273     1.58            0.141
## 2 3.53      0.114      3.66            0.0804
## 3 1.32      0.388      13.9           0.0117
## 4 1.08      0.0305     10.9            0.509
## 5 1.25      0.102      9.66            0.164
## # ... with 11 more variables: vitamin_c <dbl>,
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, ...

```

Imagine you wanted to find the average amount of each nutrient consumed. To do this the usual way, you would need to type:

```
diet %>%
  summarize(mean_retinol = mean(retinol),
           mean_alpha_catorene = mean(alpha_catorene),
           mean_beta_catorene = mean(beta_catorene),
           mean_vitamin_c = mean(vitamin_c),
           mean_vitamin_b2 = mean(vitamin_b2)
           # And on and on and on for 15 columns
           )
```

```
## # A tibble: 1 × 5
##   mean_retinol mean_alpha_catorene mean_beta_catorene
##       <dbl>             <dbl>                  <dbl>
## 1      2.06            0.152                 6.15
## # ... with 2 more variables: mean_vitamin_c <dbl>,
## #   mean_vitamin_b2 <dbl>
```

Of course this is not very efficient.

But with `across()`, this can be done in just two lines:

```
diet %>%
  summarize(across(.cols = retinol:zinc,
                  .fns = mean))
```

```
## # A tibble: 1 × 15
##   retinol alpha_catorene beta_catorene beta_cryptoxanthin
##       <dbl>             <dbl>                  <dbl>
## 1      2.06            0.152                 6.15
## # ... with 11 more variables: vitamin_c <dbl>,
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, ...
```

And recall that one of the primary benefits of `summarize()` is that it facilitates grouped summaries. Well, we can still use those here!

```
diet %>%
  group_by(age_group) %>%
  summarize(across(.cols = retinol:zinc,
                  .fns = mean))
```

```
## # A tibble: 4 × 16
##   age_group retinol alpha_catorene beta_catorene
##   <chr>        <dbl>             <dbl>                  <dbl>
## 1 20-29        2.27            0.130                 5.37
## 2 30-39        2.92            0.164                 6.18
```

```
## # ... with 12 more variables: beta_cryptoxanthin <dbl>,
## #   vitamin_c <dbl>, vitamin_b2 <dbl>, vitamin_b3 <dbl>, ...
```

Beautiful! So much information extracted so easily.

Here we grouped the data by age group, then across all the nutrient variables, we calculated their mean by age group. It can be read as: "for the 40-49 years old age group, the mean consumption of retinol is roughly of 1.343 micrograms, which seems lower than for other age groups."

Let's see another example.

The columns from `is_drug_parac` to `is_drug_other` in the `yaounde` dataset indicate, as 1 or 0, whether or not a survey respondent was treated with the named drug:

```
yao_drugs <-
  yaounde %>%
  select(age_years, sex, date_surveyed, is_drug_parac:is_drug_other)

yao_drugs
```

```
## # A tibble: 5 × 12
##   age_years   sex   date_surveyed is_drug_parac
##       <dbl>   <chr>    <date>           <dbl>
## 1       45 Female 2020-10-22          1
## 2       55 Male   2020-10-24          NA
## 3       23 Male   2020-10-24          NA
## 4       20 Female 2020-10-22          0
## 5       55 Female 2020-10-22          NA
## # ... with 8 more variables: is_drug_antibio <dbl>,
## #   is_drug_hydrocortisone <dbl>, ...
```

How could we count the number of respondents who took each drug?

We can simply take the sum of each column selecting the columns intelligently and using the `sum()` function:

```
yao_drugs %>%
  summarize(across(.cols = starts_with("is_drug"),
                  .fns = sum))
```

```
## # A tibble: 1 × 9
##   is_drug_parac is_drug_antibio is_drug_hydrocortisone
##       <dbl>           <dbl>           <dbl>
## 1         NA            NA             NA
## # ... with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, ...
```

Oh no! we get all NAs!

We were smart and selected all our columns using `starts_with()` but we forgot to consider that `sum()` has `na.rm` set to `FALSE` by default. We need to ensure the `na.rm` argument is set to `TRUE`.

The best way to do this is with lambda/anonymous function syntax:

```
yao_drugs %>%
  summarize(across(.cols = starts_with("is_drug"),
    .fns = ~ sum(.x, na.rm = TRUE)))
```

```
## # A tibble: 1 × 9
##   is_drug_parac is_drug_antibio is_drug_hydrocortisone
##       <dbl>           <dbl>                  <dbl>
## 1          162            79                   14
## # ... with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, ...
```

Again, we could also create a grouped summary:

```
yao_drugs %>%
  group_by(sex) %>%
  summarize(across(.cols = starts_with("is_drug"),
    .fns = ~ sum(.x, na.rm = TRUE)))
```

```
## # A tibble: 2 × 10
##   sex   is_drug_parac is_drug_antibio is_drug_hydrocortis...¹
##   <chr>      <dbl>           <dbl>                  <dbl>
## 1 Female     93             42                   7
## 2 Male       69             37                   7
## # ... with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, ...
```

This last code chunk counts the number of individuals, per sex (group by `sex`), who have received each drug (summing the number of people across each drug variable).

A final example.

Recall that the 13 columns from `symp_fever` to `symp_stomach_ache` in the `yao_symptoms` dataset indicate whether or not each respondent had a specific COVID-compatible symptom:

```
yao_symptoms
```

```
## # A tibble: 5 × 16
##   age_years sex   date_surveyed symp_fever symp_headache
```

```

##      <dbl> <chr>  <date>      <chr>      <chr>
## 1     45 Female 2020-10-22 No        No
## 2     55 Male   2020-10-24 No        No
## 3     23 Male   2020-10-24 No        No
## 4     20 Female 2020-10-22 No        No
## 5     55 Female 2020-10-22 No        No
## # ... with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, ...

```

How would we count the number of people with each symptom using `across()`.

We have two options.

Option 1: We could first `mutate()` the "Yes" and "No" to numeric values:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ if_else(.x == "Yes", 1, 0)))

```

```

## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>          <dbl>           <dbl>
## 1     45 Female 2020-10-22      0              0
## 2     55 Male   2020-10-24      0              0
## 3     23 Male   2020-10-24      0              0
## 4     20 Female 2020-10-22      0              0
## 5     55 Female 2020-10-22      0              0
## # ... with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, ...

```

And then use `sum()` within `summarize()`:

```

yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
               .fns = ~ if_else(.x == "Yes", 1, 0))) %>%
  summarize(across(.cols = symp_fever:symp_stomach_ache,
                  .fns = sum))

```

```

## # A tibble: 1 × 13
##   symp_fever symp_headache symp_cough symp_rhinitis
##       <dbl>           <dbl>      <dbl>           <dbl>
## 1     143            135       130            89
## # ... with 9 more variables: symp_sneezing <dbl>,
## #   symp_fatigue <dbl>, symp_muscle_pain <dbl>, ...

```

Option 2: we could jump directly to `summarize()`, by summing with a condition:

```

yao_symptoms %>%
  summarize(across(.cols = symp_fever:symp_stomach_ache,

```

```

## # A tibble: 1 × 13
##   symp_fever symp_headache symp_cough symp_rhinitis
##       <int>        <int>       <int>        <int>
## 1      143         135        130         89
## # ... with 9 more variables: symp_sneezing <int>,
## #   symp_fatigue <int>, symp_muscle_pain <int>, ...

```

This code can be read as: across each symptom column, sum all individuals who have been recorded as receiving that drug (who have a “Yes” data entry for that drug).

PRACTICE



(in RMD)

In the diet data set, the variables `fao_fgwl` to `fao_fgw21` record the number of calories consumed from different FAO food groups. (FAO stands for “Food and Agricultural Organization”; the food groups are shown in Appendix 1.).

Use `summarize()` and `across()` to calculate the mean amount of calories obtained from each good group.

```

Q_diet_FAO_mean <-
  diet %>%

```

PRACTICE



(in RMD)

In the febrile_diseases data set, the columns from `abd_pain` to `splenomegaly` in the `febrile_diseases` dataset contain information on whether a patient had a specified symptom, recorded as “yes” or “no”. Use `summarize()`, `across()` to count the number of people with each symptom.

```

Q_febrile_disease_symptoms_count <-
  febrile_diseases %>%

```

PRACTICE



(in RMD)

In the yaounde data set, calculate the median for the age, height, weight, number of bedridden days and numer of days off from work (i.e. from the variable `age_years` to the variable `n_bedridden_days`)

Use `summarize()` and `across()`, giving the `.fns` argument a lambda function to calculate the median. Careful ! A lambda function with the



right arguments is indispensable, else you will have an `NA` median for some of the variables.

```
Q_yaounde_median <-  
  yaounde %>%
```

Multiple summary statistics

When we explored `summarize()` we rejoiced with the fact that we could calculate multiple summary statistics at the same time. This is also possible within `across()`.

Coming back to the diet data survey from Vietnam, we could calculate both the mean and the median across all the nutrient variables:

```
diet %>%  
  summarise(across(.cols = retinol:zinc,  
                  .fns = list(mean = mean,  
                             median = median)))
```

```
## # A tibble: 1 × 30  
##   retinol_mean retinol_median alpha_catoren_mean  
##       <dbl>        <dbl>            <dbl>  
## 1      2.06       0.724           0.152  
## # ... with 27 more variables: alpha_catoren_median <dbl>,  
## #   beta_catoren_mean <dbl>, beta_catoren_median <dbl>, ...
```

Here it is clear that on all numeric type variables of the data set, we want to calculate the mean and the median. We can do so by providing the `.fns` argument of `across()` with a list.



Small joke: `.fns` isn't "functions" abbreviated plural for nothing ! If we could only apply one function within `across()`, it would have been named `.fn` (function singular abbreviated).

This time, for the naming, `across()` takes care of naming the resulting summary statistic columns. The syntax is : `list(desired_name_1 = function_1, desired_name_2 = function_2)`.

Let's see how you could control the naming even more, when operating on a list of functions:

```
diet %>%
  summarise(across(.cols = retinol:zinc,
    .fns = list(average = mean, median = median),
    .names = "{.fn}_{.col}"))
```

```
## # A tibble: 1 × 30
##   average_retinol median_retinol average_alpha_catorene
##             <dbl>        <dbl>                  <dbl>
## 1          2.06       0.724                 0.152
## # ... with 27 more variables: median_alpha_catorene <dbl>,
## #   average_beta_catorene <dbl>, ...
```

Here we reference the name of the function using `{.fn}` and the name of the column with `{.col}`. It is important to note that **both abbreviations are singular!** They are singular because they reference the function and the column one by one. Within the `across()` procedure, `across()` takes the functions and the columns one by one and for each one, takes the function name, such as `average`, and the column name, such as `retinol`, and makes the summary statistic `average_retinol` (i.e. `{.fn}=average` and `{.col}=retinol`).

As we are discussing mean, median, standard deviation calculations, we have to anticipate for `NA` values. Consider the code below:

```
diet %>%
  summarise(across(.cols = retinol:zinc,
    .fns = list(average = ~ mean(.x, na.rm = TRUE),
                median = ~ median(.x, na.rm = TRUE)),
    .names = "{.fn}_{.col}"))
```

```
## # A tibble: 1 × 30
##   average_retinol median_retinol average_alpha_catorene
##             <dbl>        <dbl>                  <dbl>
## 1          2.06       0.724                 0.152
## # ... with 27 more variables: median_alpha_catorene <dbl>,
## #   average_beta_catorene <dbl>, ...
```

Here we have the same code as above, except we ensure that none of the means or medians will be `NA` by adding the `na.rm=TRUE` argument to the functions. For this, as we have seen above, we need to use the lambda/anonymous function style. Here we are giving the `.fns` argument a list of lambda functions.



In the `diet` data set, calculate the mean and the standard deviation for kilocalories, water, carbohydrates, fat, and protein consumed (i.e. from the variable `kilocalories_consumed` to the variable `carbs_consumed_grams`)

PRACTICE**(in RMD)**

Use `summarize()` and `across()`, giving the `.fns` argument a list of the desired summary statistics. Make sure your means are named `COLUMN_NAME_mean` and your standard deviations are named `COLUMN_NAME_sd`.

```
Q_diet_food_composition_mean_sd <-
  diet %>%
```

PRACTICE**(in RMD)**

In the `febrile_diseases` data set, calculate the mean and the standard deviation for white blood cells, and all other blood analysis measurements (i.e. from the variable `wbc` to the variable `relymp_a`, seeing Appendix 2 for the detailed names of the variable name abbreviations)

Use `summarize()` and `across()`, giving the `.fns` argument a list of the desired summary statistics. Careful! You need to give a list of lambda functions to calculate the mean and standard deviation, paying attention to the `na.rm` arguments, else your summary statistics will be set to `NA`.

Make sure your means are named `COLUMN_NAME_mean` and your standard deviations are named `COLUMN_NAME_sd`.

```
Q_febrile_diseases_mean_blood_composition <-
  febrile_diseases %>%
```

Recap!

`across()` can be used inside many different `{dplyr}` verbs:

- `mutate(across(multiple_columns, function(s) to apply))`
- `summarize(across(multiple_columns, function(s) to apply))`

The statement defining multiple columns can be:

- a list of names e.g. `c(symp_fever, symp_headache, symp_cough)`
- a range of names e.g. `retinol:zinc`

The function(s) to apply across all columns can be:

- an existing function of R (such as `as.factor`, `mean` etc.)
- a custom (lambda/anonymous) function
- a list of existing functions (such as `list(mean = mean, sd = sd)`)
- a list of custom (lambda/anonymous) functions

Wrap up !

This was your first approach to `across()`: congrats for making it through ! Remember the power of combination of `across()` and other verbs. If you feel a summarizing or mutation operation is identical for more than one variable, then usually you should think of using `across()`.

In the upcoming lessons we will see some more data wrangling verbs: see you soon !

Contributors

The following team members contributed to this lesson:



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement

References

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

- *Summarise each group to fewer rows.* (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/summarize.html>
- *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)

Appendix 1: FAO Food Groups

code	meaning
fao_fg1	Consumed amount from Foods made from grains
fao_fg2	Consumed amount from White roots and tubers and plantain
fao_fg3	Consumed amount from Pulses
fao_fg4	Consumed amount from Nuts and seeds
fao_fg5	Consumed amount from Milk and milk products
fao_fg6	Consumed amount from Organ meat
fao_fg7	Consumed amount from Meat and poultry
fao_fg8	Consumed amount from Fish and seafood
fao_fg9	Consumed amount from Eggs
fao_fg10	Consumed amount from Dark green leafy vegetables
fao_fg11	Consumed amount from Vitamin A-rich vegetables, roots and tubers
fao_fg12	Consumed amount from Vitamin A-rich fruits
fao_fg13	Consumed amount from Other vegetables
fao_fg14	Consumed amount from Other fruits
fao_fg15	Consumed amount from Insects and other small protein foods
fao_fg16	Consumed amount from Other oils and fats
fao_fg17	Consumed amount from Savoury and fried snacks
fao_fg18	Consumed amount from Sweets
fao_fg19	Consumed amount from Sugar sweetened beverages
fao_fg20	Consumed amount from Condiments and seasonings
fao_fg21	Consumed amount from Other beverages and foods

Appendix 2: Blood sample composition

Abbreviation	Complete Name
WBC	white bloodcell
RBC	red bloodcell
HGB	hemoglobin
PLT	platelet
NEUT_A	neutrophils
LYMP_A	lymphocytes
MONO_A	monocytes
EOSI_A	eosinophils
BASO_A	basophils
NRBC_A	nucleated red blood cells
IG_A	immature granulocytes
RET_A	reticulocytes
ASLYMP_A	antibody-synthesizing lymphocytes
RELYMP_A	reactive lymphocytes

Lesson notes | Pivoting data

Created by the GRAPH Courses team

January 2023

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

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

Intro
Learning Objectives
Packages
What do wide and long mean?
When should you use wide vs long data?
Pivoting wide to long
Pivoting long to wide
Why is long data better for analysis?
Filtering grouped data
Summarizing grouped data
Plotting
Pivoting can be hard
Wrap Up !

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 `tidyverse` package.

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.

Packages

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, outbreaks, janitor, rio, here, knitr)
```

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:

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

Fig: wide dataset for a timeseries of patients.

Or you could record the data in a long format as so :

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

Fig: long dataset for a timeseries of patients.

Take a minute to study the two datasets to make sure you understand the relationship between them.

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:

country	year	metric
x	1960	10
	1970	13
	2010	15
y	1960	20
	1970	23
	2010	25
z	1960	30
	1970	33
	2010	35

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

Fig: the equivalent wide dataset

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:

Hospital	Maternity unit	Intensive care unit
Hospital A	4	2
Hospital B	5	2
Hospital C	6	3

Fig: wide dataset, where each hospital is an observational unit

Hospital	Unit	Num. of patients
Hospital A	Maternity	4
Hospital A	Intensive care	
Hospital B	Maternity	5
Hospital B	Intensive care	
Hospital C	Maternity	6
Hospital C	Intensive care	

Fig: the equivalent long dataset

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.



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.



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
```

PRACTICE



(in RMD)

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
```

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**.

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.

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 × 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 × 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 × 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 × 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")
```



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 × 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
```

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 × 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 × 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 × 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.

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`.

SIDE NOTE



```
cases_per_district_per_year %>%
  pivot_wider(values_from = n,
              names_from = district)

## # A tibble: 2 × 15
##   year_of_onset     Bo Bombali Bonthe Kailahun Kambia Kenema
##   <dbl> <int>   <int>   <int>   <int>   <int>   <int>
## 1       2014     397    1070      7     535    127    641
## 2       2015     209     120     77     35    294    139
## # ... with 8 more variables: Koinadugu <int>, Kono <int>,
## # Moyamba <int>, `Port Loko` <int>, Pujehun <int>, ...
```

Here the unique observation units (our rows) are now the years (2014, 2015).



The population dataset from the `tidyverse` 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 <-  
  tidyverse::population
```

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.

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 × 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 × 8
## # Rowwise:
##   country      x2010 x2011 x2012 x2013 x2014 x2015
##   <chr>        <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
## # ... with 1 more variable: max_count <dbl>
```

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()`?

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 :

SIDE NOTE



```
infant_deaths_wide %>%
  mutate(max_count = max(x2010, x2011, x2012, x2013, x2014,
                        x2015))
```

```
## # A tibble: 5 × 8
##   country      x2010 x2011 x2012 x2013 x2014 x2015
##   <chr>        <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
```

SIDE NOTE

```
## 5 Argentina          9550 9230 8860 8480 8100 7720
## # ... with 1 more variable: max_count <dbl>
```

...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 `tidyverse` 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
```

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))
```

country	mean_deaths	sd_deaths
Afghanistan	68450	4466.
Albania	350.	45.2
Algeria	21033.	484.
Angola	72767.	4513.
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 × 8
## # Rowwise:
##   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
## # ... with 1 more variable: mean_deaths <dbl>

```

And finding the standard deviation would be very difficult. (We can't think of any way to achieve this, actually.)

For this practice question, you will again work with the long format population dataset from the `tidyverse` 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

```

Plotting

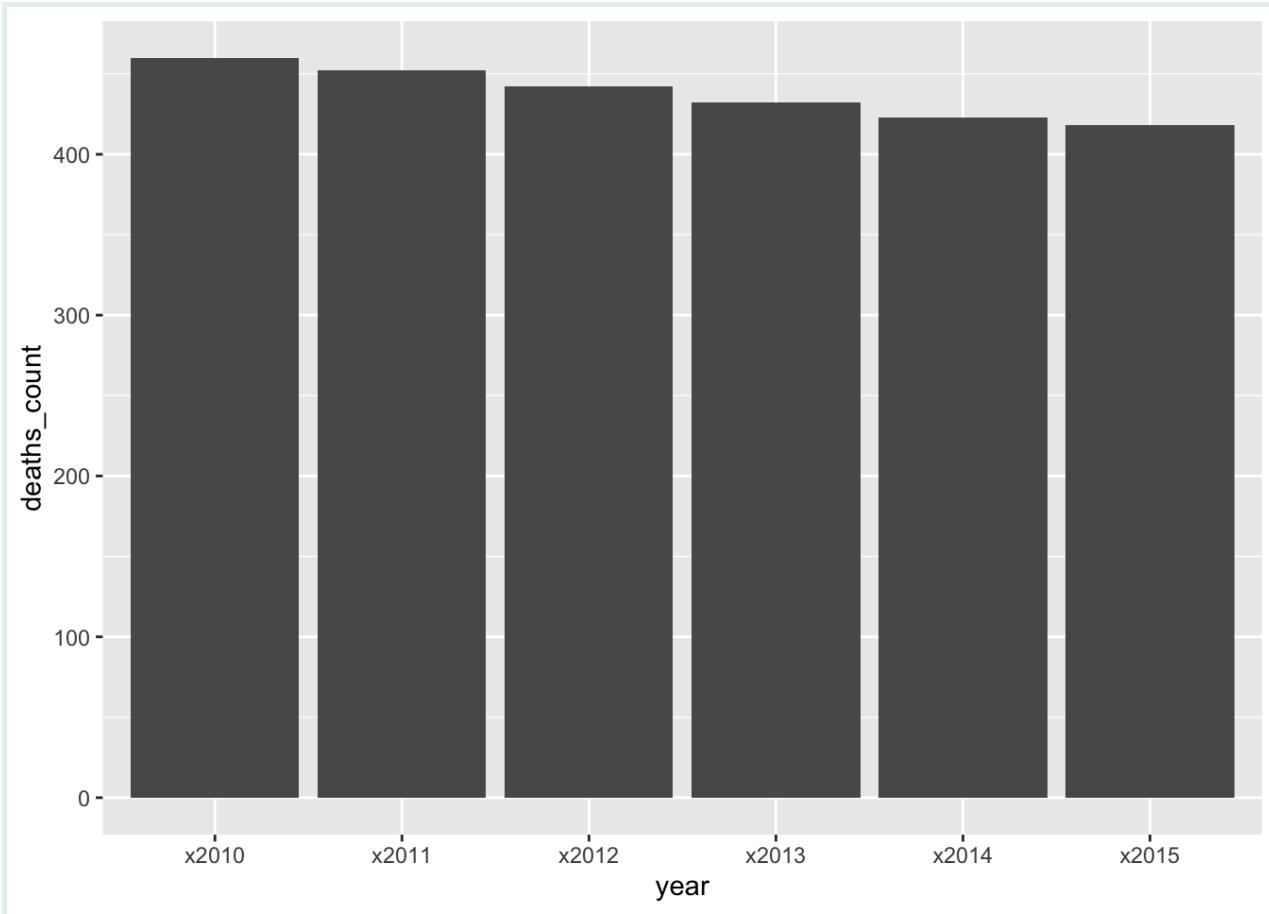
Finally, one of the data analysis tasks that is MOST hindered by wide formats is plotting. You may not yet have any prior knowledge of `{ggplot}` and how to plot so we will see the figures without going in depth with the code. What you need to remember is: many plots with `ggplot` are also only possible 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))

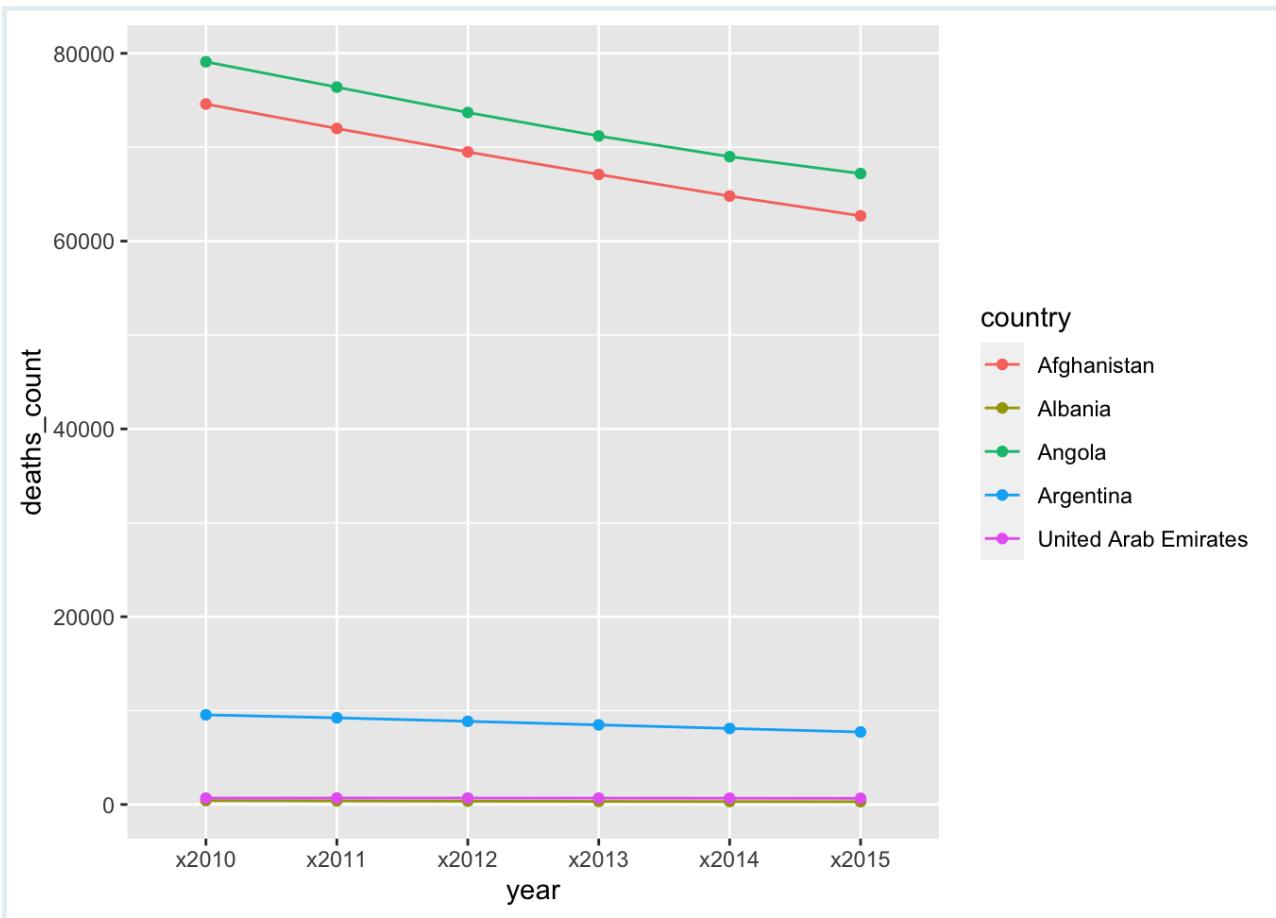
```



The plotting works because we can give the variable `year` for the x-axis. In the long format, `year` is a variable 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).

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 `tidyverse` team, as it is quite rich in examples. You could also post your questions about pivoting on forums like Stack Overflow.

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 !

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

Lesson notes | Advanced pivoting

Created by the GRAPH Courses team

January 2023

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

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

Intro
Learning Objectives
Packages
Datasets
Wide to long
Understanding <code>names_sep</code> and <code>".value"</code>
Value type <i>before</i> the separator
A non-time-series example
Escaping the dot separator
What to do when you don't have a neat separator?
Long to wide
Wrap Up !

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 !

Learning Objectives

1. Master complex pivoting from wide to long and long to wide
 2. Know how to use separators as a pivoting tool
-

Packages

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, outbreaks, janitor, rio, here, knitr)
```

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

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 × 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 × 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 × 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.

Consider this other artificial data set:

```
adult_stats <-
  tribble::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"
  )
```

PRACTICE



(in RMD)

```
## # A tibble: 3 × 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
-------	------	-----	-----

PRACTICE

(in RMD)

```
# Q_adult_long <-  
#   adult_stats %>%  
#   pivot_longer(____)
```

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
```

SIDE NOTE



```
## # A tibble: 5 × 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 × 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
```

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 × 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 × 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()`:

```
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 × 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
```

```
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 × 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 × 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 is 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 <-
  tribble::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 × 7
##   child t1_age t2_age t1_fat t2_fat t1_bmi t2_bmi
##   <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 × 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

```

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_
      "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 × 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(____)
#
growth_stats %>%
  pivot_longer(2:7,
               names_sep = "_",
               names_to = c("year", ".value"))

```

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 × 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 × 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!



Consider the following data set from Zambia about enteropathogens and their biomarkers.

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

```
## # A tibble: 297 × 7
##   ID    LPS_1  LPS_2  LBP_1  LBP_2  IFABP_1  IFABP_2
##   <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
```

```
enteropathogens_zambia_wide
```



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

```
enteropathogens_zambia_long <-
  enteropathogens_zambia_wide %>%
  pivot_longer(!ID,
              names_to = c(".value", "sample_count"),
              names_sep = "_")
```



```
## # A tibble: 5 × 5
##   ID sample_count    LPS     LBP  IFABP
##   <dbl> <chr>      <dbl>   <dbl>   <dbl>
## 1 1002 1          222. 38414. 1294.
## 2 1002 2          390. 6840.  610.
## 3 1003 1          181. 26888. 22.5
## 4 1003 2           NA     NA     NA
## 5 1004 1          257. 49183.  0
```

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 × 5
##   couple father_height father_weight mother_height
##   <chr>       <dbl>        <dbl>        <dbl>
## 1 a            180          80          160
## 2 b            185          90          150
## 3 c            182          93          143
## # ... with 1 more variable: mother_weight <dbl>
```

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 × 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.

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 × 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**



Consider again the `adult_stats` data you saw above. Now the column names have been changed slightly.

```

adult_stats_dot_sep <-
  tribble::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 × 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 <-
#   adult2_stats %>%
#   pivot_longer(______)

adult_stats_dot_sep %>%   pivot_longer(2:5,
                                         names_sep = "\\.",
                                         names_to = c(".value", "year"))

```

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, third_visit_location, third_visit_cost)

```

```
## Rows: 880 Columns: 22
## — Column specification ——————
## Delimiter: ","
## chr (10): Sex, Education, Employment, Alcohol, Smoking, Form of TB, Ch...
## dbl (12): id, Age, WtInKgs, HtInCms, bmi, Diabetes, first visit cost, ...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
tb_visits
```

```
## # A tibble: 5 × 7
##       id first_visit_location first_visit_cost
##   <dbl> <chr>                  <dbl>
## 1 100202 GH                      0
## 2 100396 Pvt. docto              1500
## 3 100590 Pvt. docto              2000
## 4 100687 Pvt. hospi             20000
## 5 100784 Pvt. docto              1000
## # ... with 4 more variables: second_visit_location <chr>,
## #   second_visit_cost <dbl>, third_visit_location <chr>, ...
```

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 × 7
##       id first_visit_location first_visit_cost
##   <dbl> <chr>                  <dbl>
## 1 100202 GH                      0
## 2 100396 Pvt. docto              1500
## 3 100590 Pvt. docto              2000
## 4 100687 Pvt. hospi             20000
## 5 100784 Pvt. docto              1000
## # ... with 4 more variables: second_visit_location <chr>,
## #   second_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 × 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 × 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()`.

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"))

```

```

## Rows: 61 Columns: 9
## — Column specification

## Delimiter: ","
## dbl (9): household_id, enerc_kcal_w_1, enerc_kcal_w_2,
dry_w_1, dry_w_...
##
## i Use `spec()` to retrieve the full column specification
for this data.
## i Specify the column types or set `show_col_types = FALSE`
to quiet this message.

```

diet_diversity_vietnam_wide



```

## # A tibble: 5 × 9
##   household_id enerc_kcal_w_1 enerc_kcal_w_2 dry_w_1
dry_w_2          <dbl>          <dbl>          <dbl>      <dbl>
<dbl>          348            2268.          1386.     548.
## 1           281.            2775.          1240.     600.
## 2           284.            3104.          2075.     646.
## 3           451.            2802.          2146.     620.
## 4           807.            1298.          1191.     269.
## 5           211.            1298.          1191.     269.
## # ... with 4 more variables: water_w_1 <dbl>,
## #   water_w_2 <dbl>, 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.

```

# Q_diet_diversity_vietnam_long <-
#   diet_diversity_vietnam_wide %>%
#   pivot_long(____)

```

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"))
```

```
## # Rows: 417 Columns: 5  
## — Column specification ——————  
## Delimiter: ","  
## dbl (5): ID, group, LPS, LBP, IFABP  
##  
## i Use `spec()` to retrieve the full column specification for this data.  
## i Specify the column types or set `show_col_types = FALSE` to quiet this  
message.
```

```
enteropathogens_zambia_long
```

```
## # A tibble: 5 × 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 × 7
##       ID LPS_1 LPS_2  LBP_1 LBP_2 IFABP_1 IFABP_2
##   <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"))
```

```
## # Rows: 122 Columns: 6
## └─ Column specification ──────────────────────────────────────────────────
##   Delimiter: ","
##   dbl (6): visit_number, household_id, energ_kcal_w, dry_w, water_w, fat_w
## 
##   i Use `spec()` to retrieve the full column specification for this data.
##   i Specify the column types or set `show_col_types = FALSE` to quiet this
## message.
```

```
diet_diversity_vietnam_long
```

```
## # A tibble: 5 × 6
##   visit_number household_id energ_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 × 9
##   household_id enerc_kcal_w_1 enerc_kcal_w_2 dry_w_1 dry_w_2
##       <dbl>          <dbl>        <dbl>    <dbl>    <dbl>
## 1         348          2268.      1386.     548.    281.
## 2         354          2775.      1240.     600.    284.
## 3         53          3104.      2075.     646.    451.
## 4         18          2802.      2146.     620.    807.
## 5         211          1298.      1191.     269.    288.
## # ... with 4 more variables: water_w_1 <dbl>,
## #   water_w_2 <dbl>, 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 × 4
##   household_id enerc_kcal_w_1 enerc_kcal_w_2
##       <dbl>          <dbl>        <dbl>
## 1         14          1040.      1663.
## 2         17          2100.      1286.
## 3         18          2802.      2146.
## 4         22          3187.      1582.

```

```
## 5           24          2359.          2026.  
## # ... with 1 more variable: total_energy_kcal <dbl>
```

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 × 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_renamed` dataset that we manipulated above and pivot it back to its wide format.

```
# Q_tb_visit_wide <-  
#   tb_visits_renamed %>%  
#   pivot_wider(____)
```

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.

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education

References

Lesson notes | Intro to ggplot2

Created by the GRAPH Courses team

January 2023

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

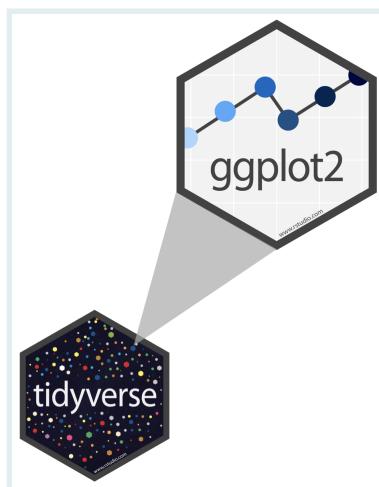
The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

Introduction
Learning objectives
Packages
Measles outbreaks in Niger
The <code>nigerm</code> dataset
The layered Grammar of Graphics
Working through the essential layers
Building a <code>ggplot()</code> in steps
Modifying the layers
Changing aesthetic mappings
Changing <code>geom_*</code> functions
Additional aesthetic mappings inside <code>aes()</code>
Fixed aesthetics outside <code>aes()</code>
Additional GG layers
Learning outcomes

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.



{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!

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.



Illustration by Allison Horst

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)
```

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.

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:

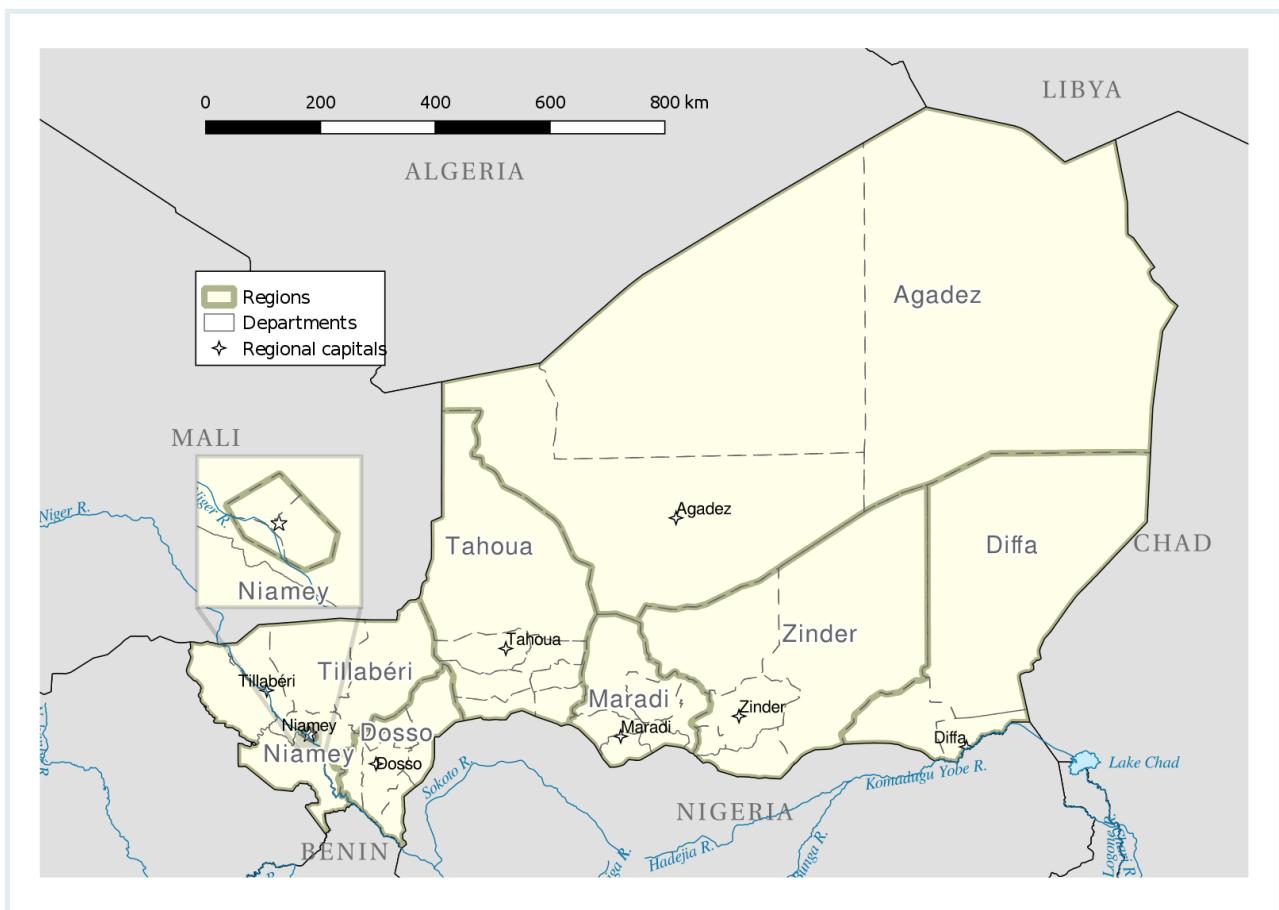
```
# Print Niger measles (nigerm) data frame
nigerm
```

The `nigerm` data frame has 4 variables (or columns):

1. **year**: Calendar year (ranges from 1995 to 2005)

3. **region**: Region in which the cases were recorded (see figure below)

4. **cases**: Number of measles cases reported



Administrative divisions of Niger: Districts and Regions

Several papers have investigated these trends, linking measles to human activity, migration, and seasonality.

JOURNAL OF THE ROYAL SOCIETY INTERFACE

**PROCEEDINGS
OF THE ROYAL SOCIETY B
BIOLOGICAL SCIENCES**

Open Access **Check for updates**

Research articles

Investigating persistent measles dynamics in Niger and associations with rainfall

Alexandre Blake, Ali Djibo, Ousmane Guindo and Nita Bharti

Published: 26 August 2020 | <https://doi.org/10.1098/rsif.2020.0480>

Restricted access **Check for updates**

Research articles

Rural–urban gradient in seasonal forcing of measles transmission in Niger

Matthew J. Ferrari, Ali Djibo, Rebecca F. Grais, Nita Bharti, Bryan T. Grenfell and Ottar N. Bjørnstad

Published: 28 April 2010 | <https://doi.org/10.1098/rspb.2010.0536>

Research articles that have used this dataset, and analyzed it in R!

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
```

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.

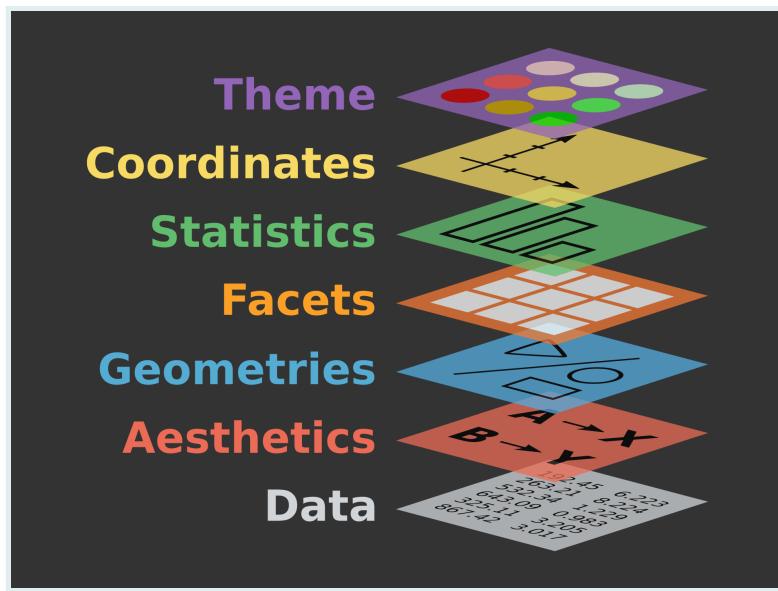
The layered Grammar of Graphics

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*.

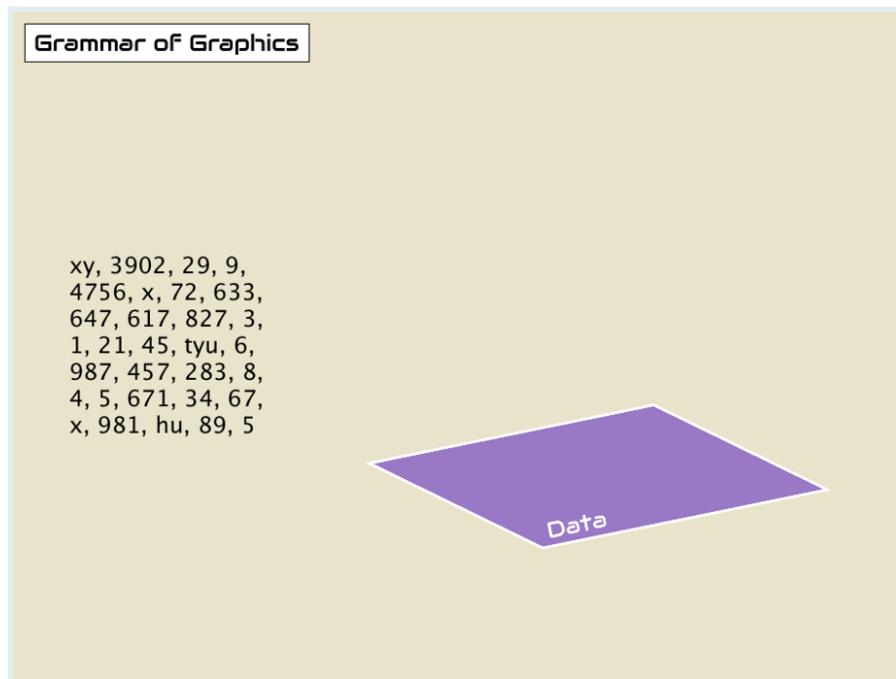


The Grammar of Graphics layers have specific names that you will see throughout the course.

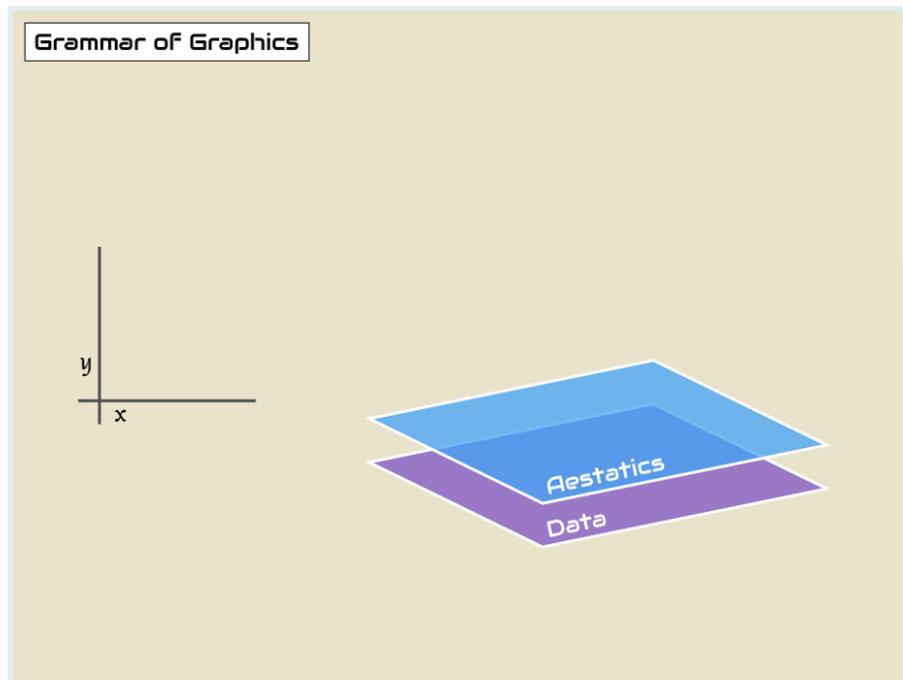
The three layers at the bottom of this figure - **data**, **aesthetics**, and **geometries** - are required for building any plot.

Let's define what they mean:

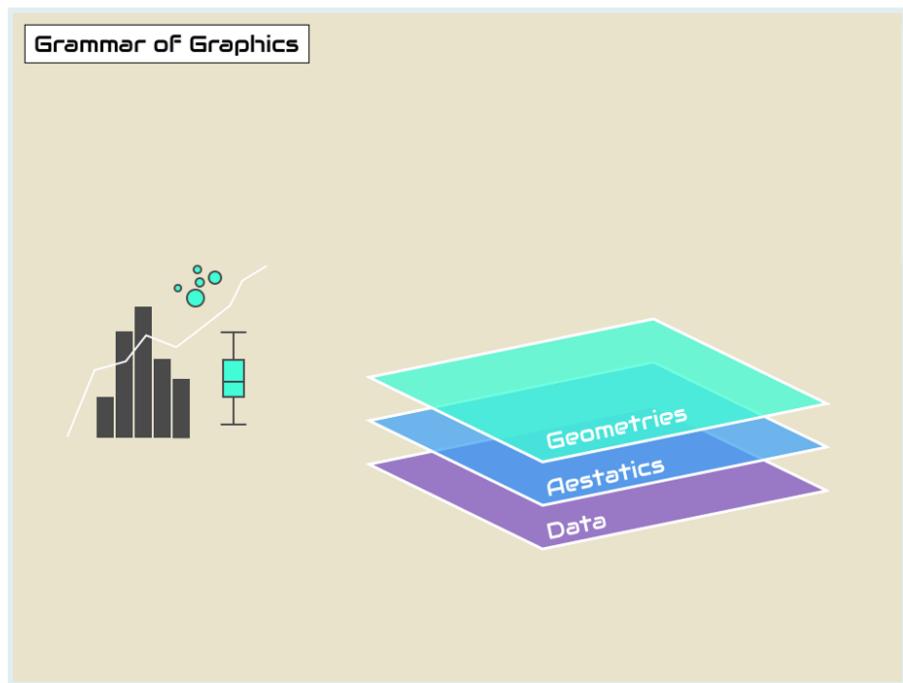
1. **data**: the dataset containing the variables of interest.



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.

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

CHALLENGE

these terms to make plots in R, you will become fluent in no time.

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
```

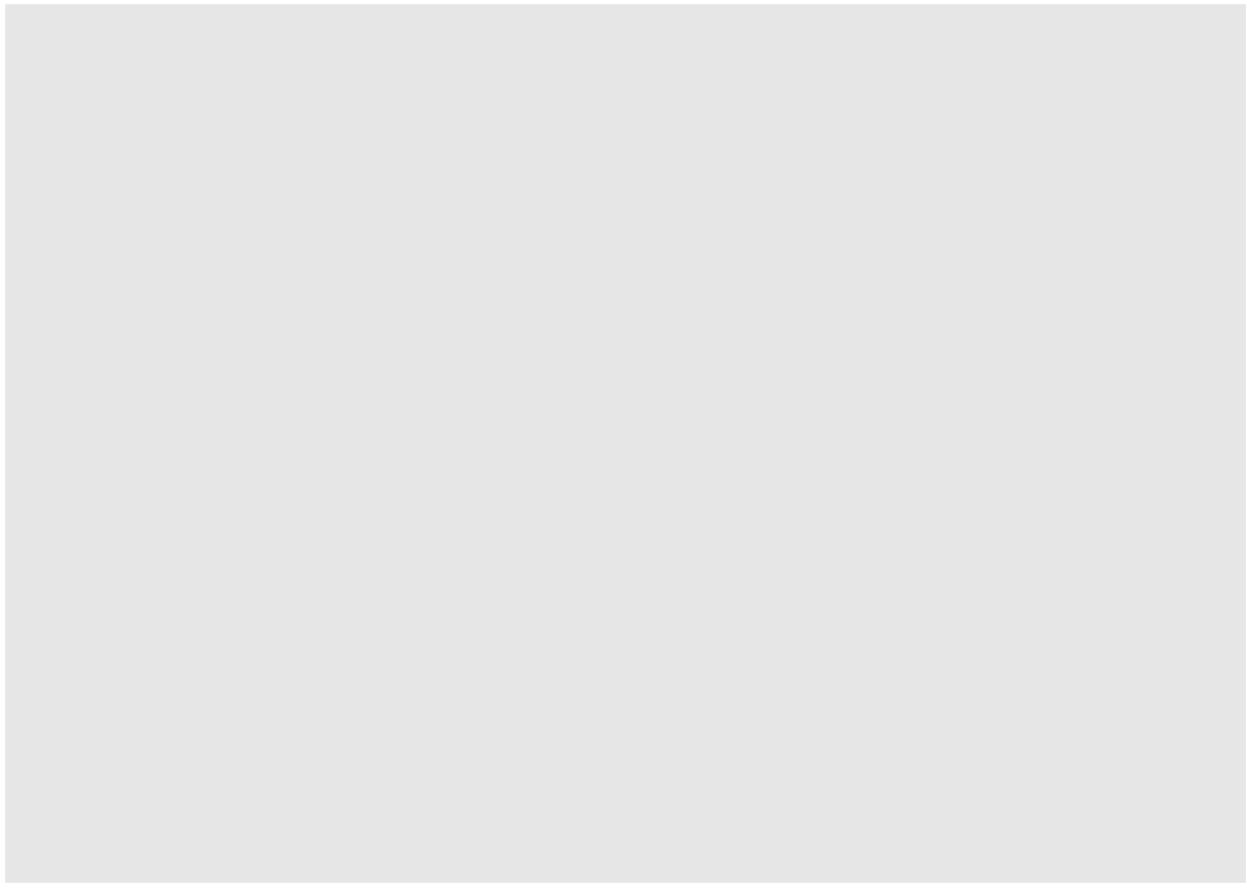
Building a `ggplot()` in steps

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()
```

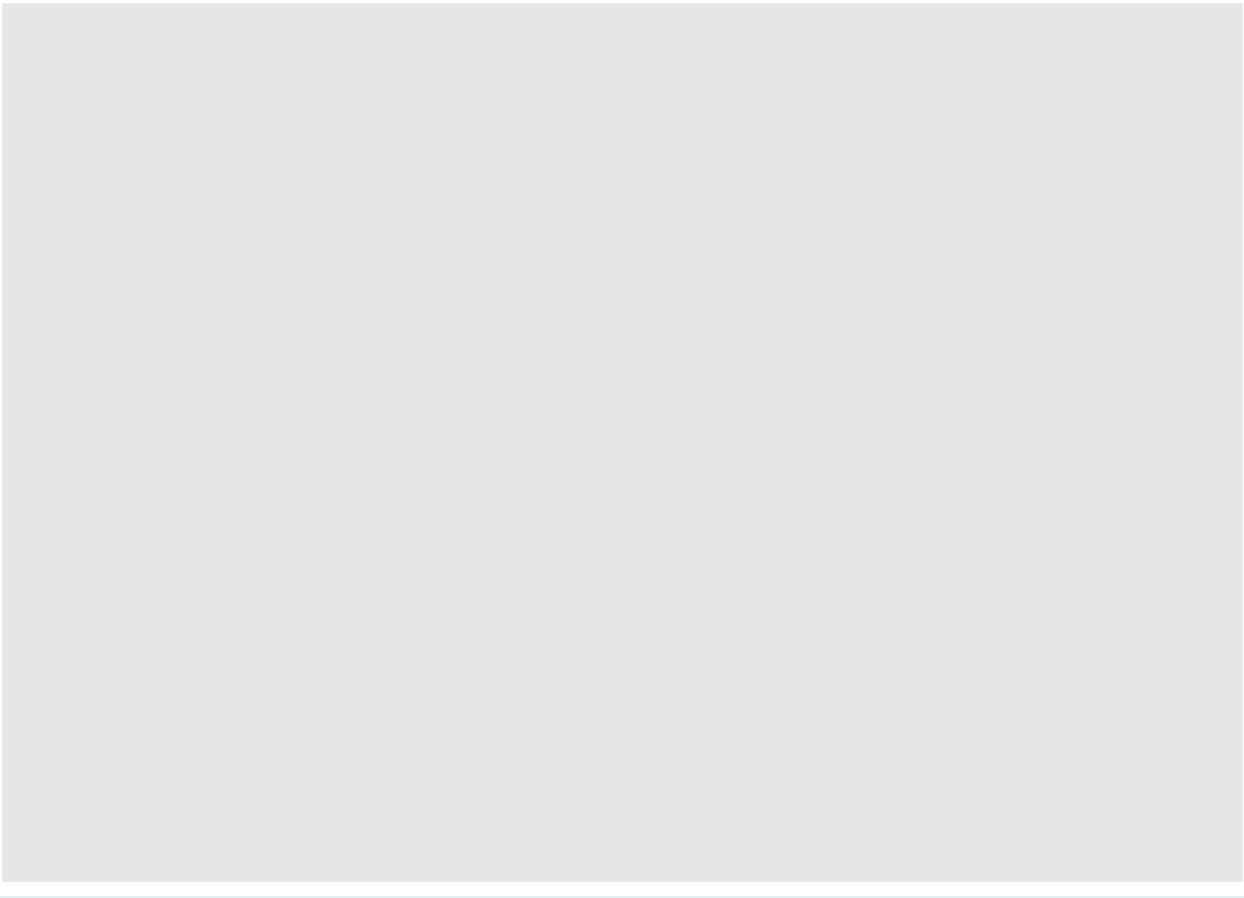


As you can see, this gives us nothing but a blank canvas. But not to worry, we're about to add some more elements.

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
```



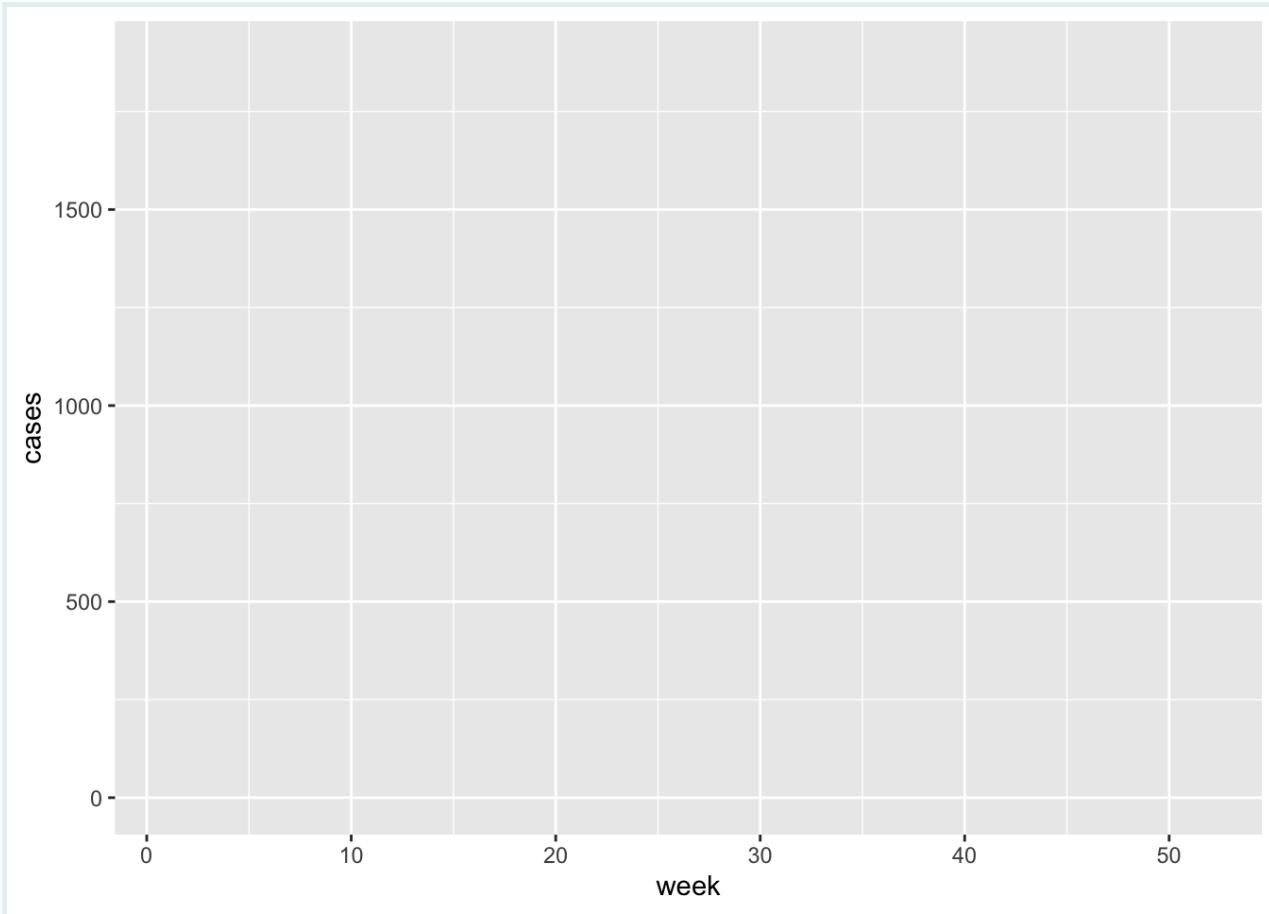
This gives us blank plot again, since we've only supplied one out of the three inputs required for a complete graphic. Next we need to assign variables to aesthetic mappings.

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.



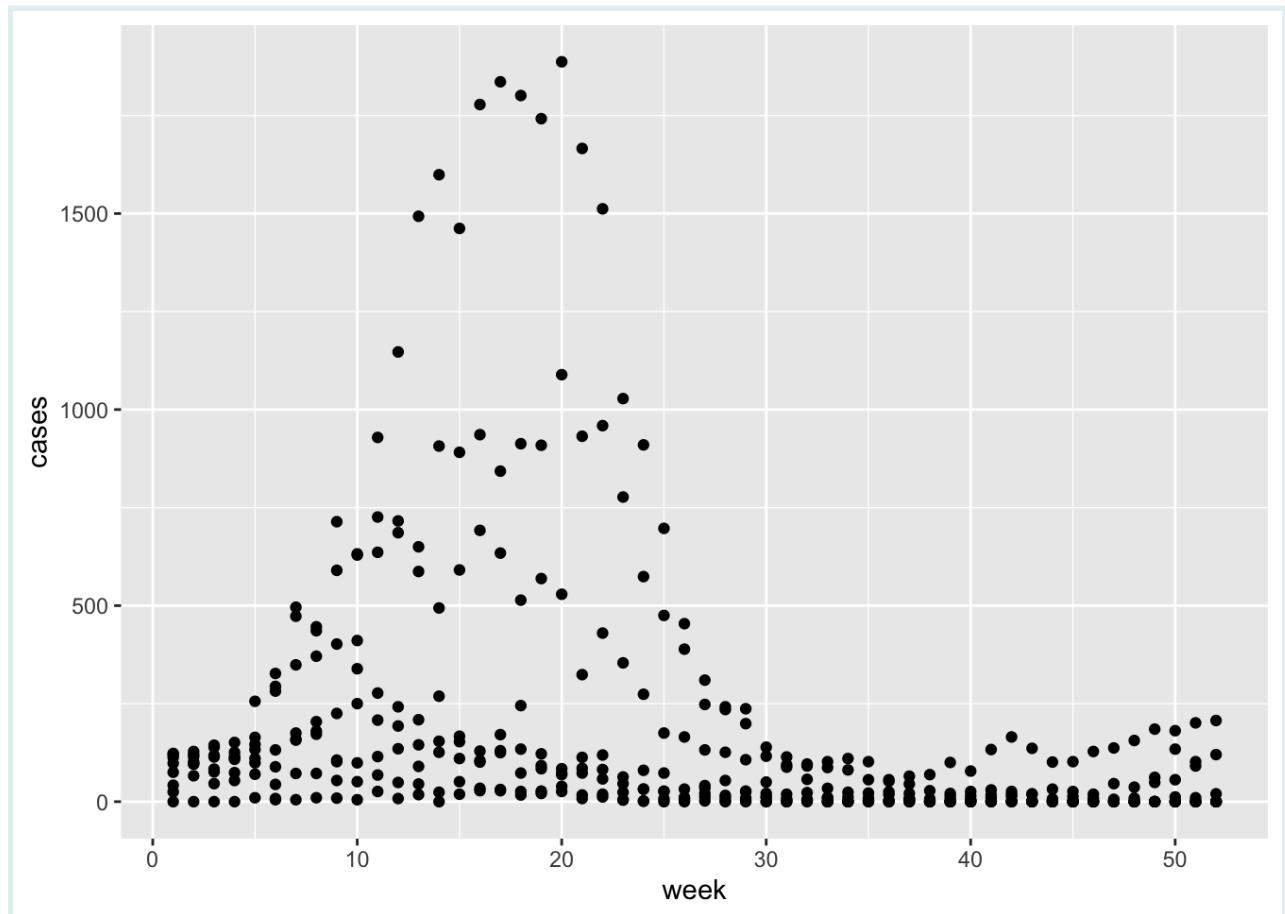
KEY POINT `aes()` stands for aesthetics - things we can see. Variables are always inside the `aes()` function, which in return is inside a `ggplot()`. Take a moment to observe the double closing brackets `))` - the first one belongs to `aes()`, the second one to `ggplot()`.

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).

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:

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.



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.

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.

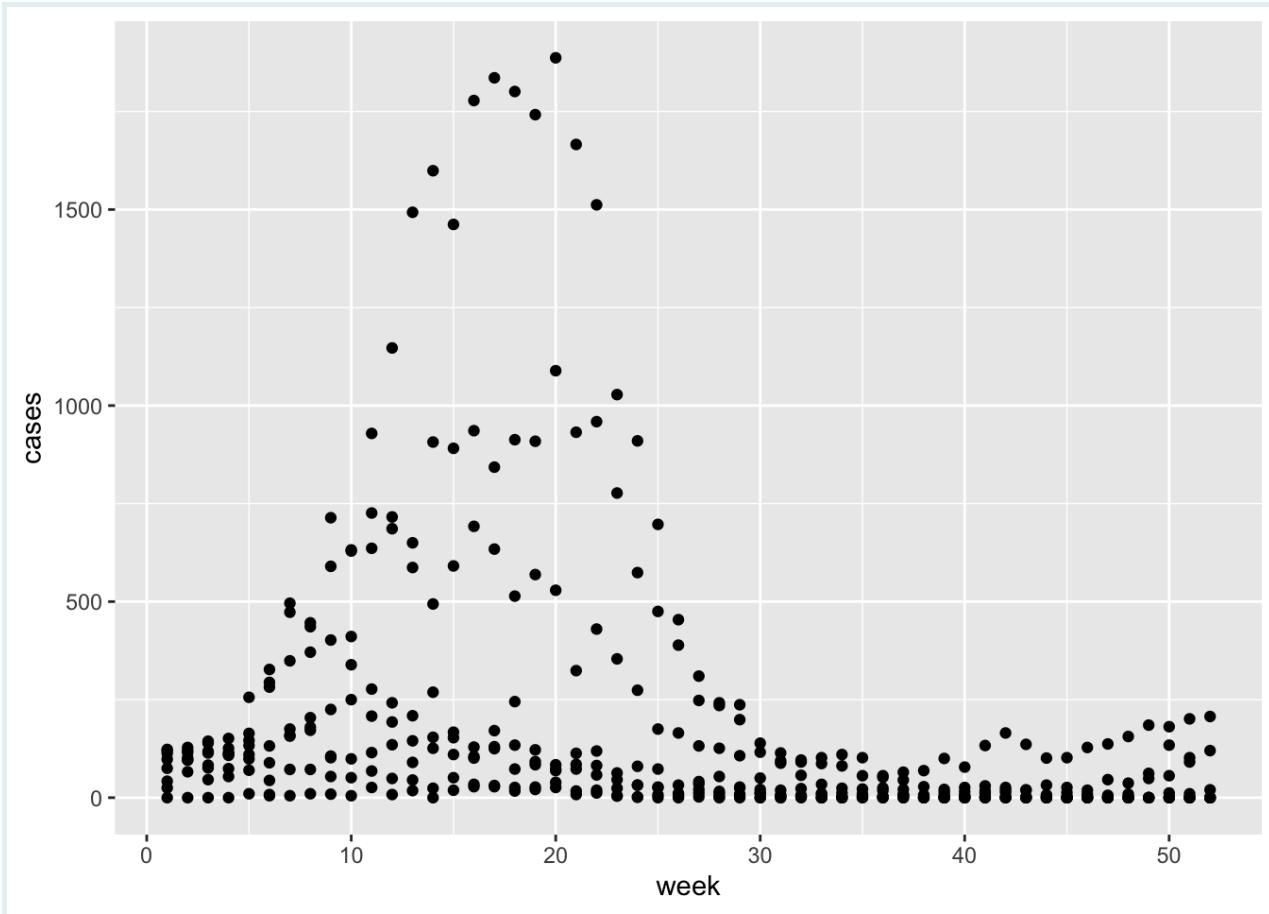
We can tinker with our existing code to switch up the data, aesthetics, and geometry inputs supplied to `ggplot()`, and create variations of the original plot. In fact, you've already done this by changing the dataset from `nigerm96` to `nigerm04` in the practice question.

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.

Changing aesthetic mappings

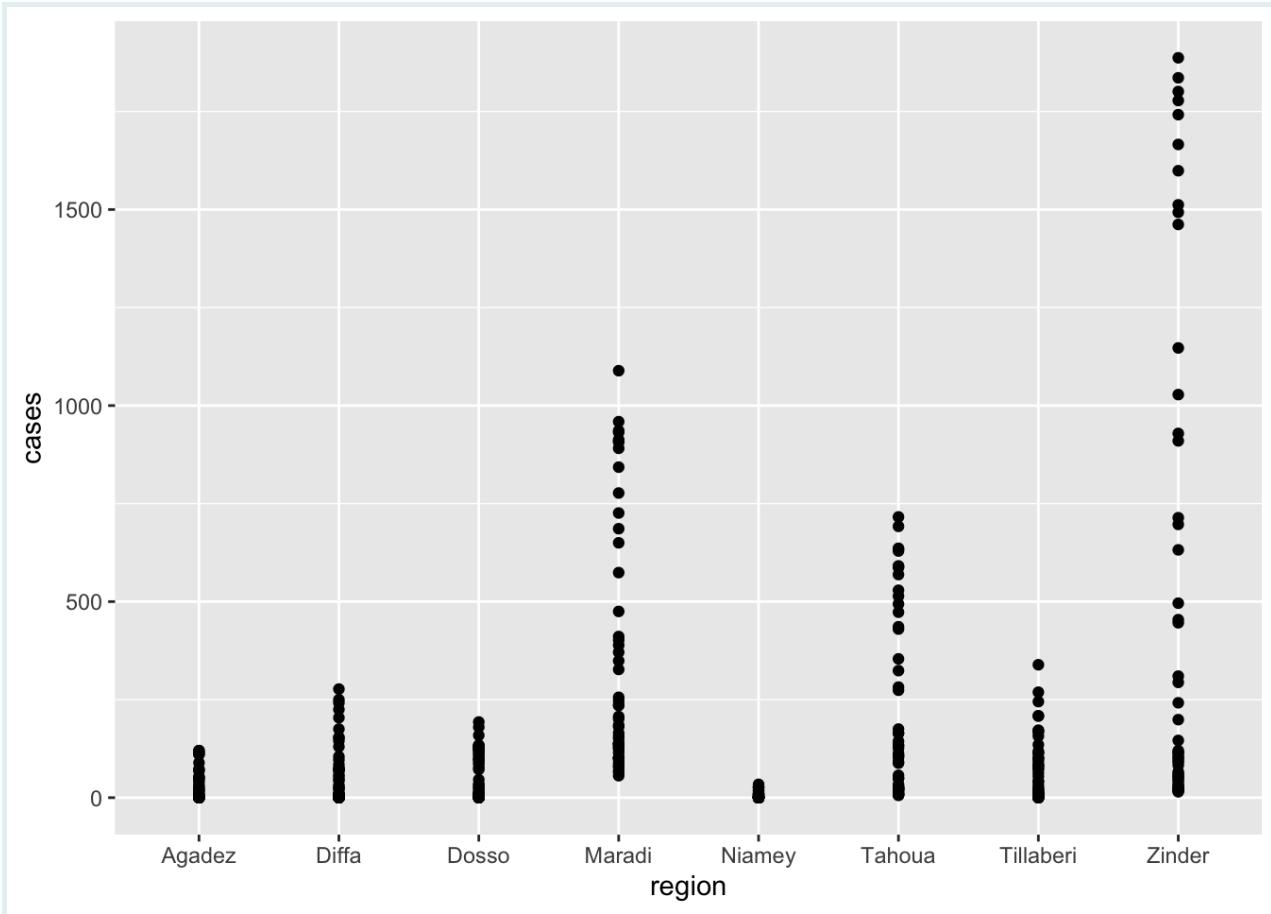
We created a scatter plot of `cases` vs `week` for `nigerm96` with this code:

```
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**:

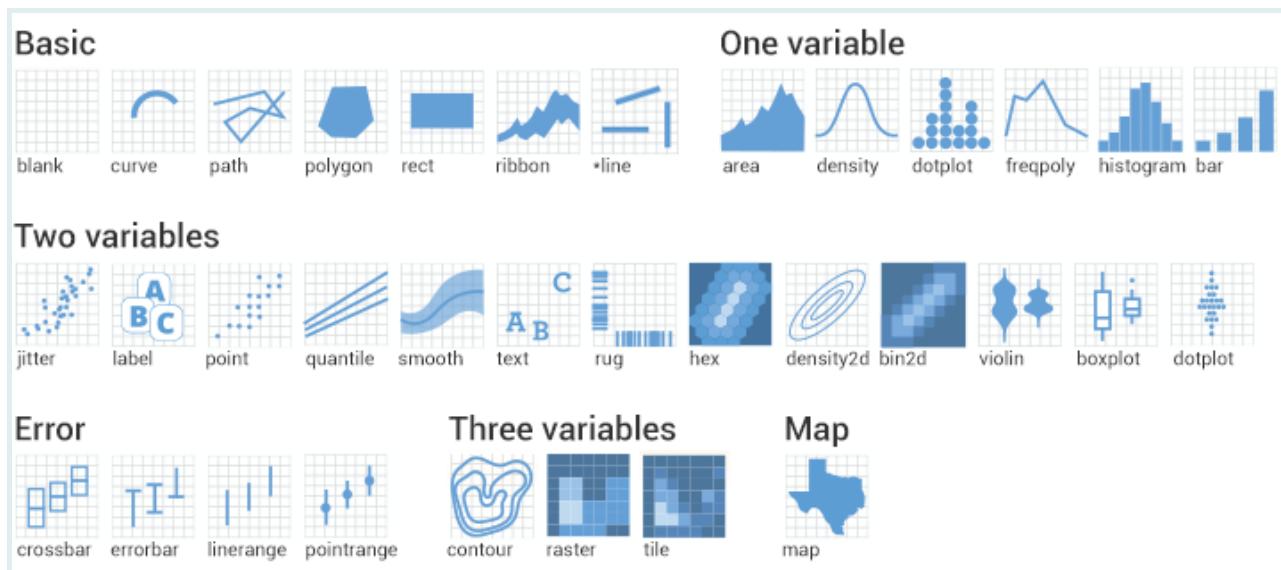
```
ggplot(data = nigerm96,
       mapping = aes(x = region, # change which variable to map on the x-
                     axis
                     y = cases)) +
  geom_point()
```



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.

Changing `geom_*` functions

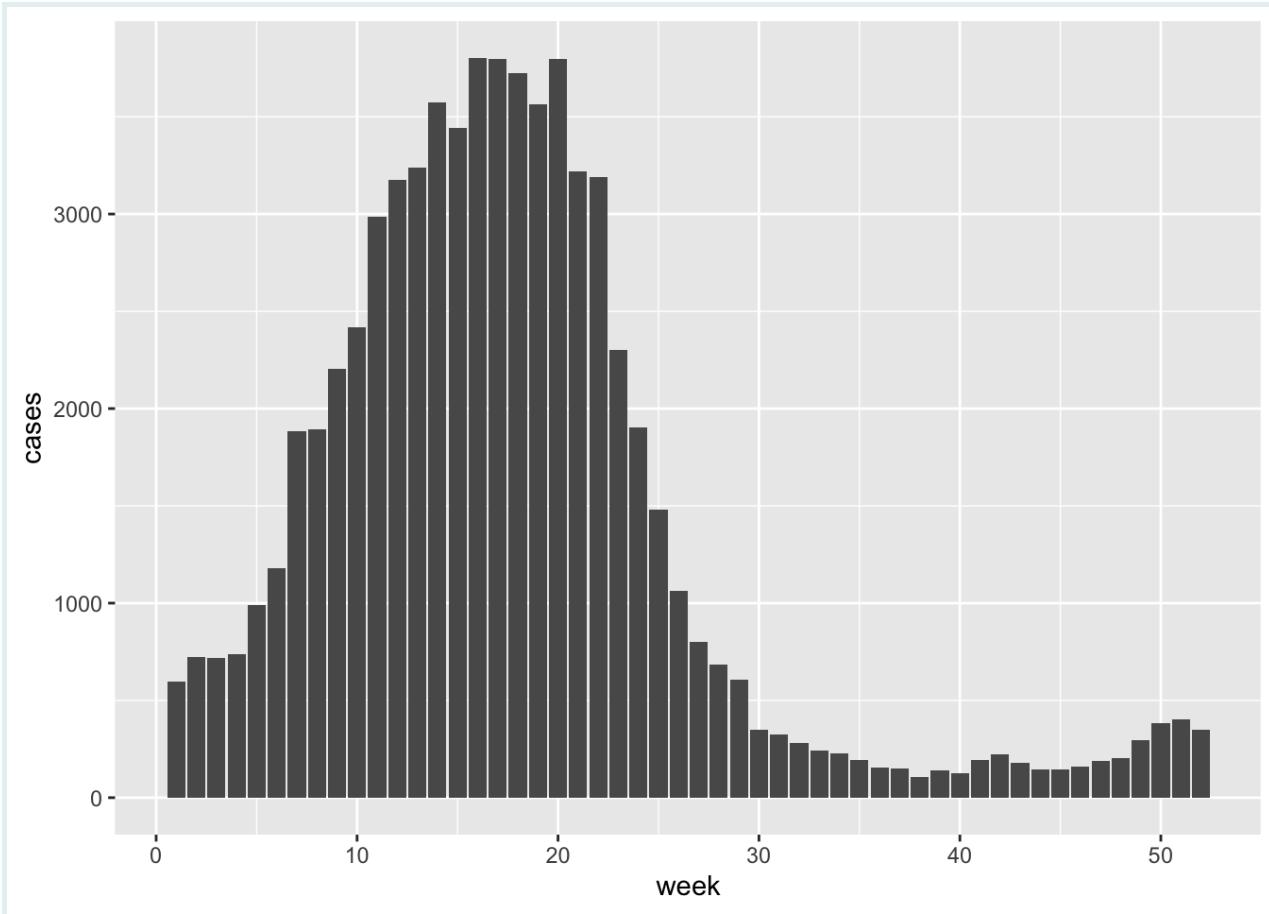
Similarly, we can modify the geometry layer to create a different type of plot, while still using the same aesthetic mappings.



{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()`.

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
```



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 than 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.



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

Error?



positions with a histogram, so it throws an error.

PRACTICE

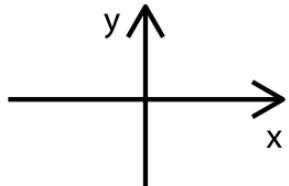


Use the `nigerm04` data frame to create a bar plot of weekly cases with the `geom_col()` function. Map `cases` on the y-axis and `week` on the x-axis.

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.

position



shape



size



color



line width



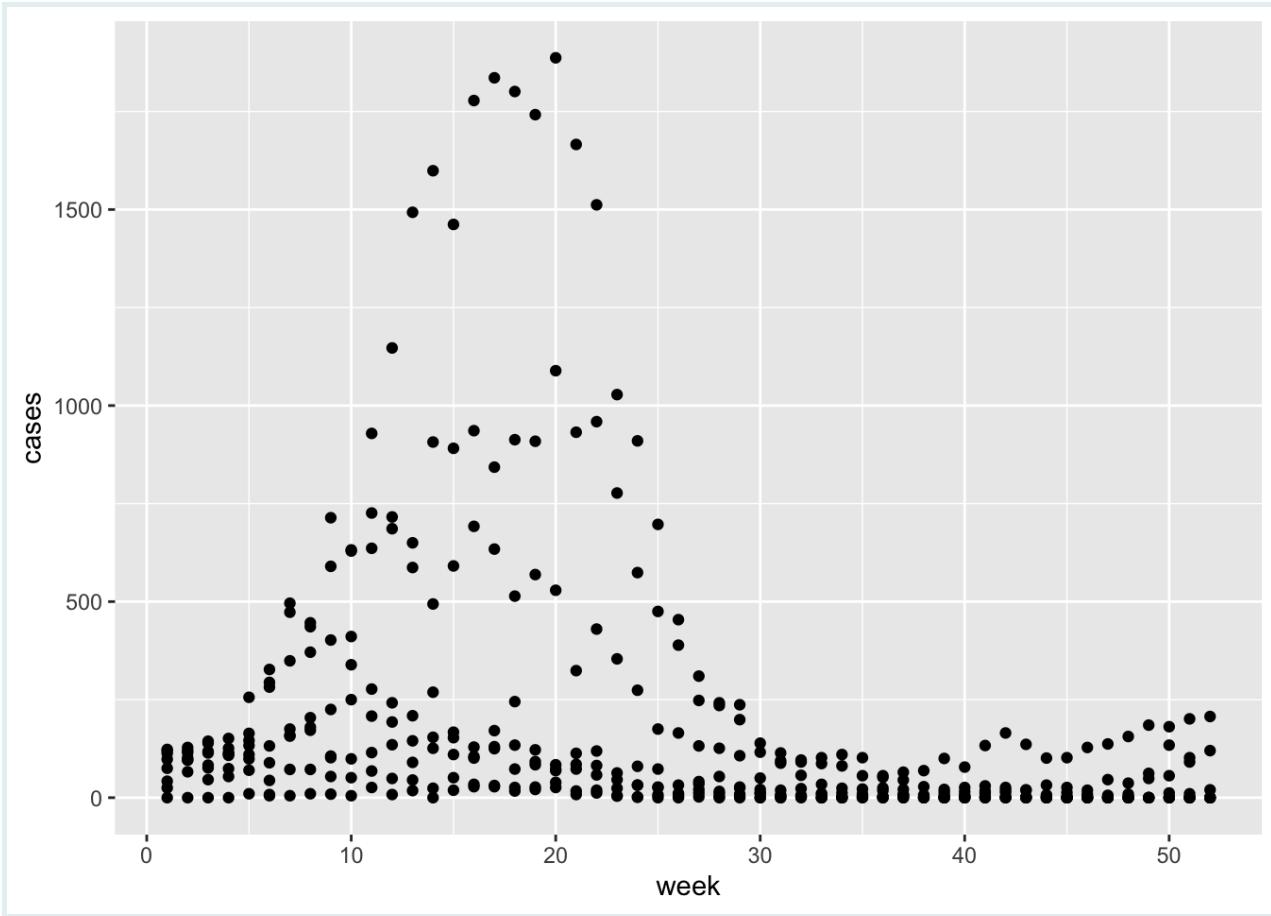
line type



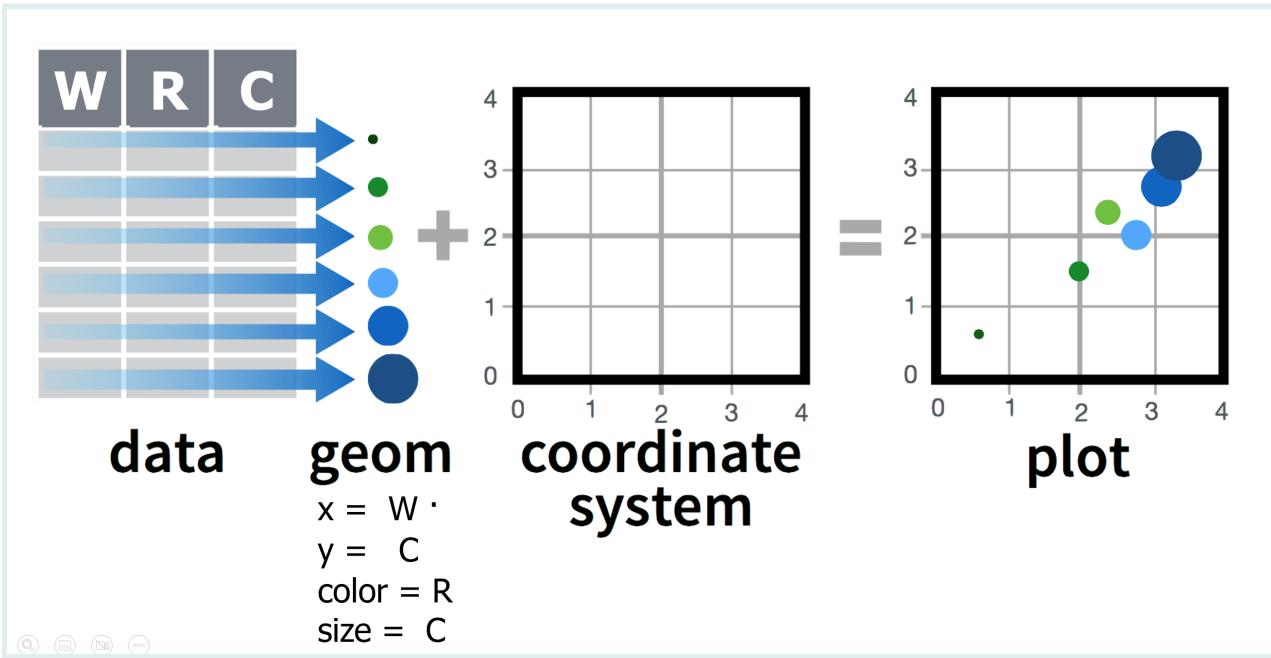
Common aesthetic attributes used in `ggplot` graphics.

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.

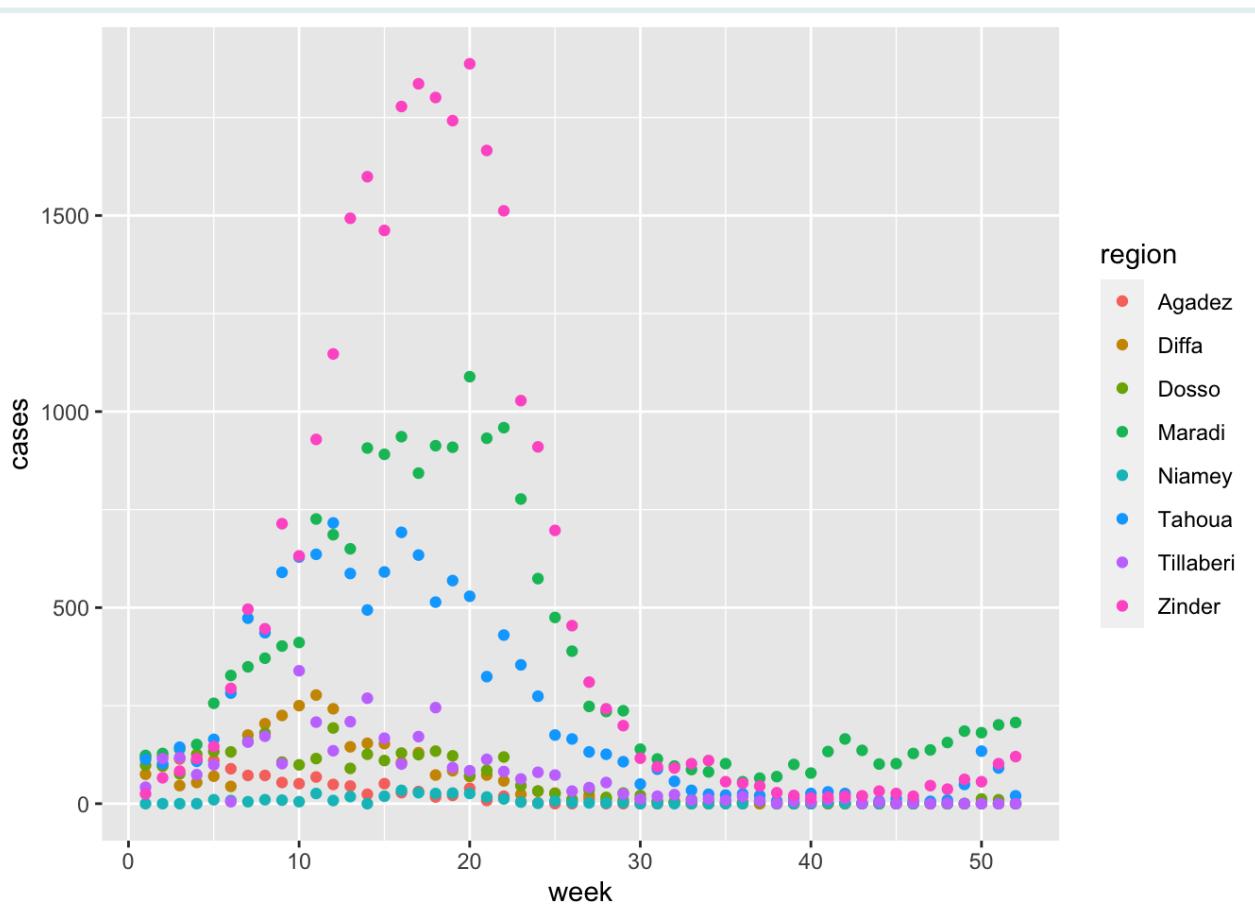




To see the full list of aesthetics that can be used with a particular `geom_*` function look it up the function documentation. You can do this by pressing F1 on a function, e.g., `geom_point()` to open the Help tab, and scroll down to the “Aesthetics” section. If F1 is hard to summon on your keyboard, 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.

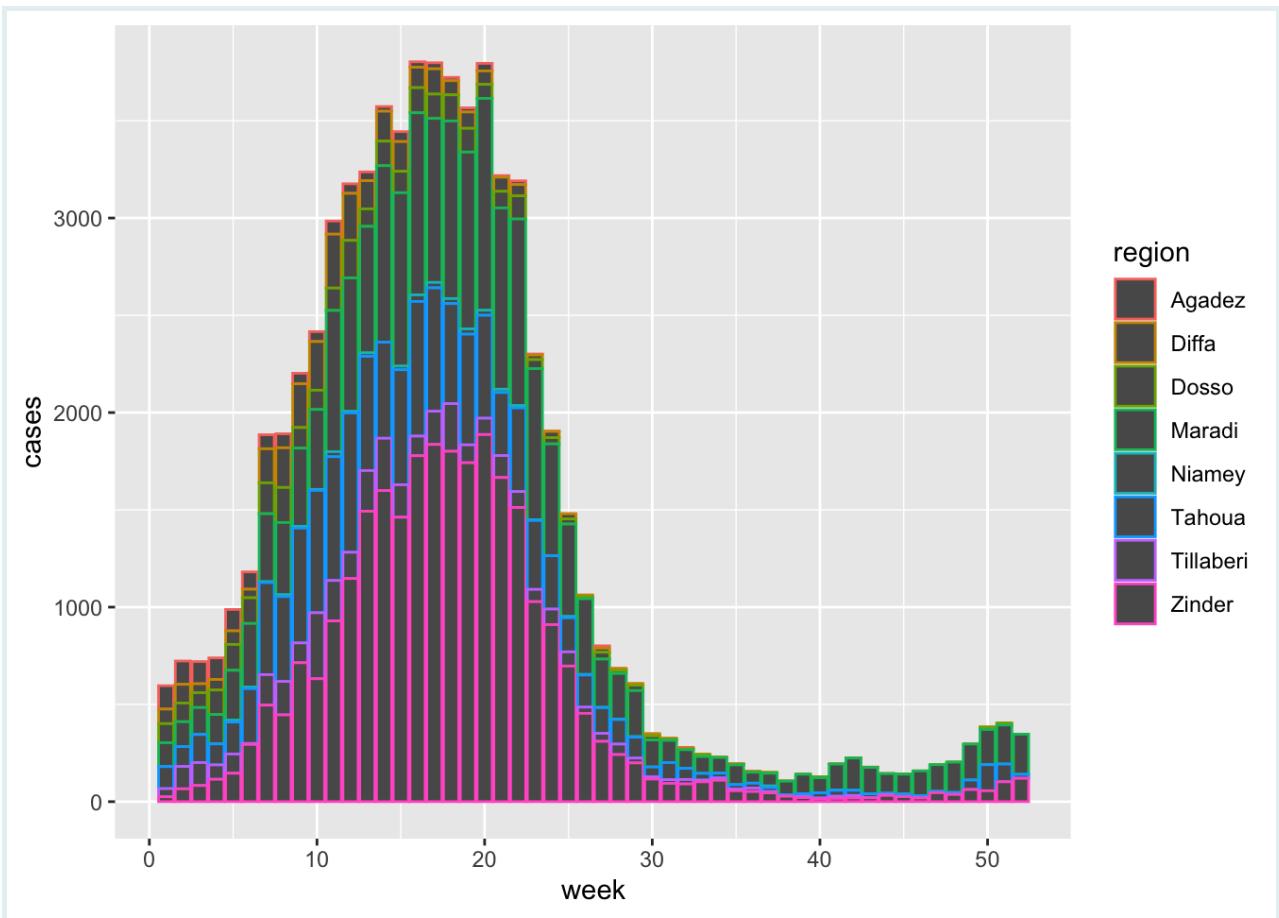
Zinder had the largest number of cases and the steepest epidemic curve, followed by Maradi and Niamey.

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
        geom_col()
```

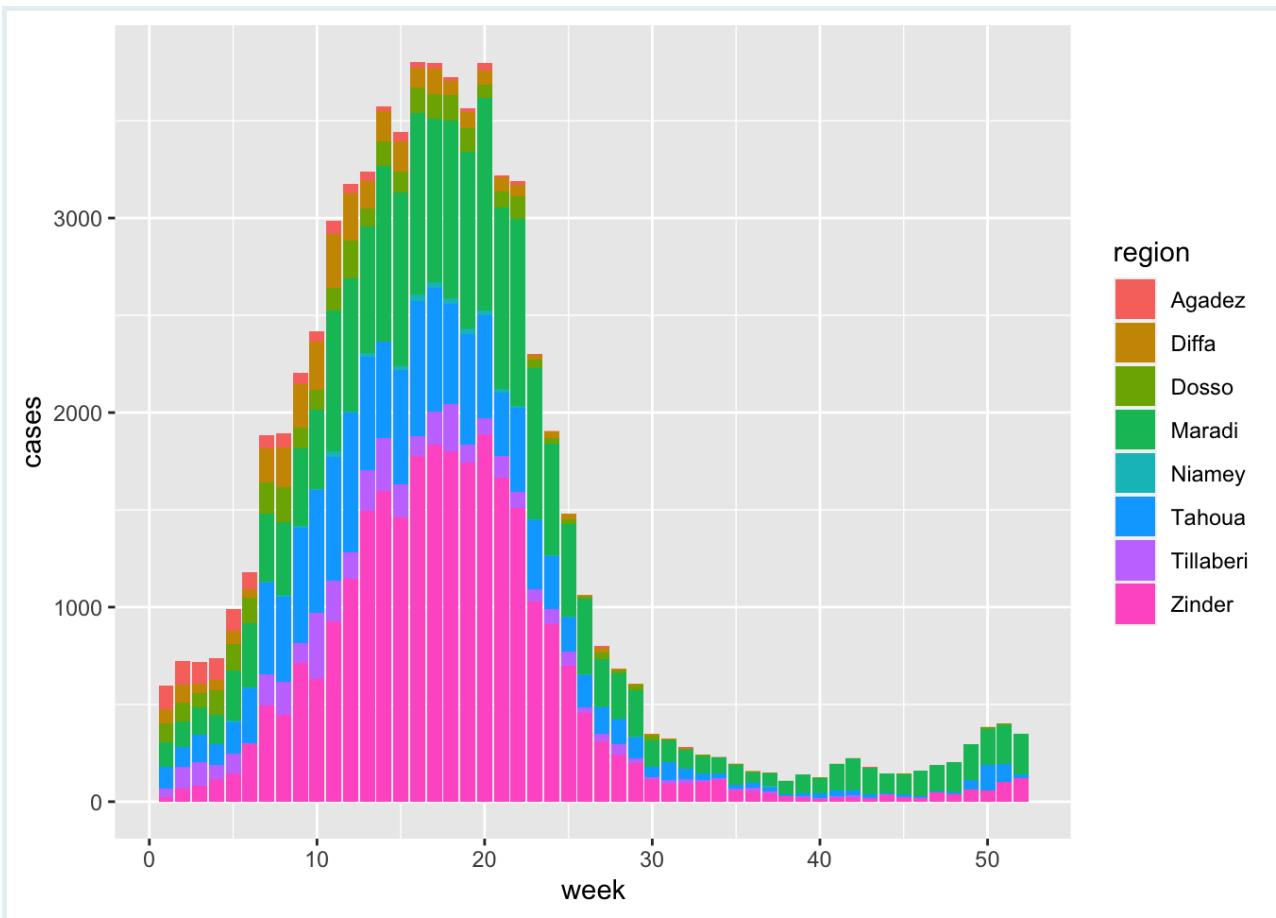


This gives us a stacked bar plot, where the bars are divided into smaller sections. This shows us the proportional contribution of individual regions (i.e., the height or length of each subsection represents how much each region contributes to the total number of cases that week).

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 = nigerm96,
       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()`.

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.

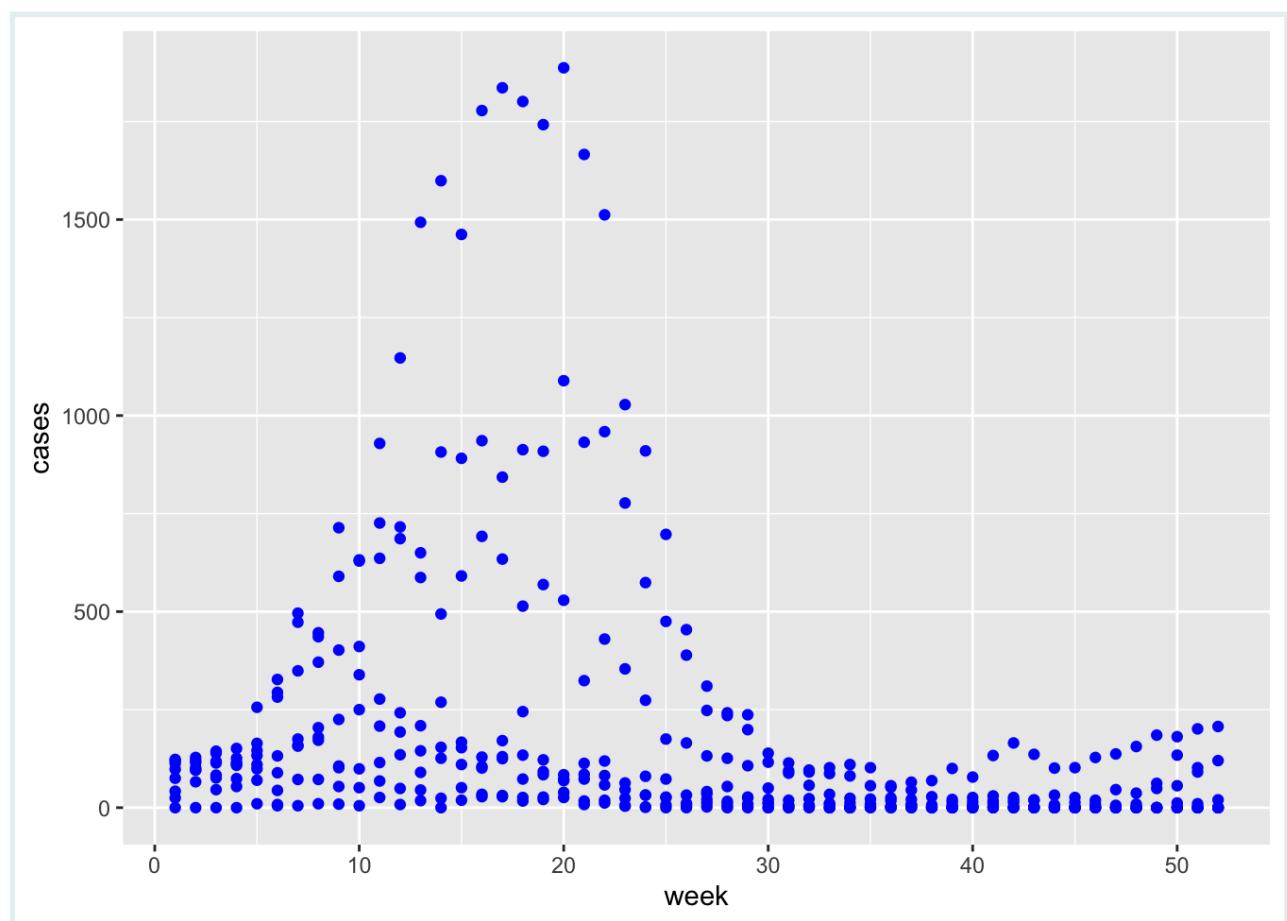
When we apply an aesthetic to modify the geometric objects according to a variable (e.g., the color of points changes according to the `region` variable), that's an aesthetic

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.

SIDE NOTE

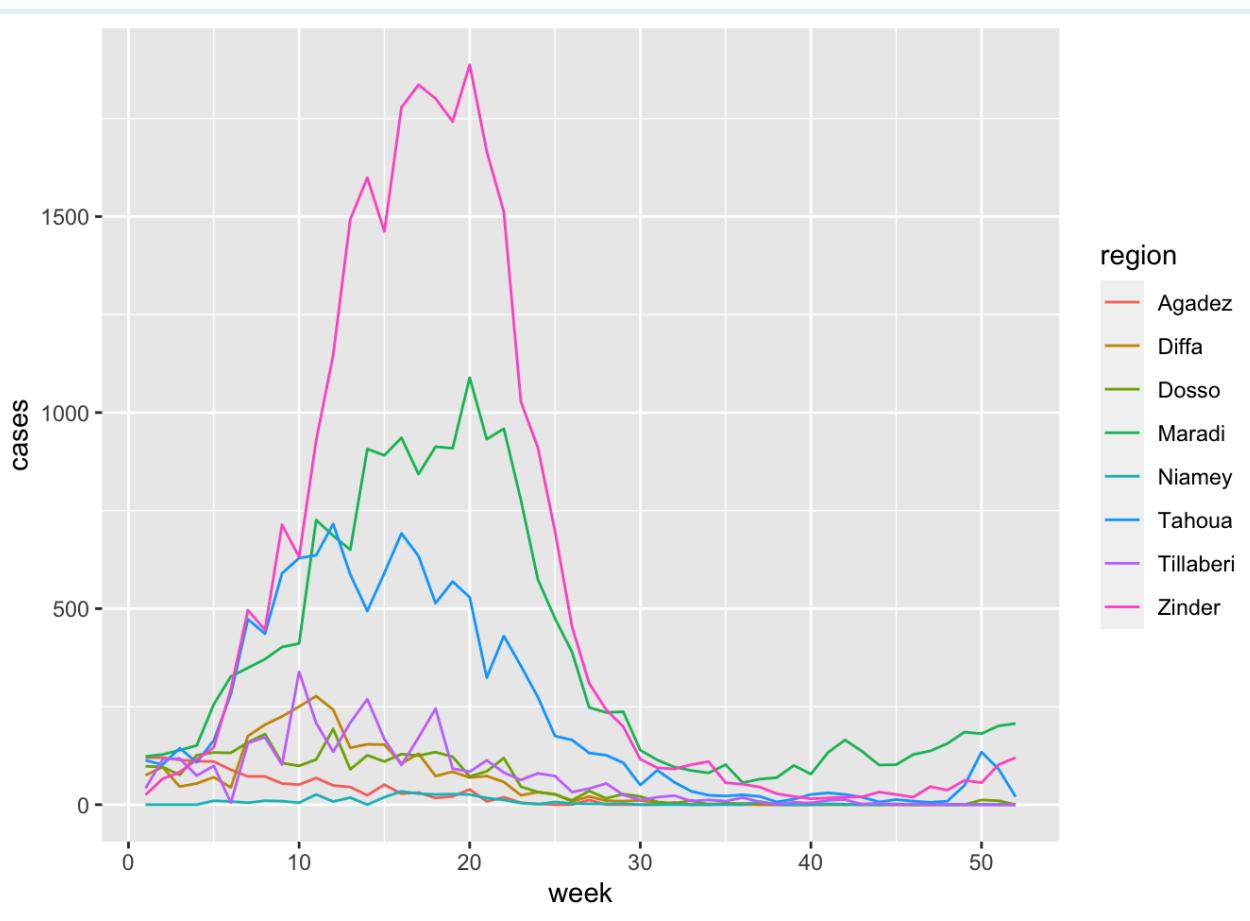


SIDE NOTE

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()`.

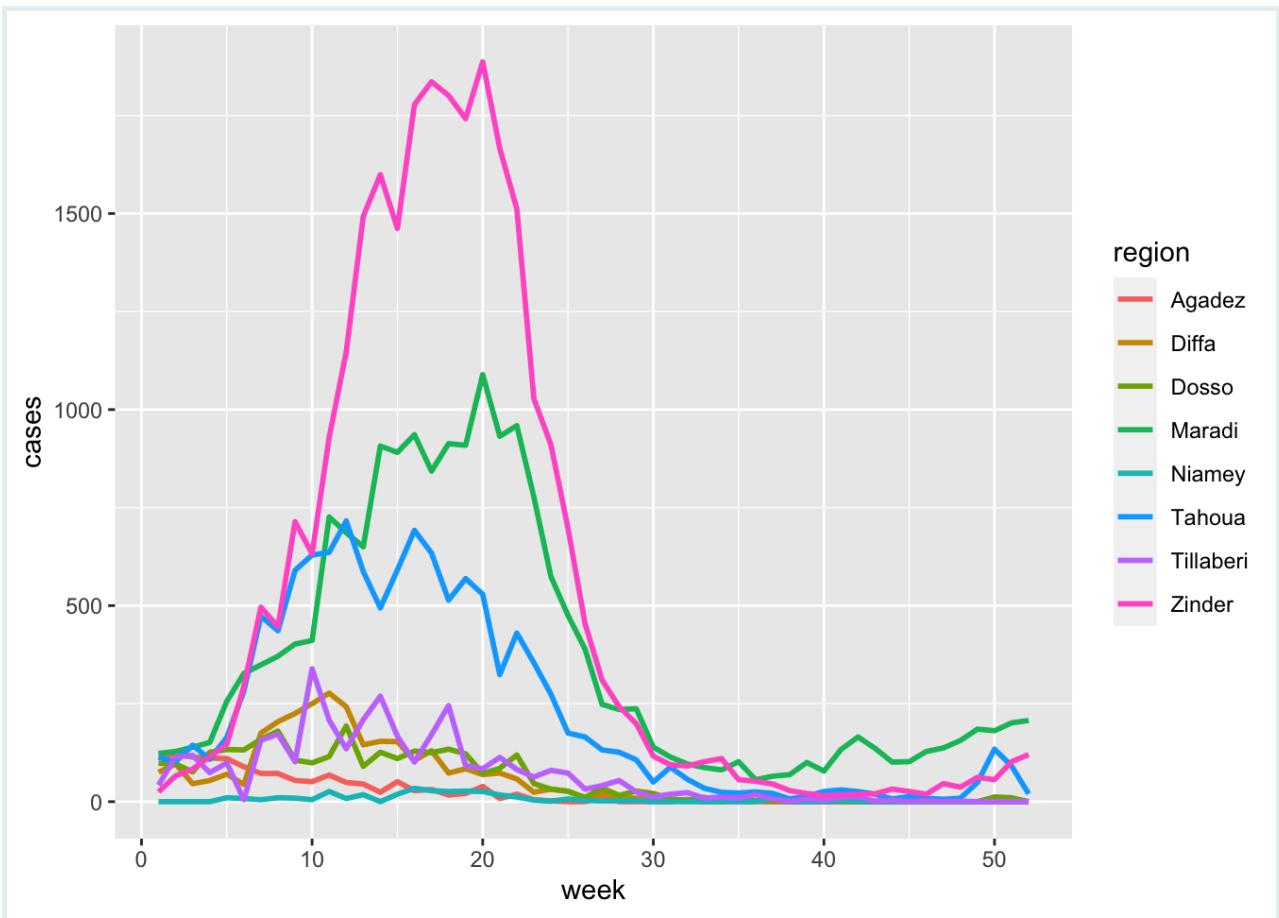
Now let's add a fixed aesthetic called `size`. The default line width used by `geom_line()` is 0.5 mm, which looks like this:

```
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 = nigerm96,
       mapping = aes(x = week,
                      y = cases,
                      color = region)) +
  geom_line(size = 1)
```



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.

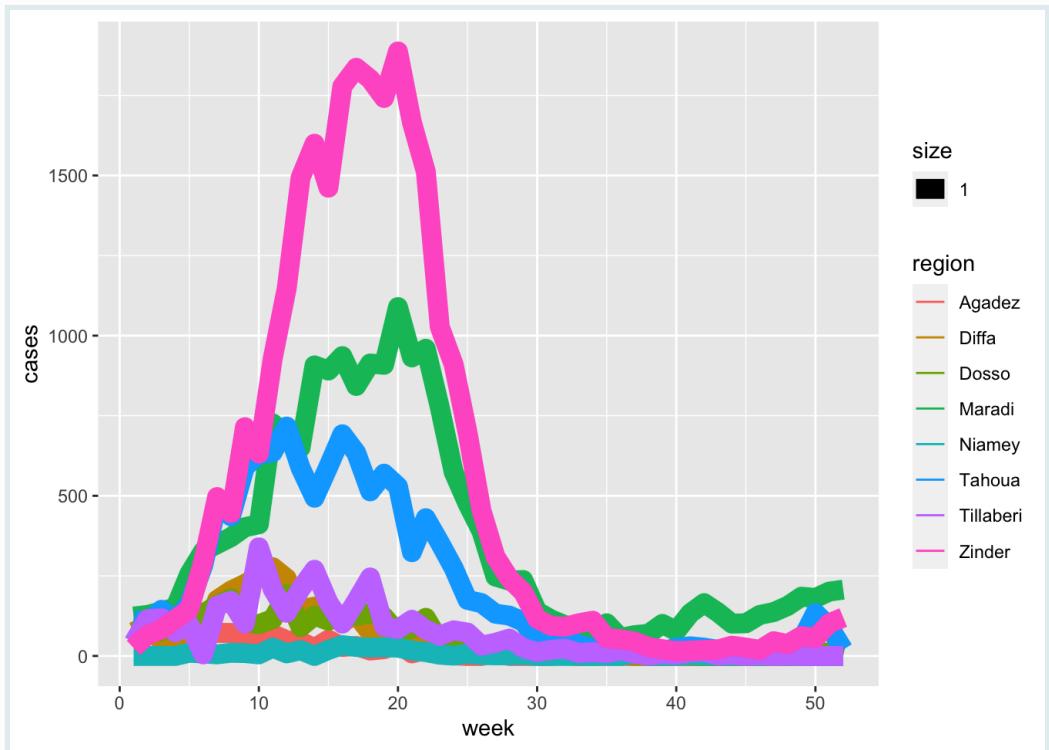
WATCH OUT



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 = nigerm96,
       mapping = aes(x = week,
                     y = cases,
                     color = region,
                     size = 1)) +      # INCORRECT
  placement
  geom_line()
```

WATCH OUT



`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.



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".

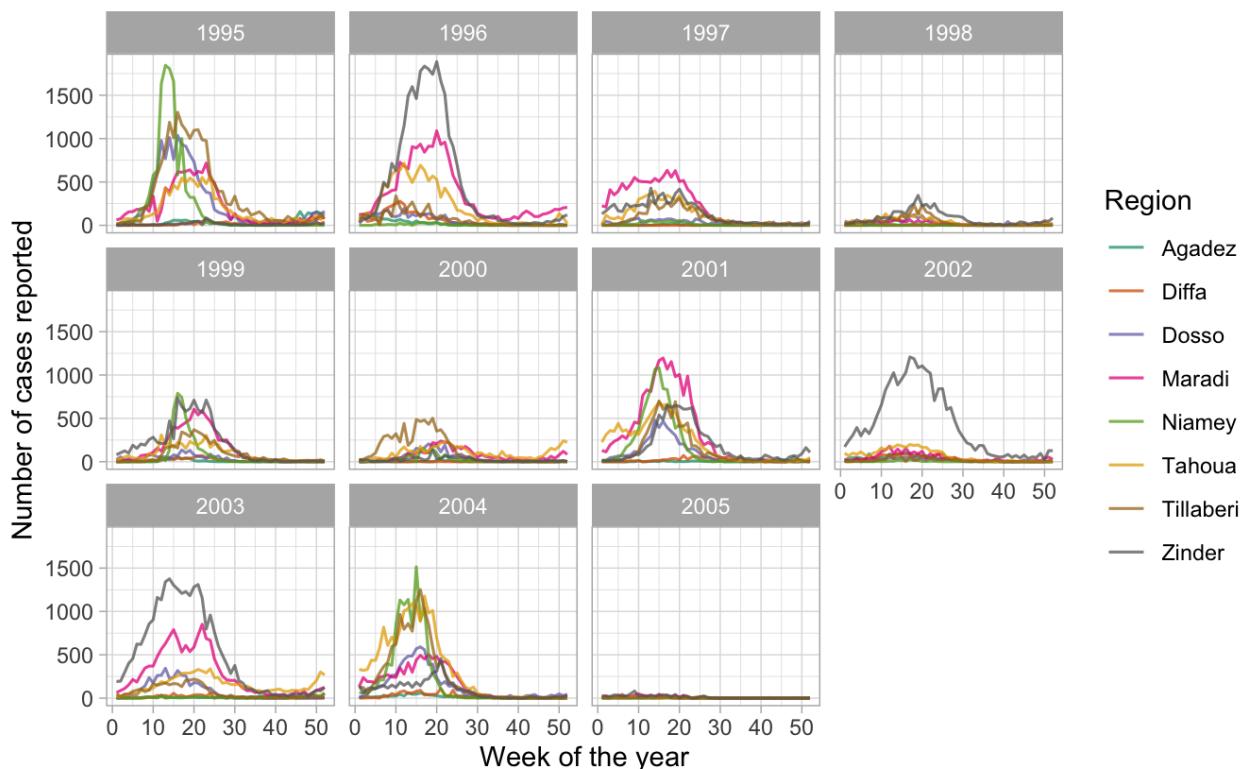
Additional GG layers

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:

Seasonal patterns of measles incidence in Niger

Weekly reported at region level (1995-2005)



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()
```

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**.

Contributors

The following team members contributed to this lesson:



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science

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 Granados. 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://argosshare.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>.

This work is licensed under the Creative Commons Attribution Share Alike license.



Lesson notes | Scatter plots and smoothing lines

Created by the GRAPH Courses team

January 2023

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

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

Introduction
Learning Objectives
Childhood diarrheal diseases in Mali
Scatter plots via <code>geom_point()</code>
Aesthetic modifications
Mapping data to aesthetics
Setting fixed aesthetics
Adding a trend line
Summary

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!

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()`.

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](#).



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.

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

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:

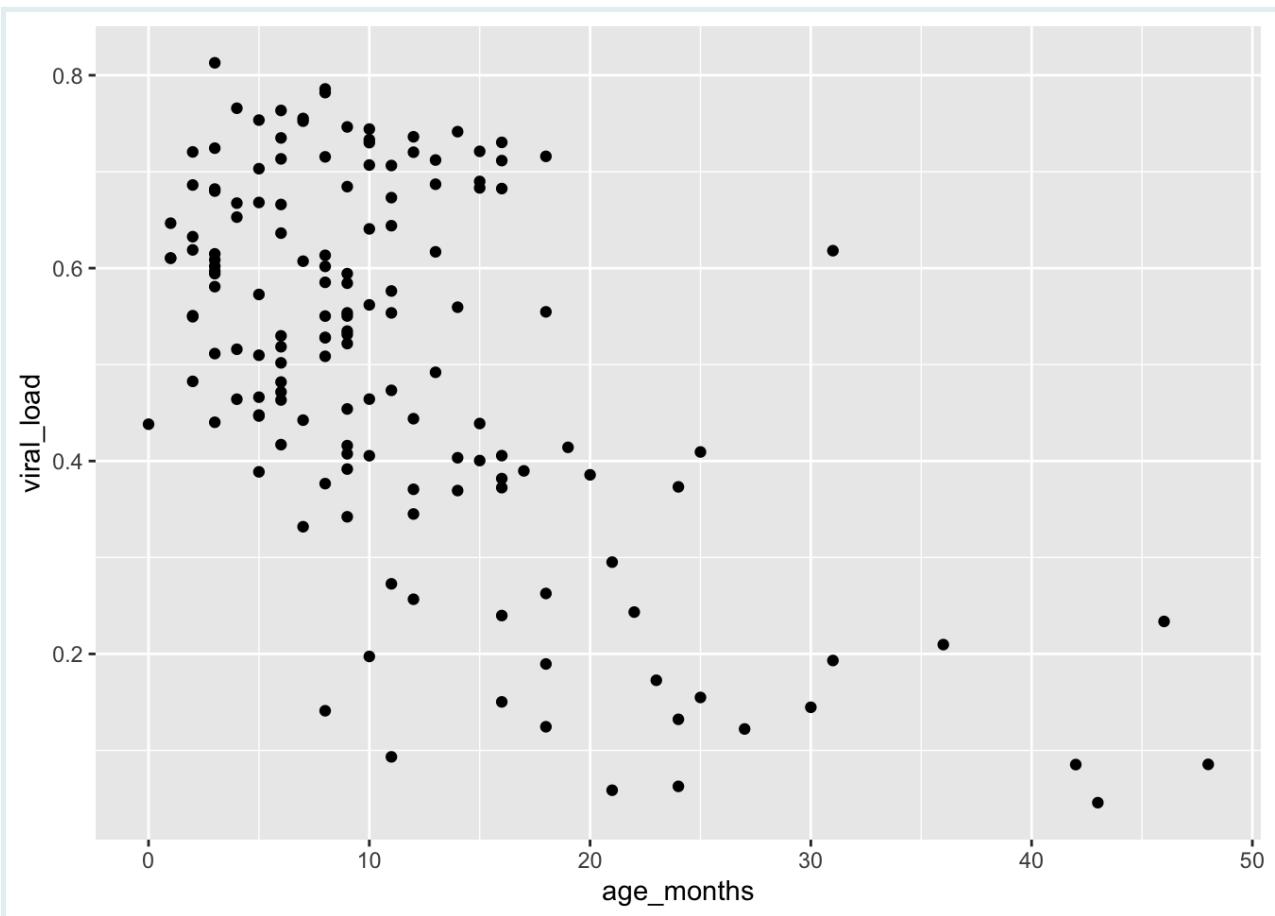
1. We provide the `malidd` data frame with the `data` argument, by inputting `data = malidd`.
2. We define the variables to be plotted in the `aesthetics` function of the `mapping` argument, by inputting `mapping = aes(x = age_months, y = viral_load)`. Specifically, the variable `age_months` is mapped 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.



- Using the `malidd` data frame, create a scatter plot showing the relationship between age and height (`height_cm`).

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()`.

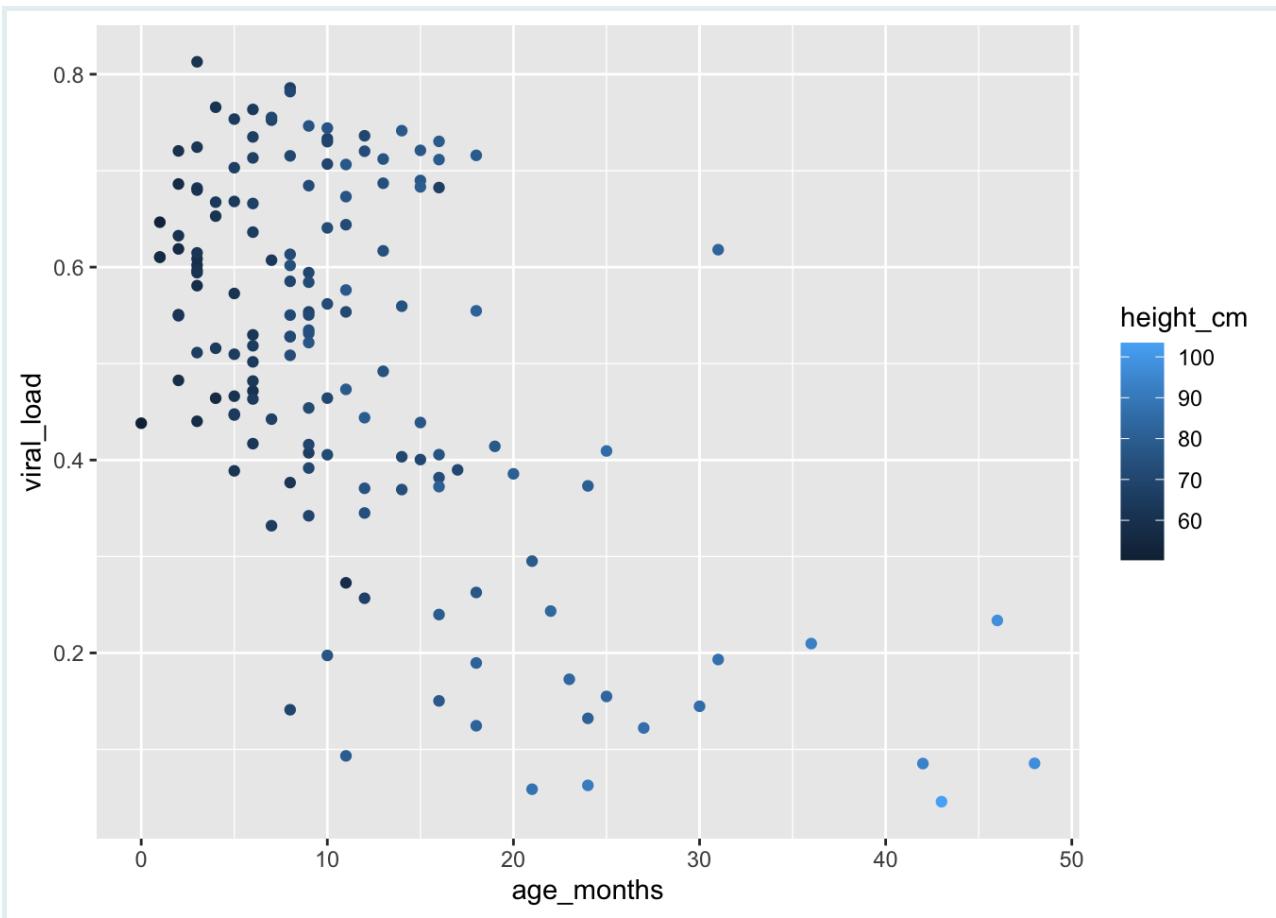
Mapping data to aesthetics

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))
```



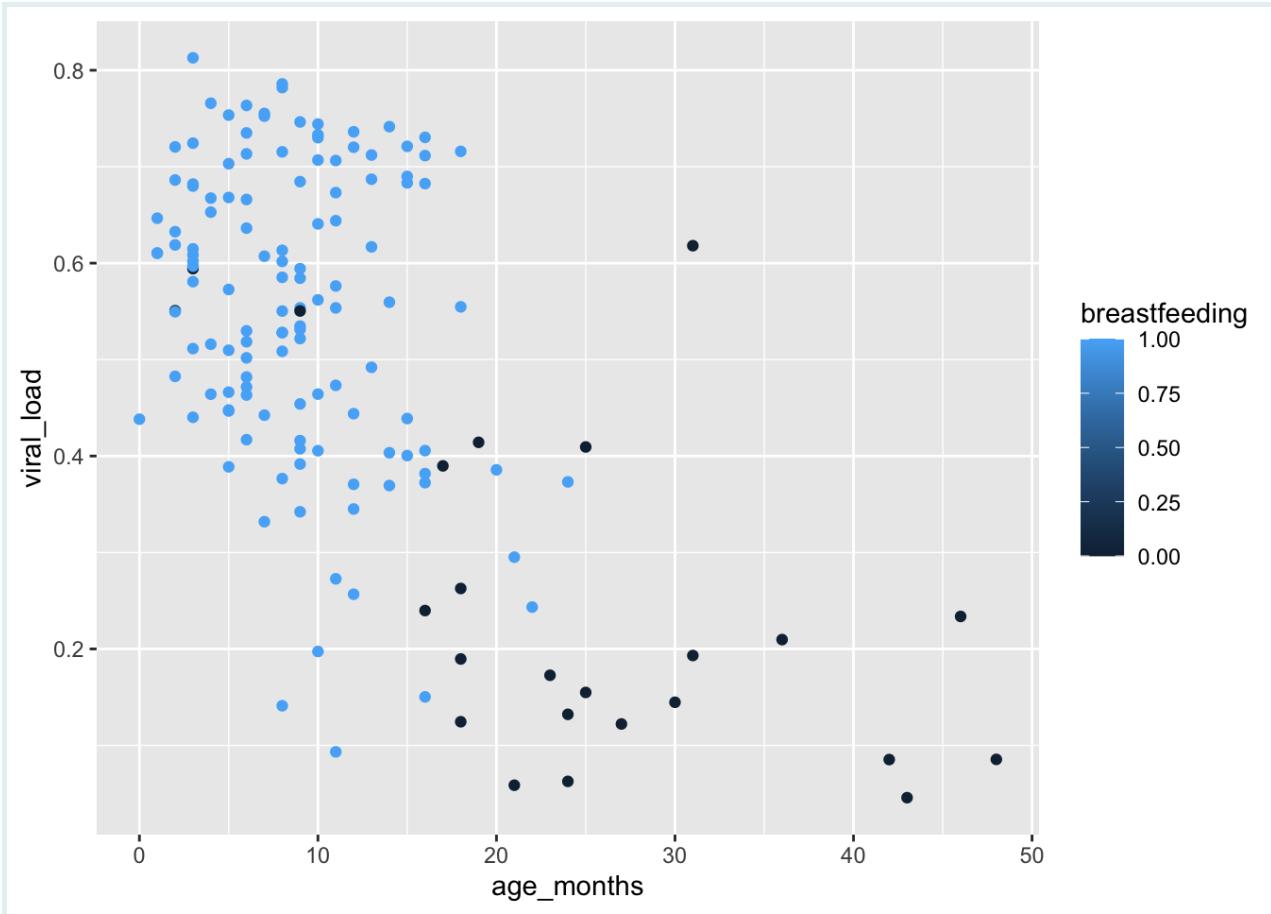
We see that `{ggplot2}` has automatically assigned the values of our variable to an aesthetic, a process known as **scaling**. `{ggplot2}` will also add a legend that explains which levels correspond to which values.

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`:

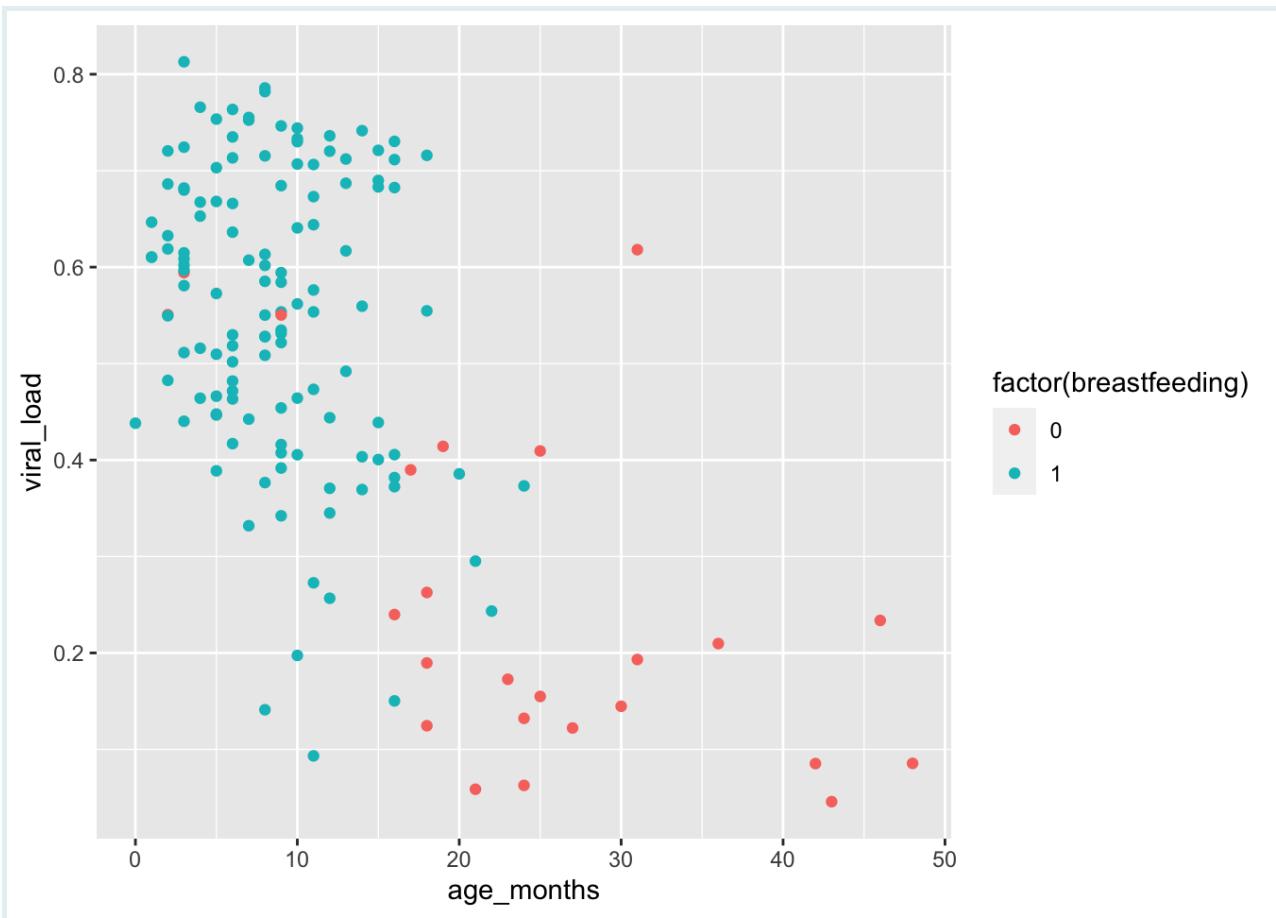
```
class(malidd$breastfeeding)
```

```
## [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 `nigeria` data frame 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.

- Using the `malidd` data frame, create a scatter plot showing the relationship between age and viral load, and map a third variable, `frequrespi`, to color:





```
# Type and view your answer:  
age_height_fever <- "YOUR ANSWER HERE"  
age_height_fever
```

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:

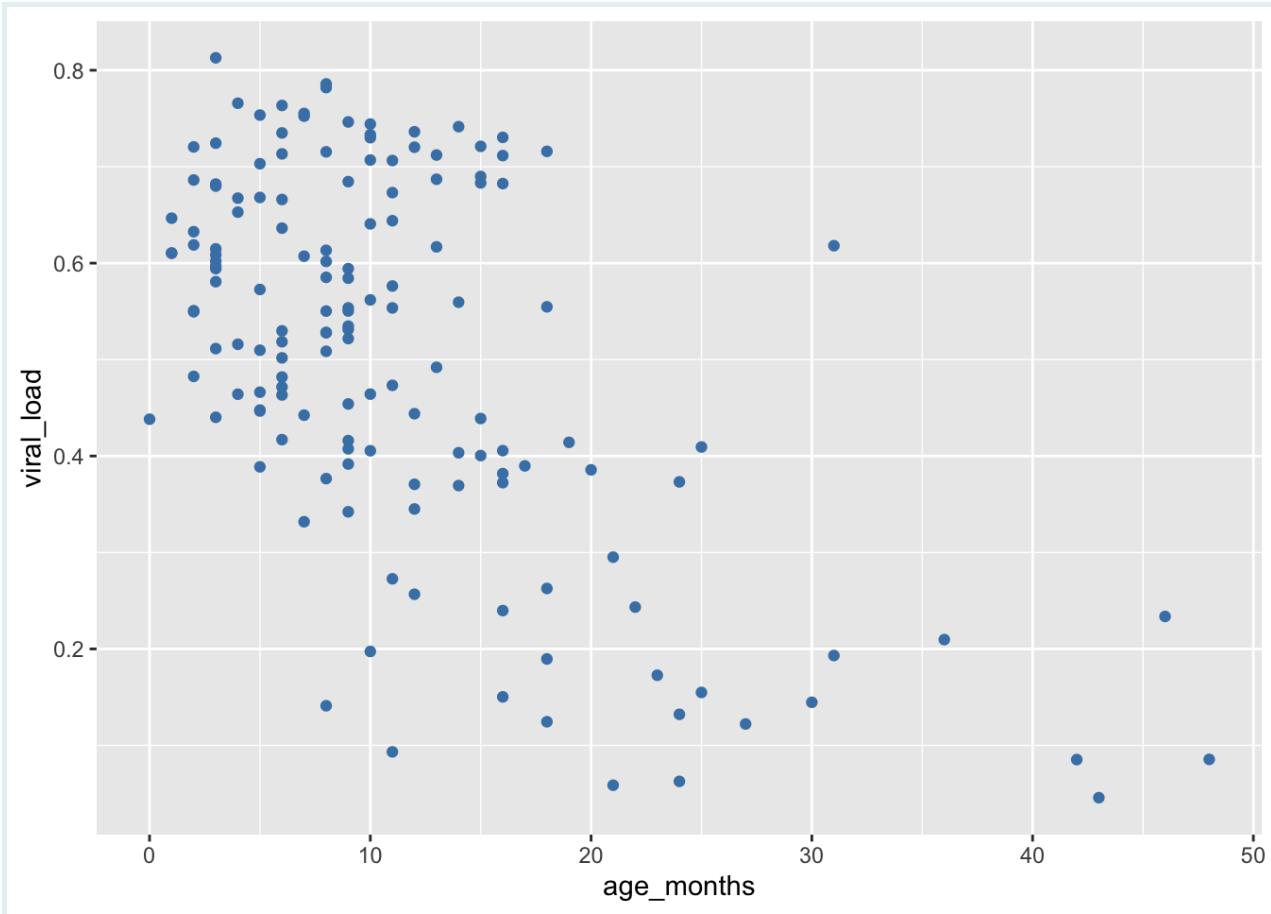
- `color` - point color or point outline color
- `size` - point size
- `alpha` - point opacity
- `shape` - point shape
- `fill` - point fill color (only applies if the point has an outline)

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.

Changing `color`, `size` and `alpha`

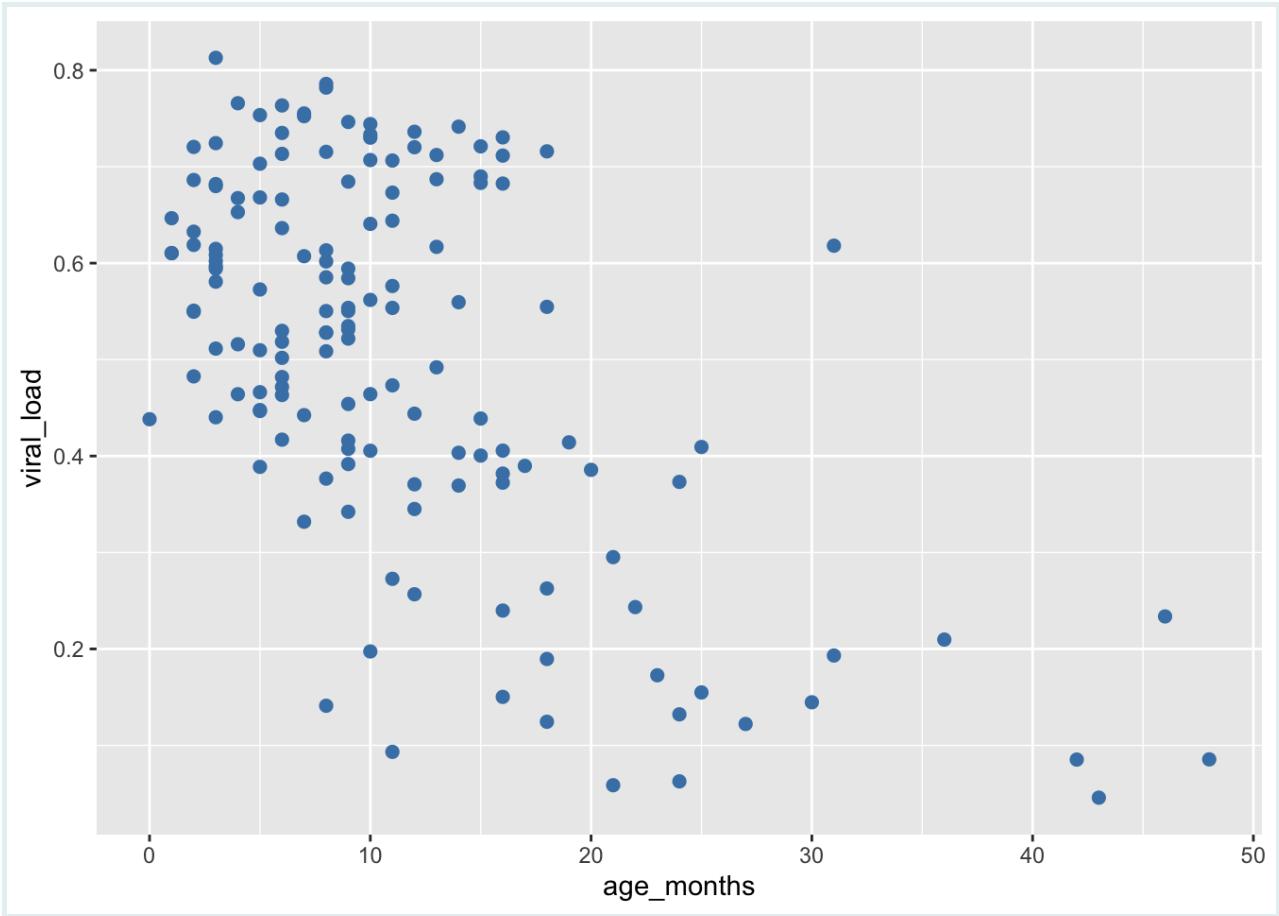
Let's change the color of the points to a fixed value by setting the `color` argument directly within `geom_point()`. The color we choose must be a character string that R recognizes as a color. Here we will set the point colors to steel blue:

```
# 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
```



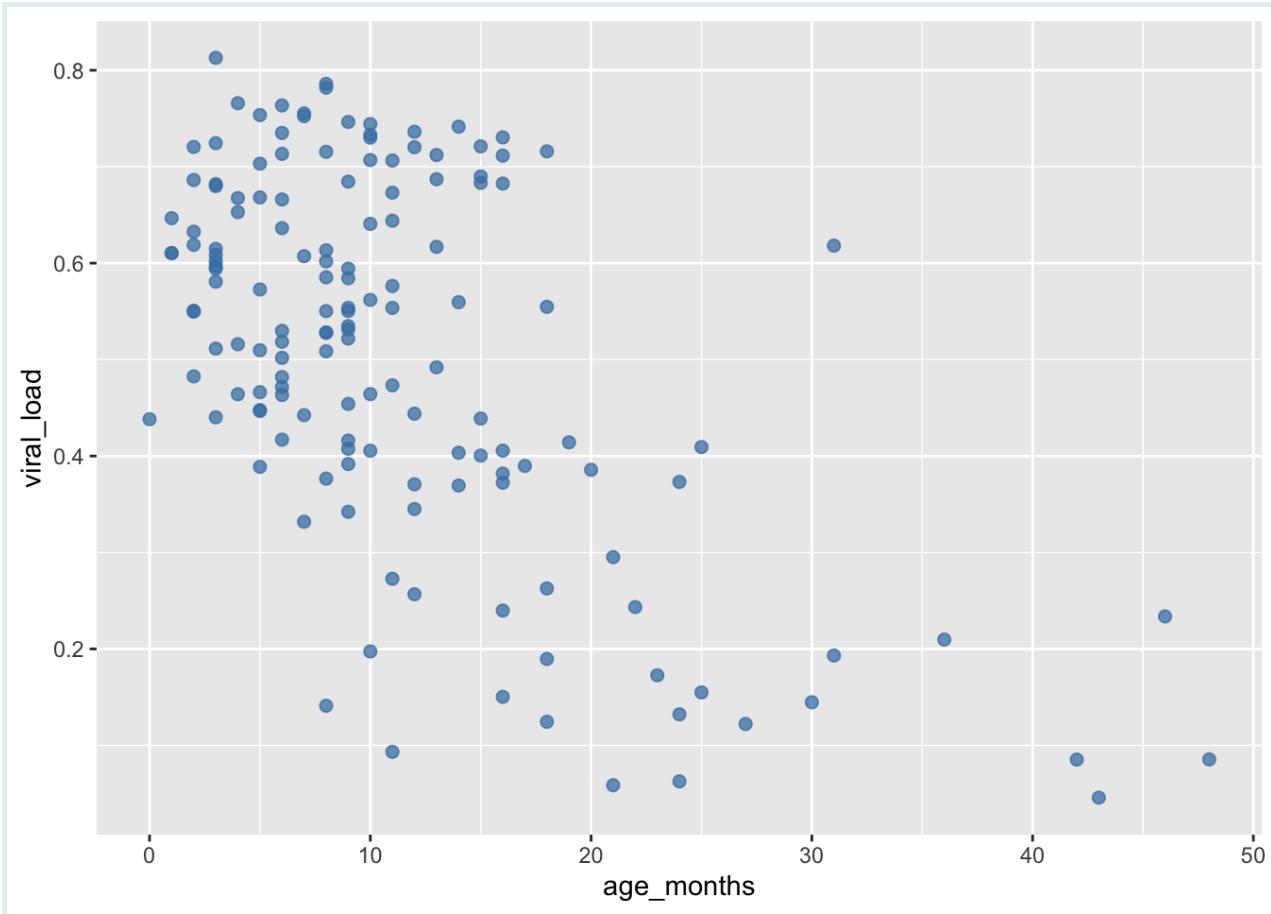
In addition to changing the default color, now we will modify the `size` aesthetic of the points by assigning it to a fixed number (in millimeters). The default size is 1 mm, so let's chose a larger value:

```
# 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.



- 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%.

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:

0	1	2	3	4
□	○	△	+	×
5	6	7	8	9
◇	▽	☒	✳	❖
10	11	12	13	14
⊕	⊗	田	⊗	□
15	16	17	18	19
■	●	▲	◆	●
20	21	22	23	24
●	●	■	◆	▲

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`.

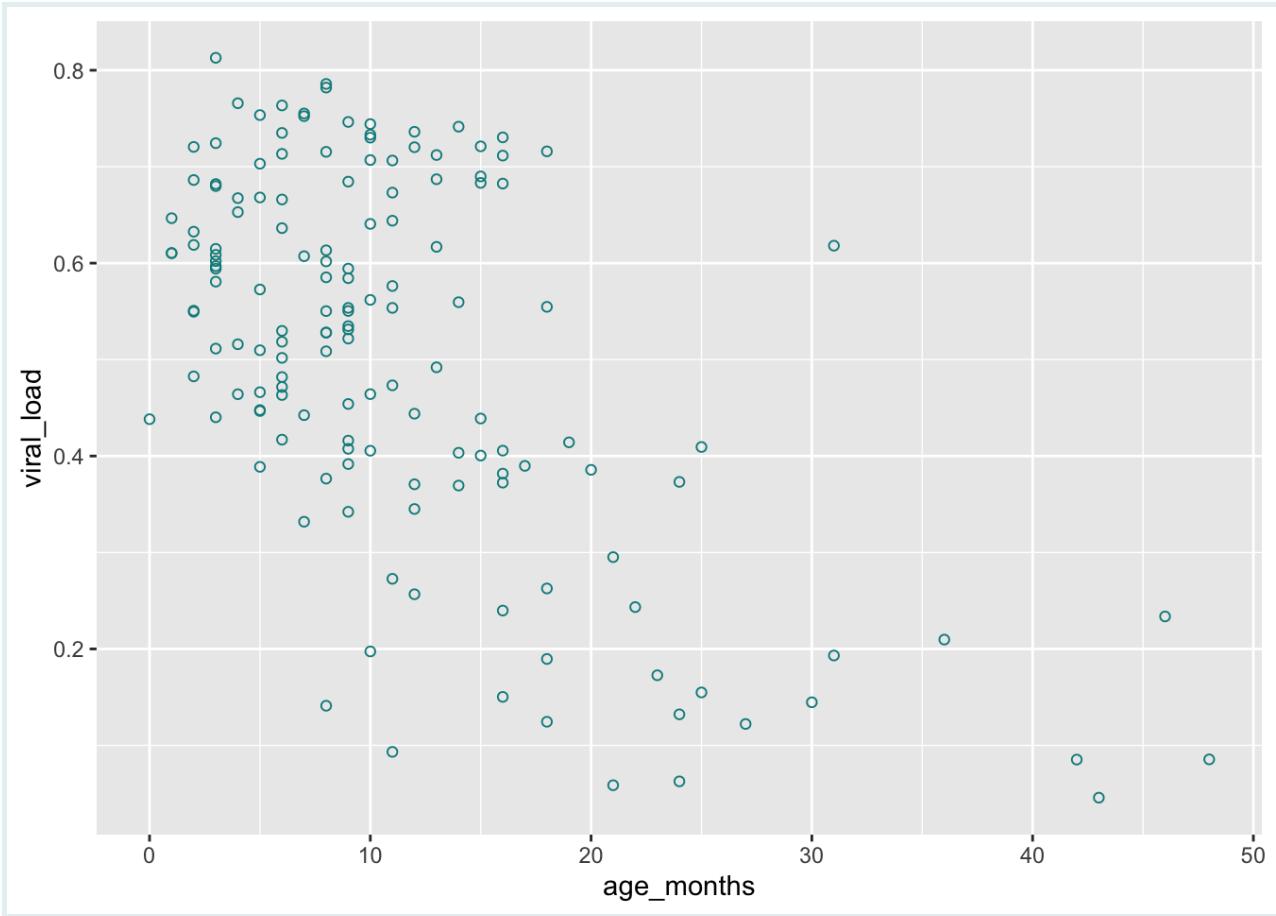
First let's modify our original scatterplot by changing the shapes to a something that can be filled in:

```
# 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:

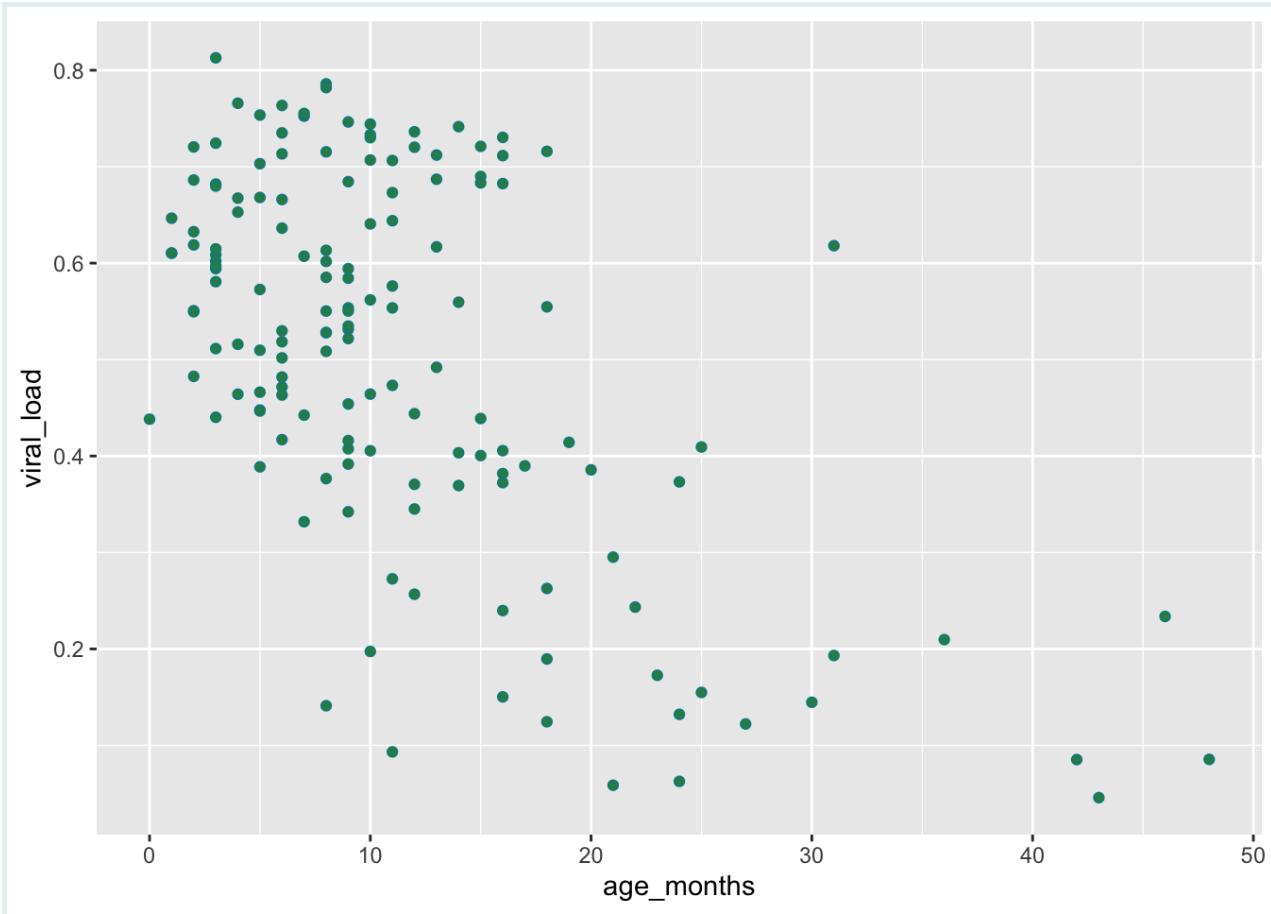
```
# Set outline color of the shapes by adding `color = cyan4`  
  
ggplot(data = malidd,  
        mapping = aes(x = age_months,  
                      y = viral_load)) +  
  geom_point(shape = 21,                      # set shapes to display  
             color = "cyan4")                  # set outline color
```



Now let's fill in the points:

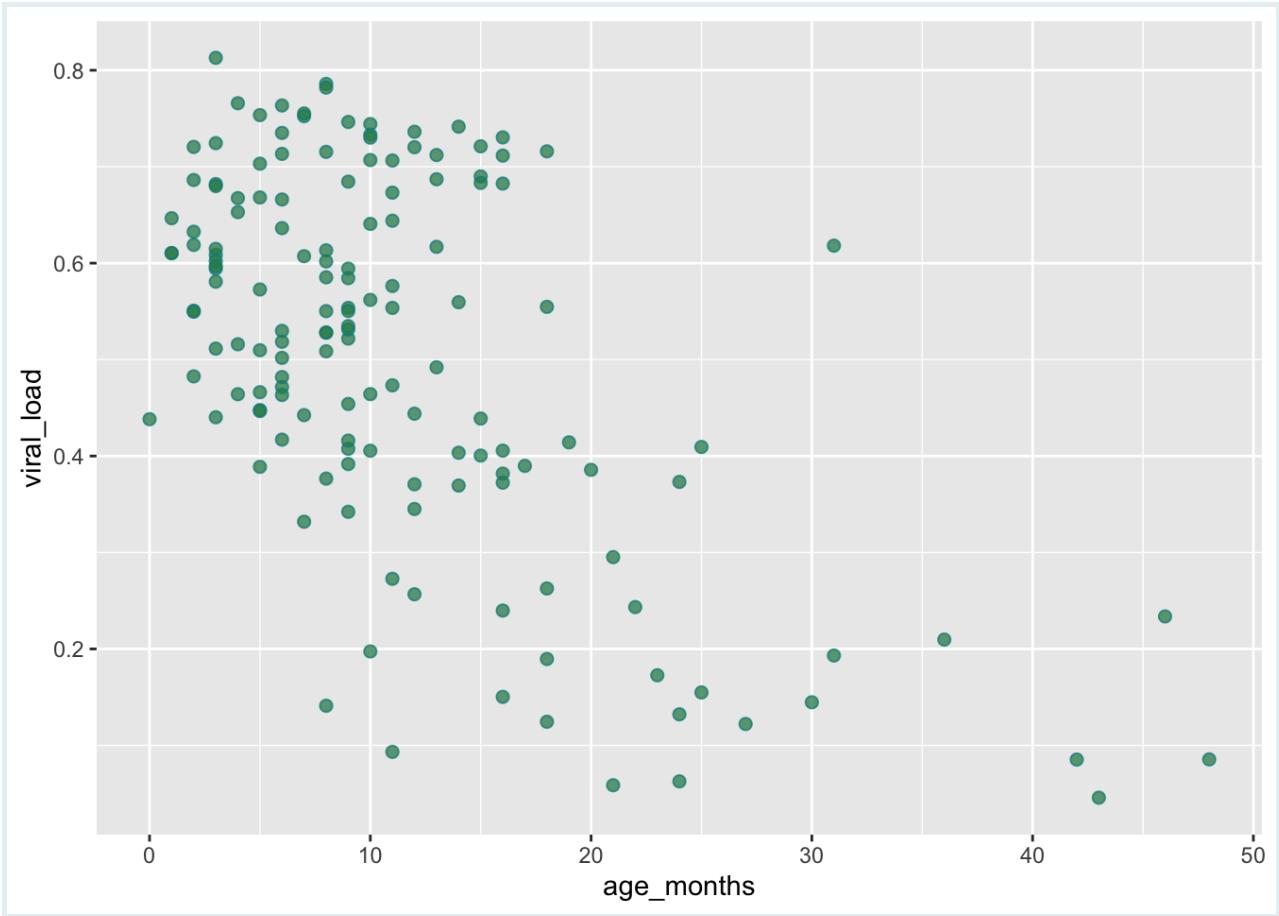
```
# Set interior color of the shapes by adding `fill = "seagreen"`

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
```



Adding a trend line

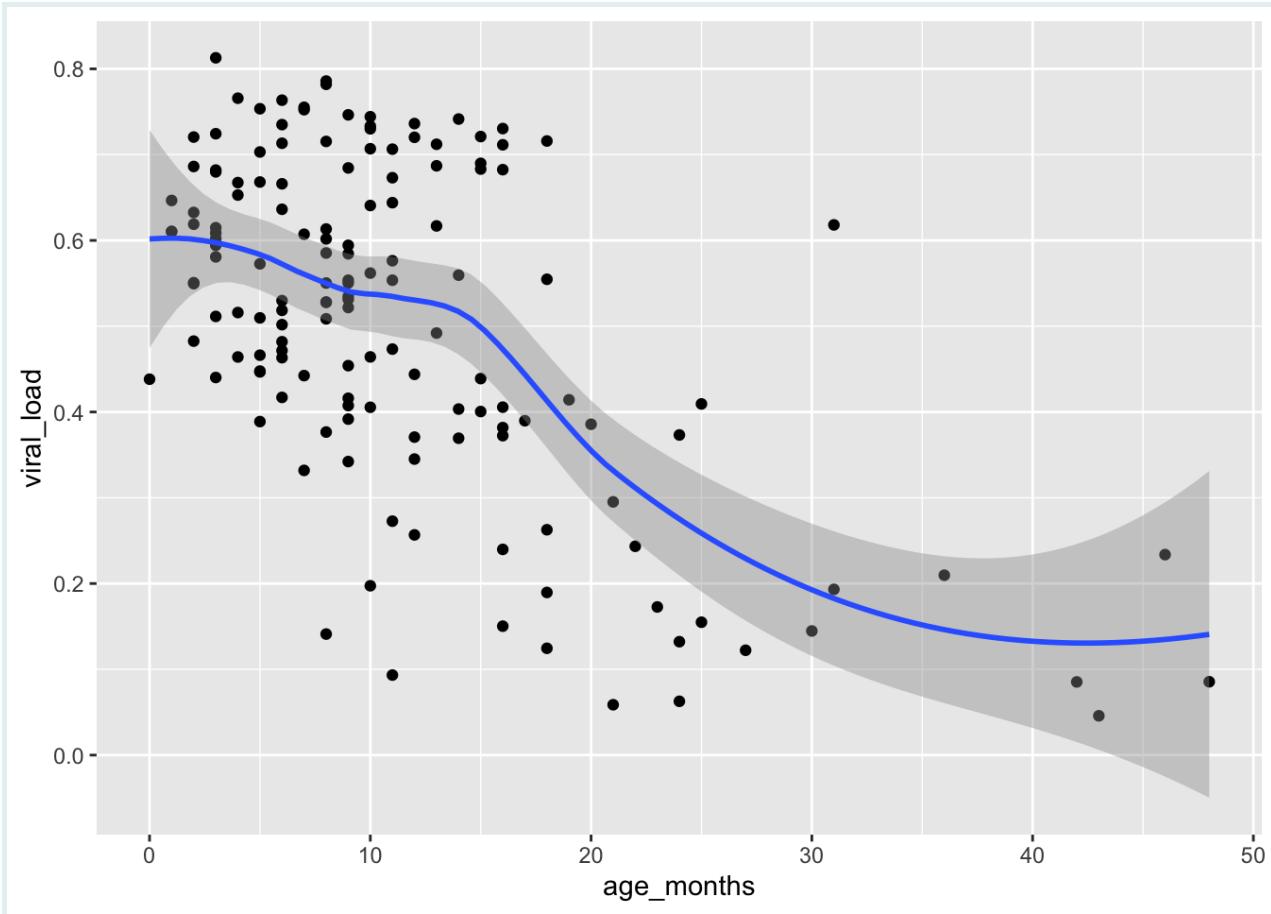
It can be hard to view relationships or trends with just points alone. Often we want to add a smoothing line in order to see what the trends look like. This can be especially helpful when trying to understand regressions.

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 an another geometric layer added onto our plot.

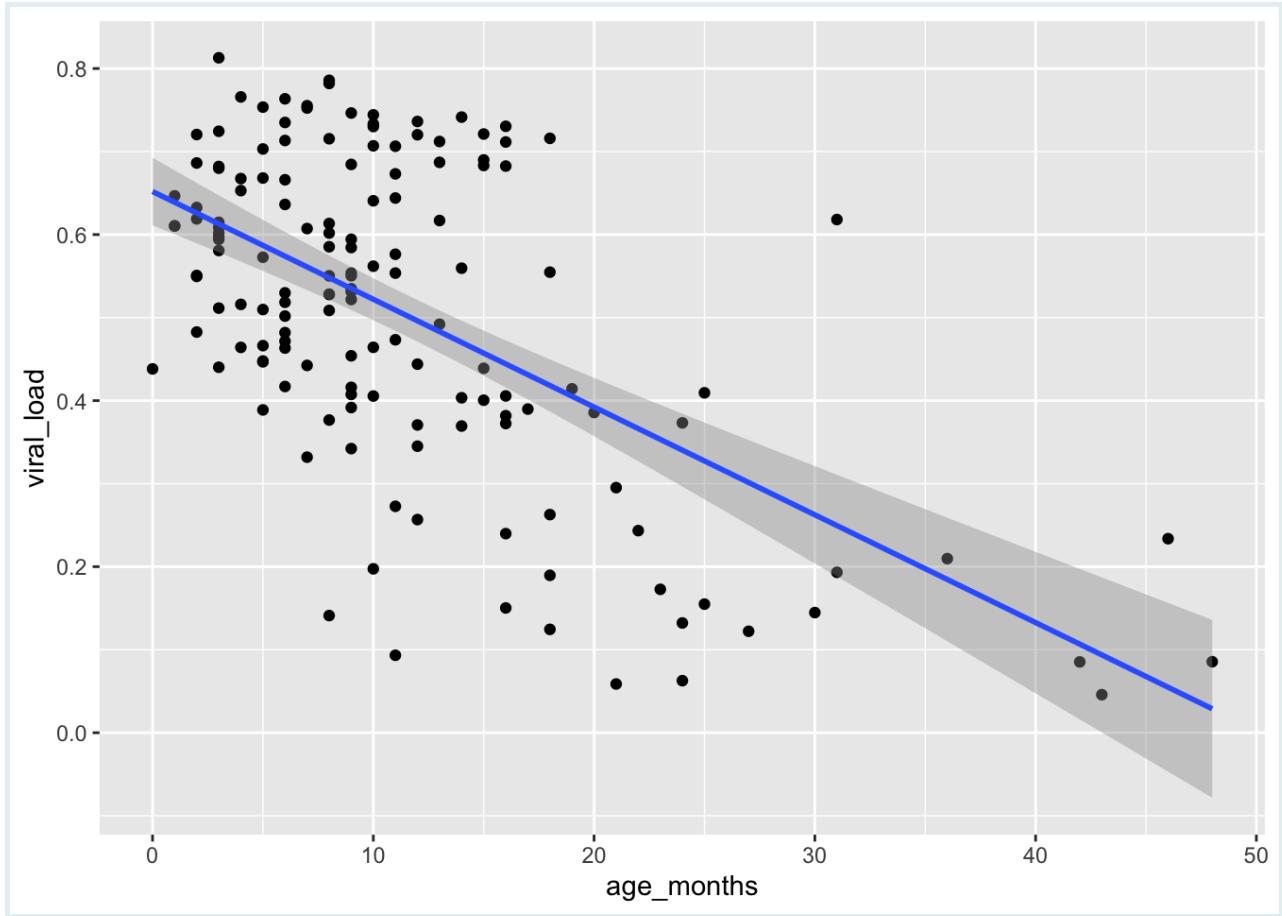
The default smoothing function used in this scatter plot is “loess” which stands for for **locally weighted scatter plot** smoothing. Loess smoothing is a process used by many statistical softwares. In {ggplot2} this generally should be done when you have less than 1000 points, otherwise it can be time 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'
```

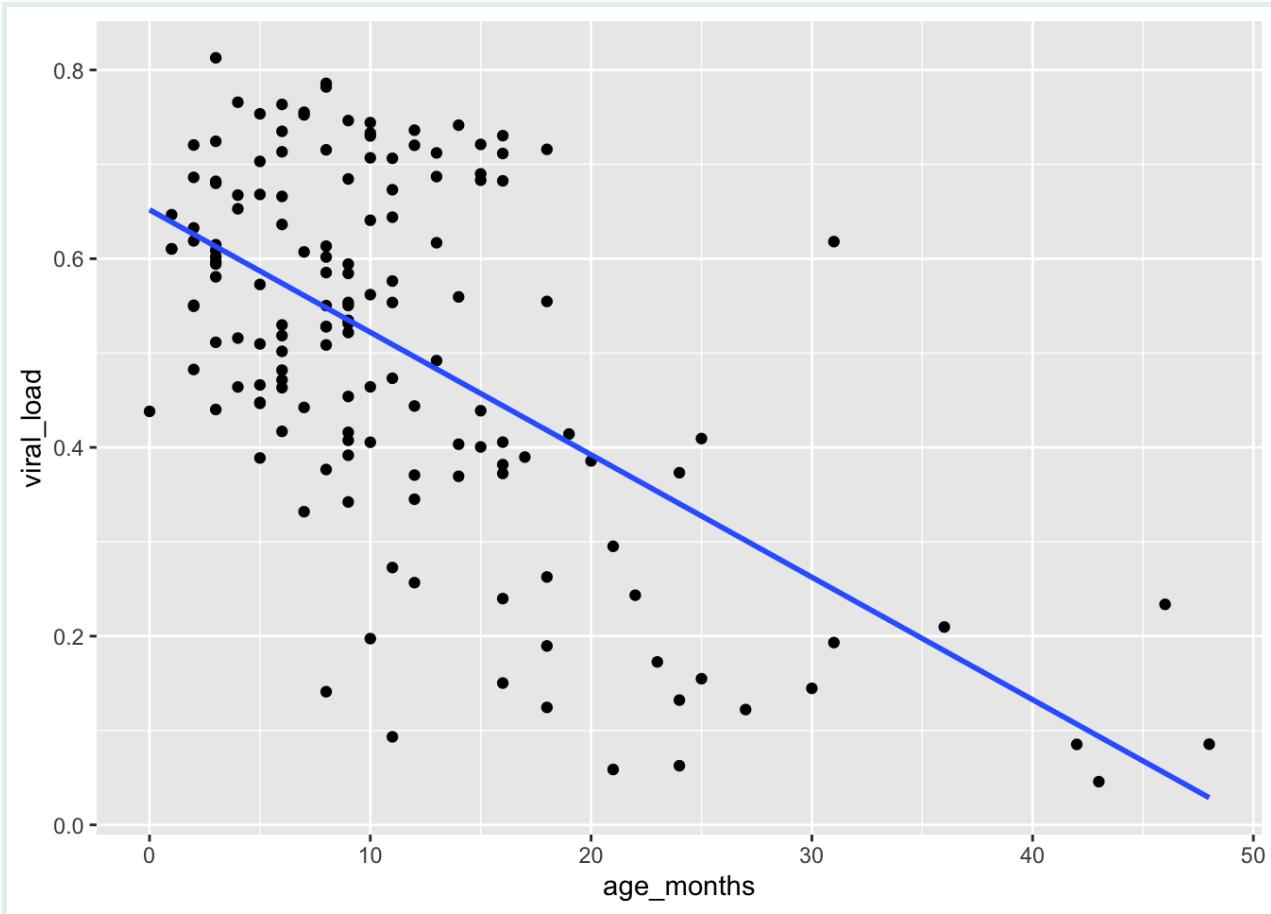


By default, 95% confidence limits for these lines are displayed.

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)
```

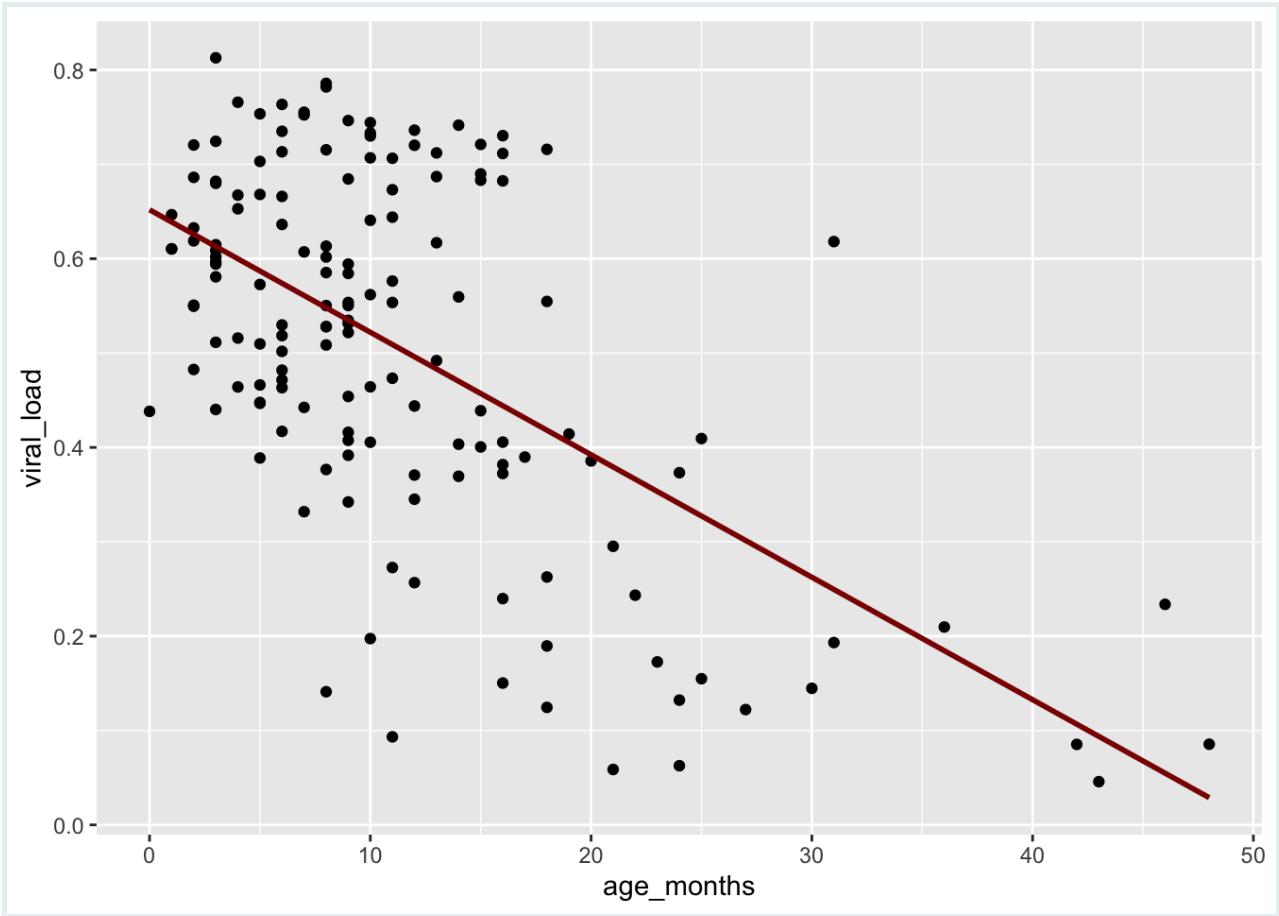
```
## `geom_smooth()` using formula = 'y ~ x'
```



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'
```

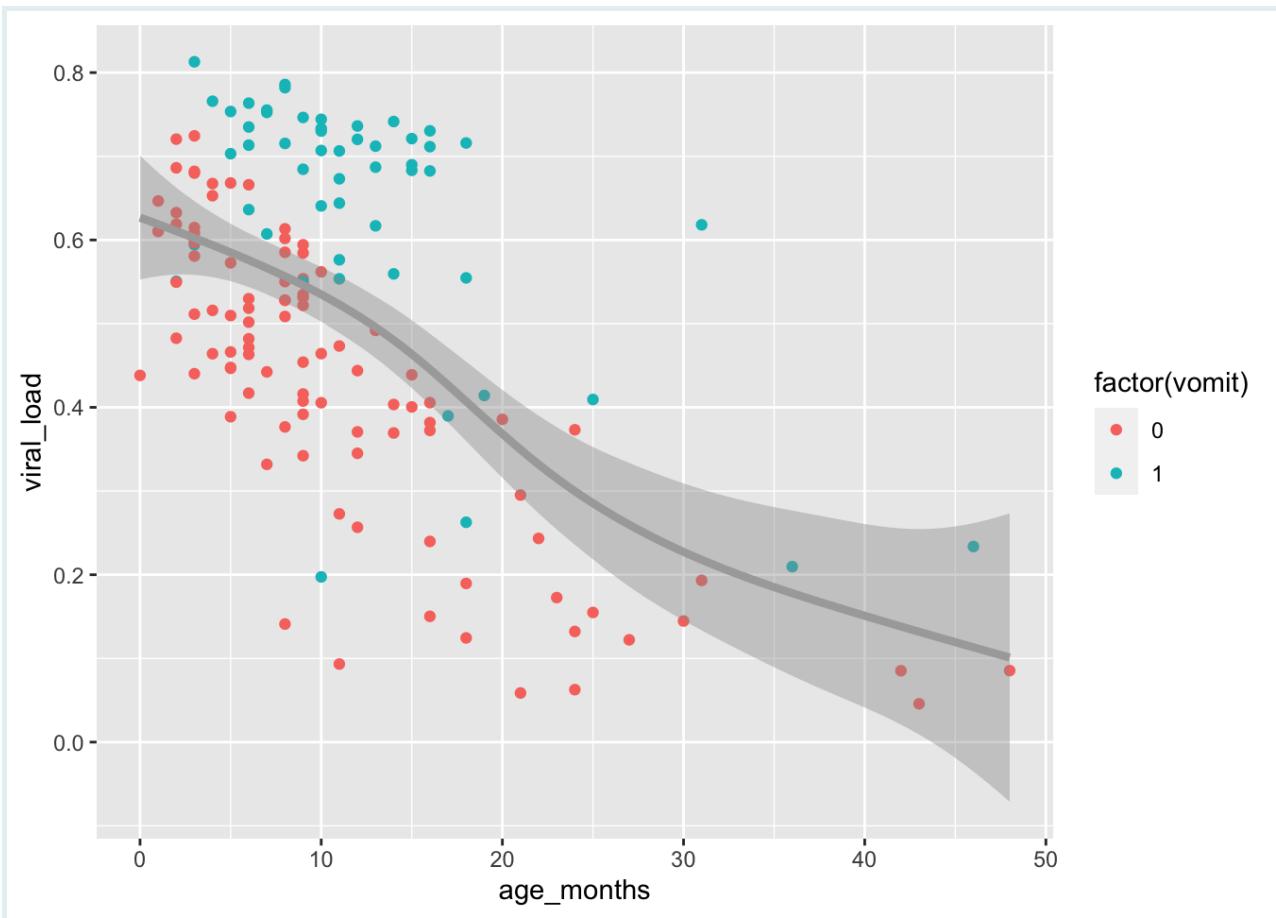


This linear regression concurs with what we initially observed in the first scatter plot. A *negative relationship* exists between `age_months` and `viral_load`: as age increases, viral load tends to decrease.

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")
```

```
## `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.



- Create a scatter plot with the `age_months` and `viral_load` 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"
```

PRACTICE

age_height_3

(in RMD)

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.

Contributors

The following team members contributed to this lesson:



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science



ADMIN TEAM

GRAPH Courses Administration Team
The GRAPH Courses team is building epidemiological training courses to enhance disease surveillance and data science for public health across the globe

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 Granados. 2020. *Workshop 3: Introduction to Data Visualisation with {ggplot2}*. <http://r.qcbs.ca/workshop03/book-en/>.

```
.r tgc_license()
```

Lesson notes | Lines, scales, and labels

Created by the GRAPH Courses team

January 2023

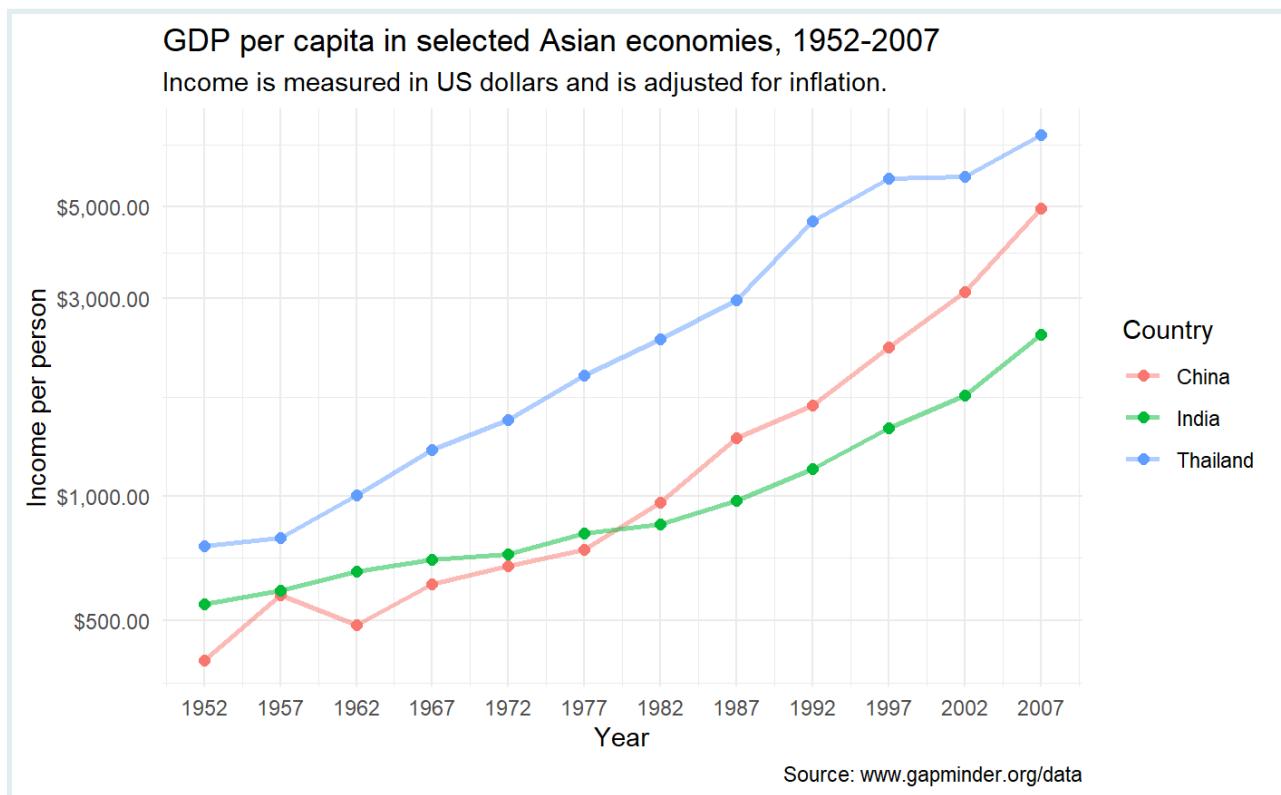
This document serves as an accompaniment for a lesson found on <https://thegraphcourses.org>.

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

Learning Objectives
Introduction
Packages
The gapminder data frame
Line graphs via <code>geom_line()</code>
Fixed aesthetics in <code>geom_line()</code>
Combining compatible geoms
Mapping data to multiple lines
Modifying continuous x & y scales
Scale breaks
Logarithmic scaling
Labeling with <code>labs()</code>
Preview: Themes
Wrap up

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.



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.

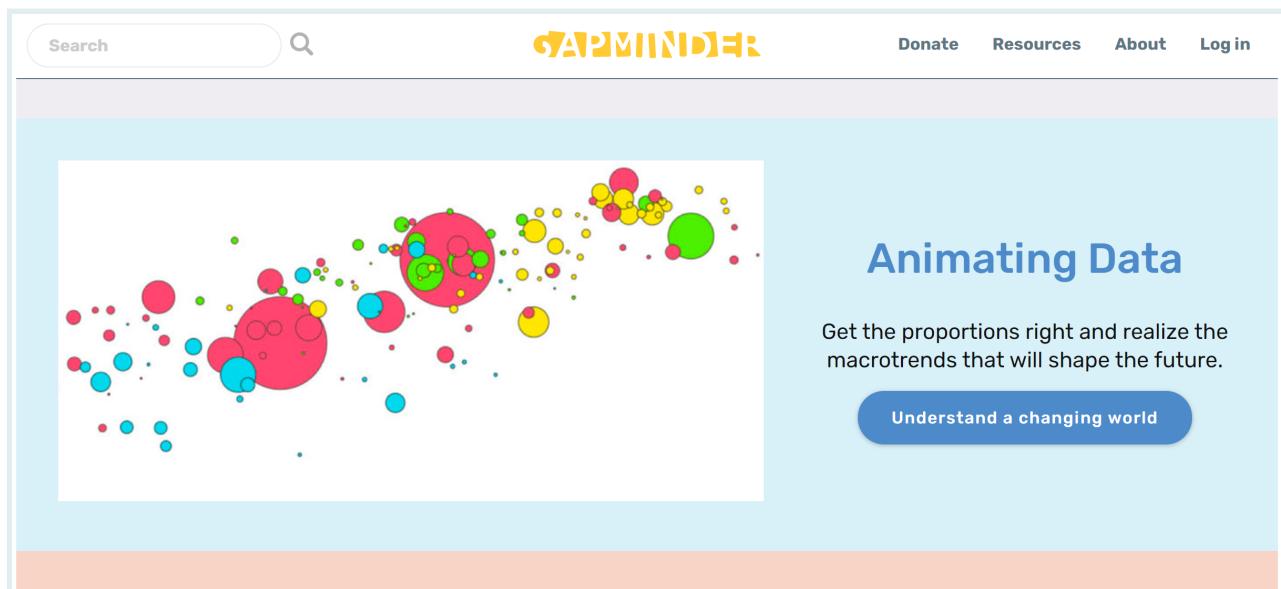
The most common examples of line graphs have some notion of **time on the x-axis**: hours, days, weeks, years, etc. Since time is sequential, we connect consecutive observations of the variable on the y-axis with a line. Line graphs that have some notion of time on the x-axis are also called **time series plots**.

Packages

```
# Load packages  
pacman::p_load(tidyverse,  
                 gapminder,  
                 here)
```

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 complied by the Gapminder Foundation.



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
```

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.

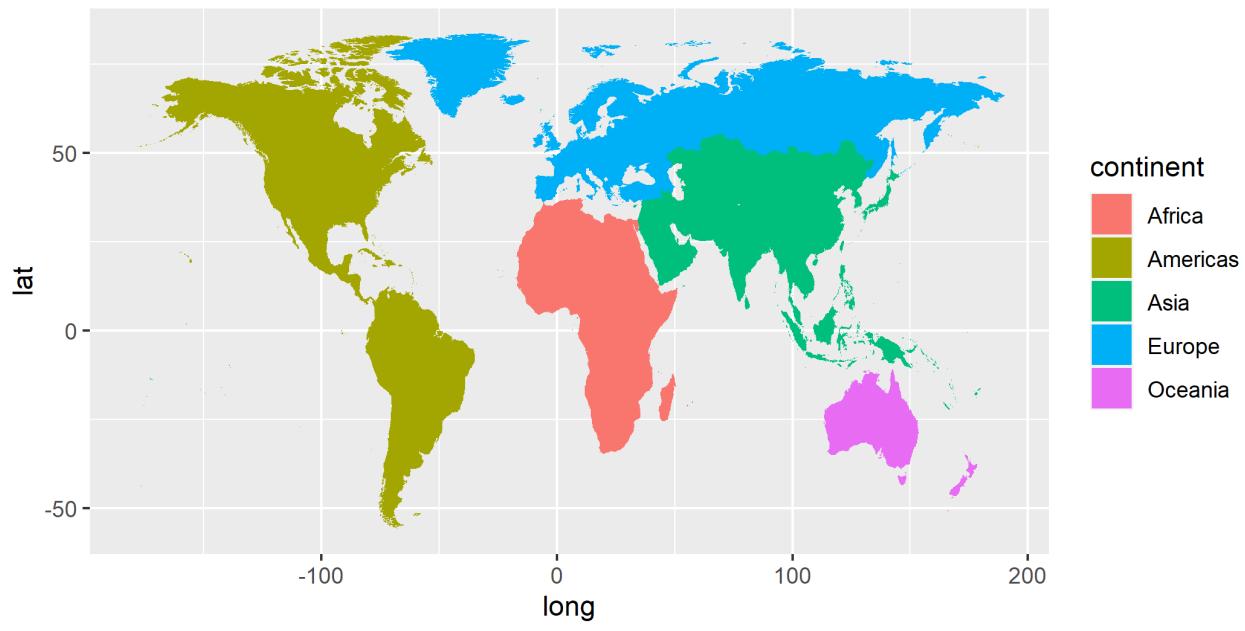
```
# Data structure
str(gapminder)

## # 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 ...
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 16317921 22227415 ...
## $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

This version of the `gapminder` dataset contains information for **142 countries**, divided into **5 continents**.

Gapminder world regions

Five regions in the `continent` variable of `gapminder`



```
# Data summary  
summary(gapminder)
```

```
##           country      continent       year   lifeExp     pop  
gdpPercap  
##  Afghanistan: 12    Africa :624   Min.   :1952   Min.   :23.60   Min.  
:6.001e+04  Min.   : 241.2  
##  Albania     : 12    Americas:300   1st Qu.:1966   1st Qu.:48.20   1st  
Qu.:2.794e+06  1st Qu.: 1202.1  
##  Algeria     : 12    Asia   :396   Median :1980   Median :60.71   Median  
:7.024e+06  Median : 3531.8  
##  Angola      : 12    Europe :360   Mean   :1980   Mean   :59.47   Mean  
:2.960e+07  Mean   : 7215.3  
##  Argentina   : 12    Oceania: 24   3rd Qu.:1993   3rd Qu.:70.85   3rd  
Qu.:1.959e+07  3rd Qu.: 9325.5  
##  Australia   : 12                Max.   :2007   Max.   :82.60   Max.  
:1.319e+09  Max.   :113523.1  
##  (Other)      :1632
```

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
```

REMINDER



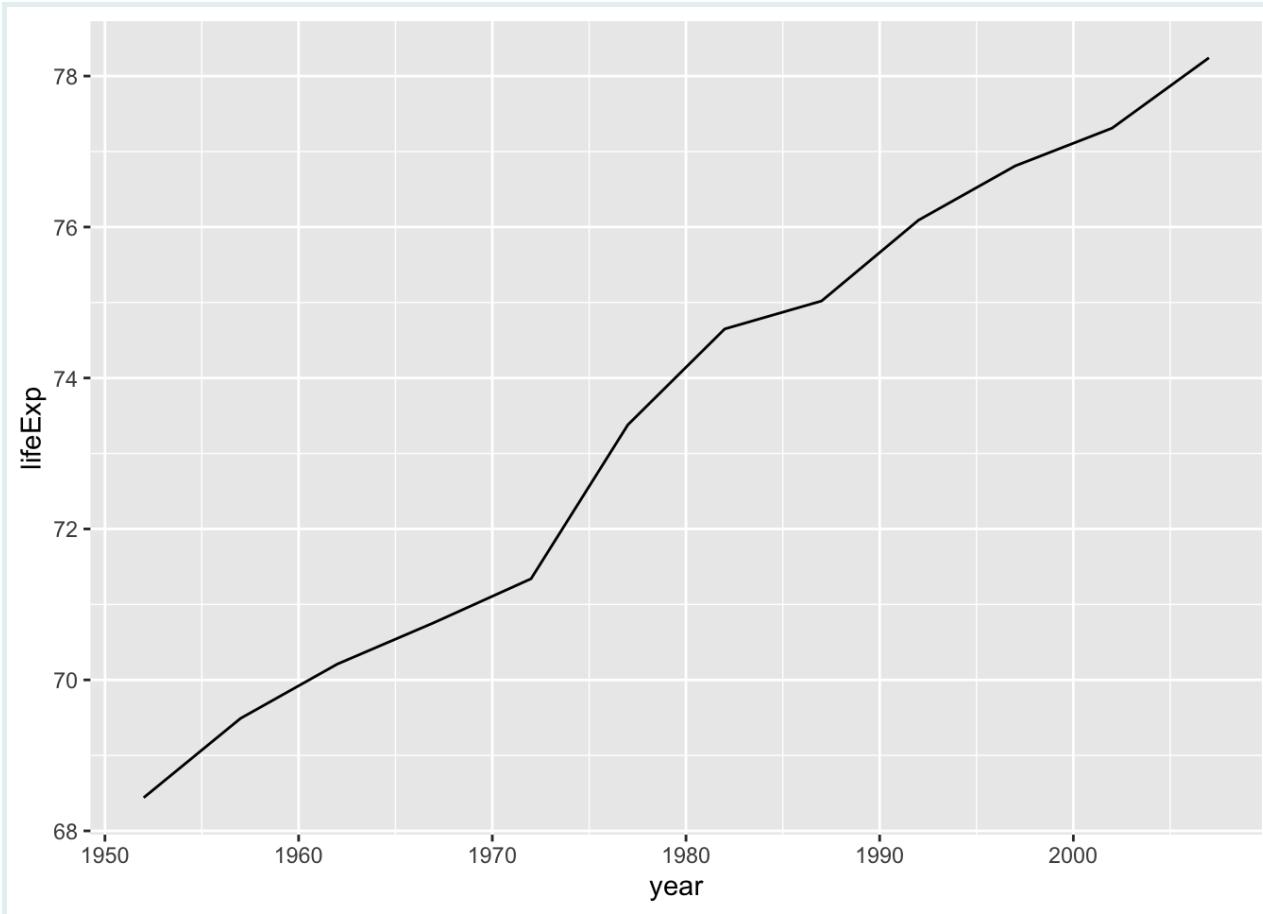
The code above is covered in our course on Data Wrangling using the `{dplyr}` package. Data wrangling is the process of transforming and modifying existing data with the intent of making it more appropriate for analysis purposes. For example, this code segment used the `filter()` function to create a 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).

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()
```

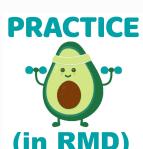


Much as with the `ggplot()` code that created the scatterplot of age and viral load with `geom_point()`, let's break down this code piece-by-piece in terms of the grammar of graphics:

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()`.



Create a time series plot of the GPD per capita (`gdpPercap`) recorded in the `gap_US` data frame by using `geom_line()` to create a line graph.

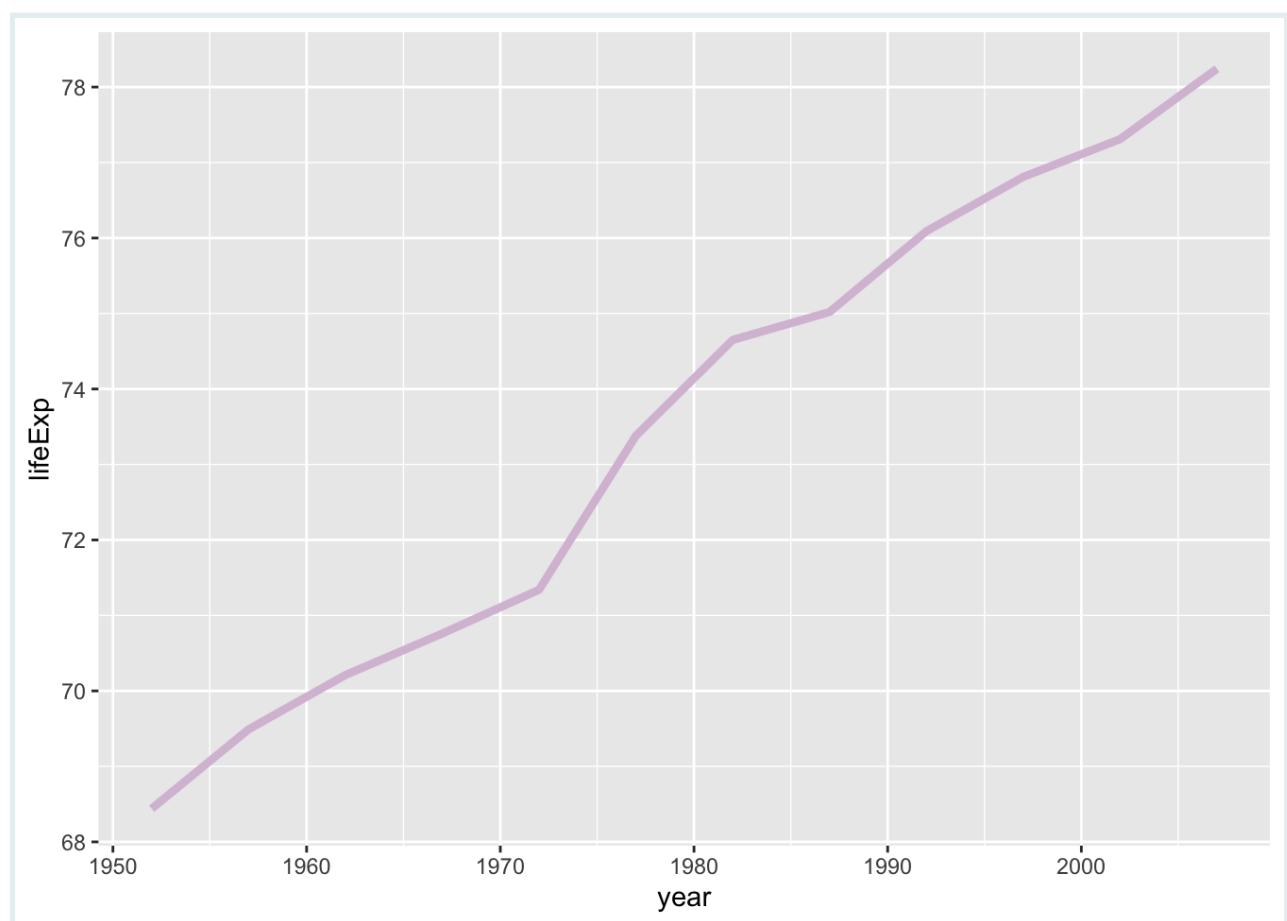
Fixed aesthetics in `geom_line()`

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)
```

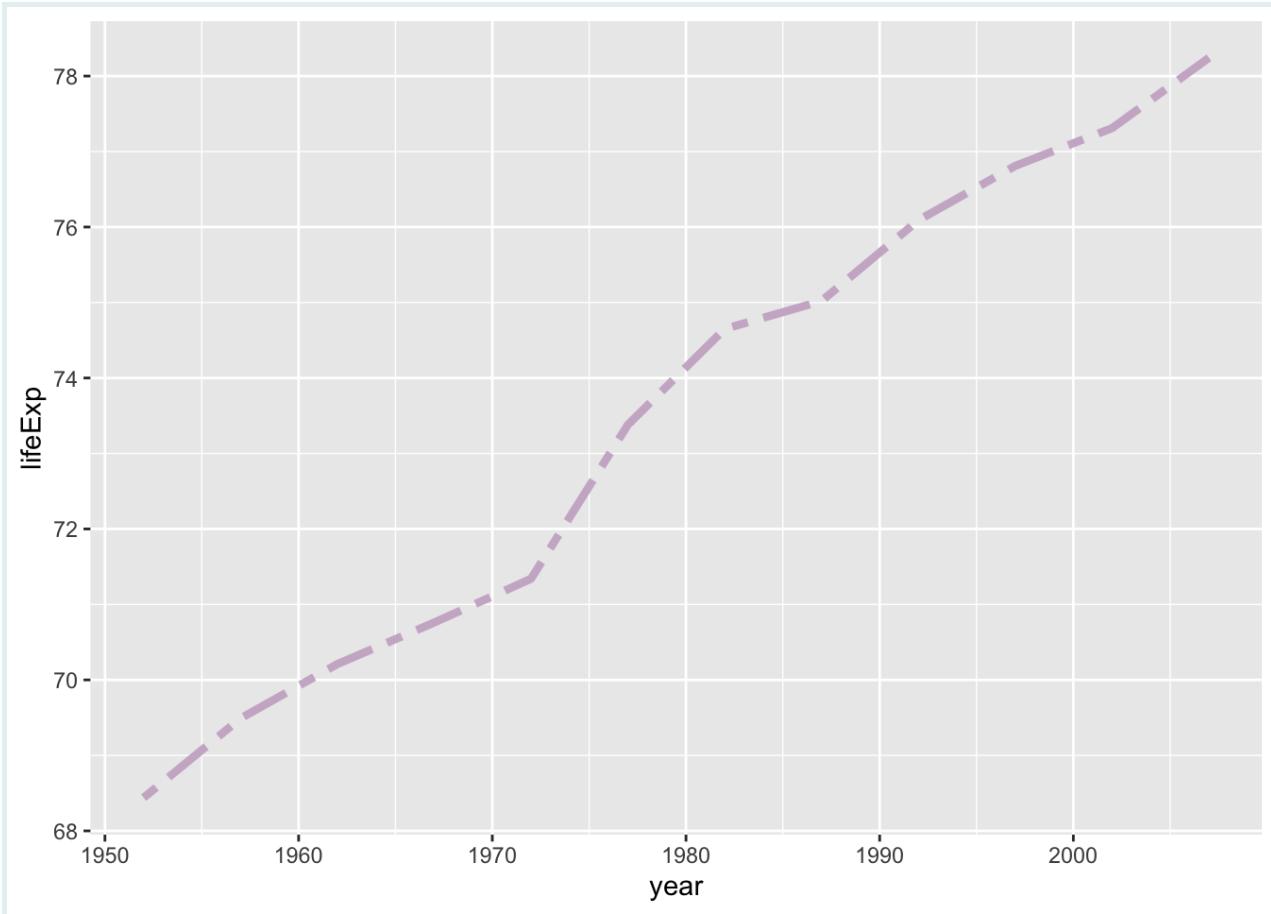


In this lesson we introduce a new fixed aesthetic that is specific to line graphs: `linetype` (or `lty` for short).

<hr/>	lty = 0 or 'blank'
<hr/>	lty = 1 or 'solid'
<hr/>	lty = 2 or 'dashed'
<hr/>	lty = 3 or 'dotted'
<hr/>	lty = 4 or 'dotdash'
<hr/>	lty = 5 or 'longdash'
<hr/>	lty = 6 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")
```



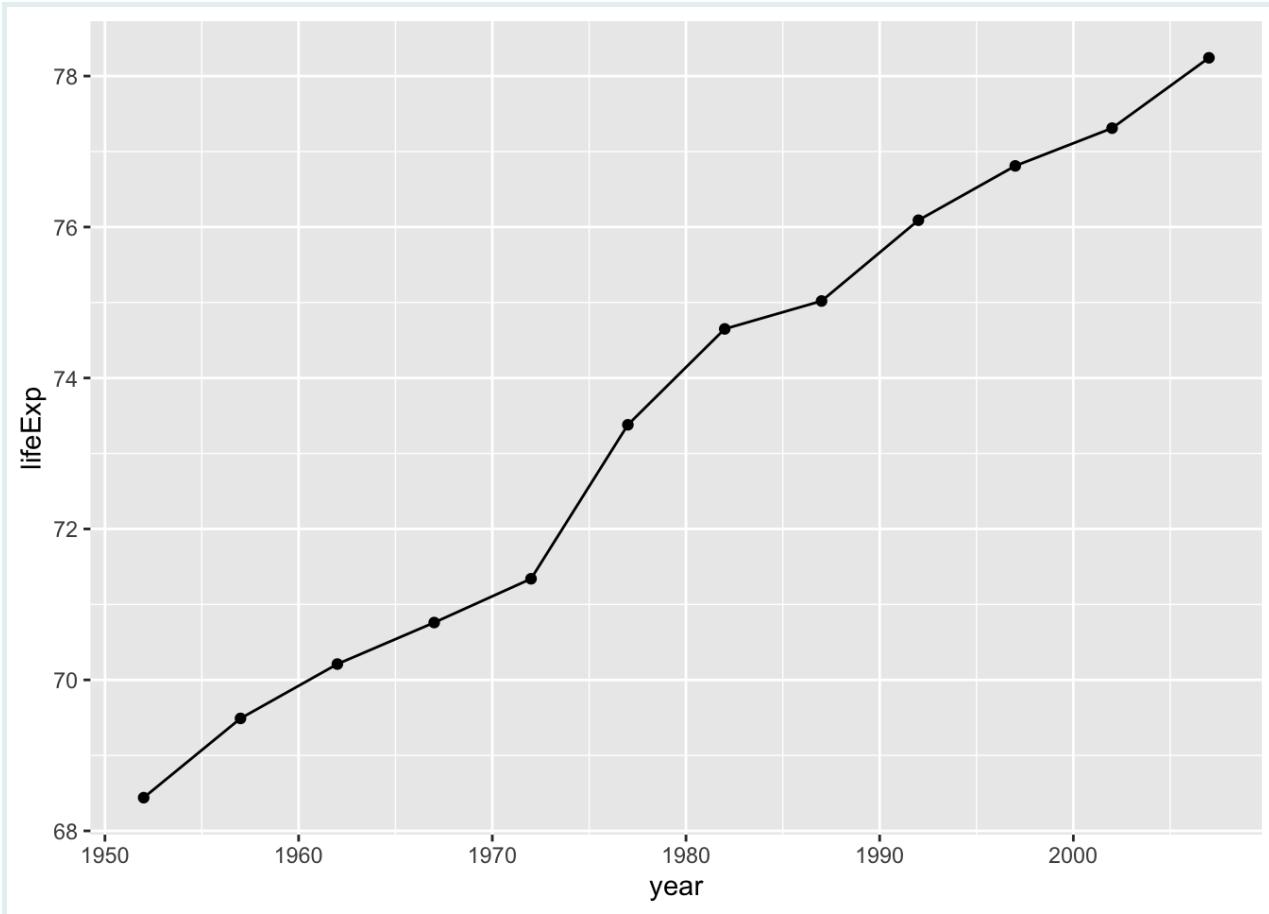
In these line graphs, it can be hard to tell where exactly there data points are. In the next plot, we'll add points to make this clearer.

Combining compatible geoms

As long as the geoms are compatible, we can layer them on top of one another to further customize a graph.

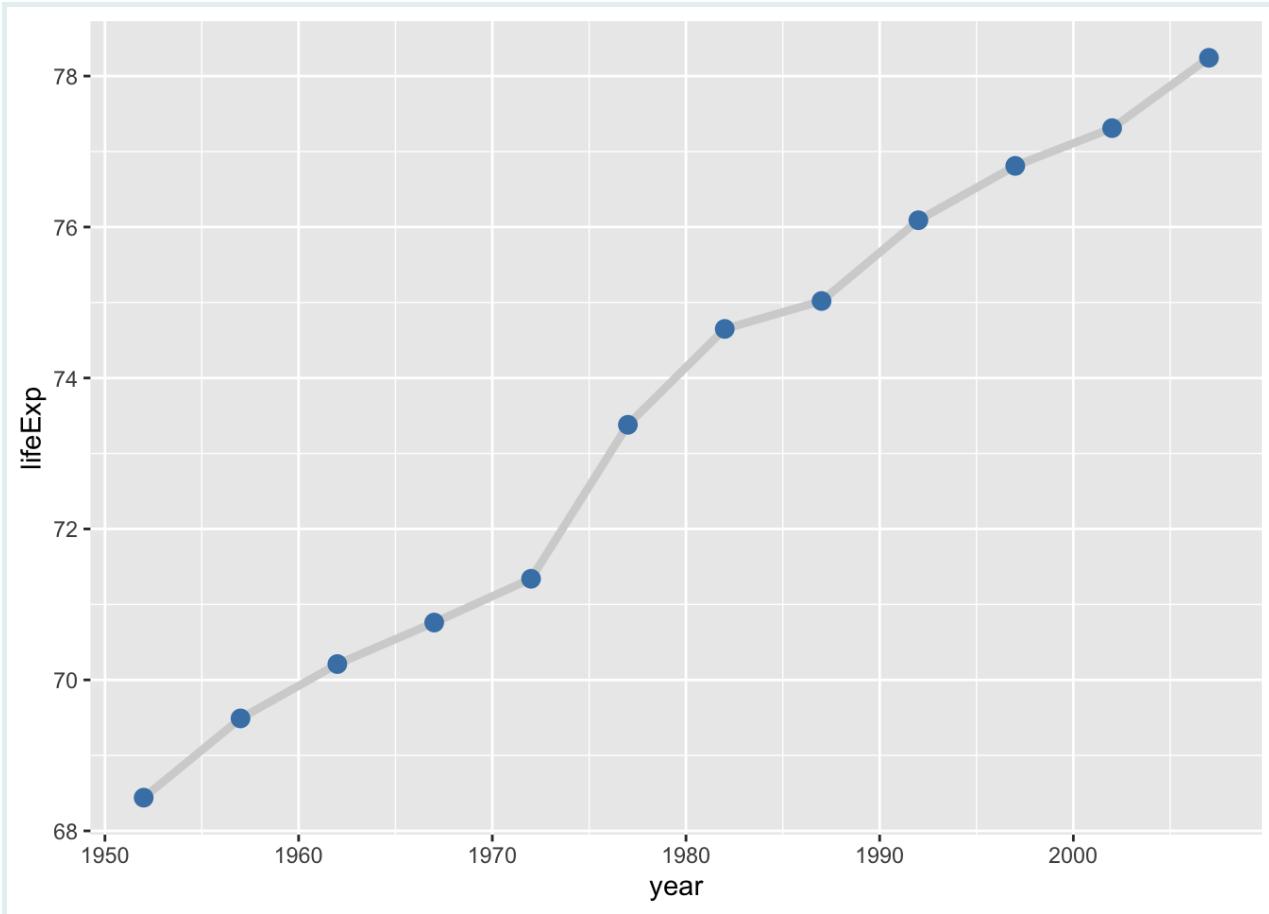
For example, we can add points to our line graph using the `+` sign to add a second geom layer with `geom_point()`:

```
# 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")
```



Building on the code above, visualize the relationship between time and **GDP 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.

Mapping data to multiple lines

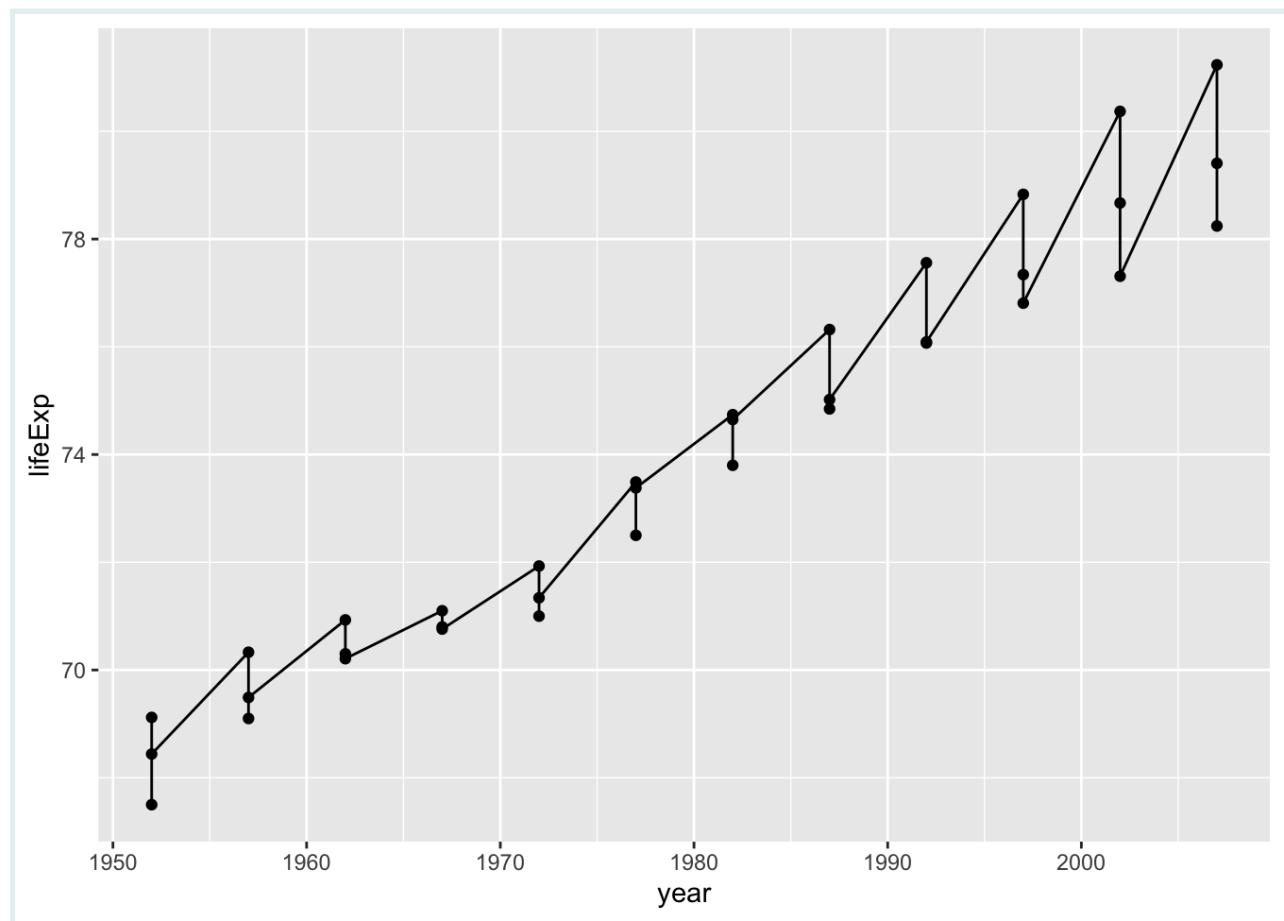
In the previous section, we only looked at data from one country, but what if we want to plot data for multiple countries and compare?

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
```

If we simply enter it using the same code and change the data layer, the lines are not automatically separated by country:

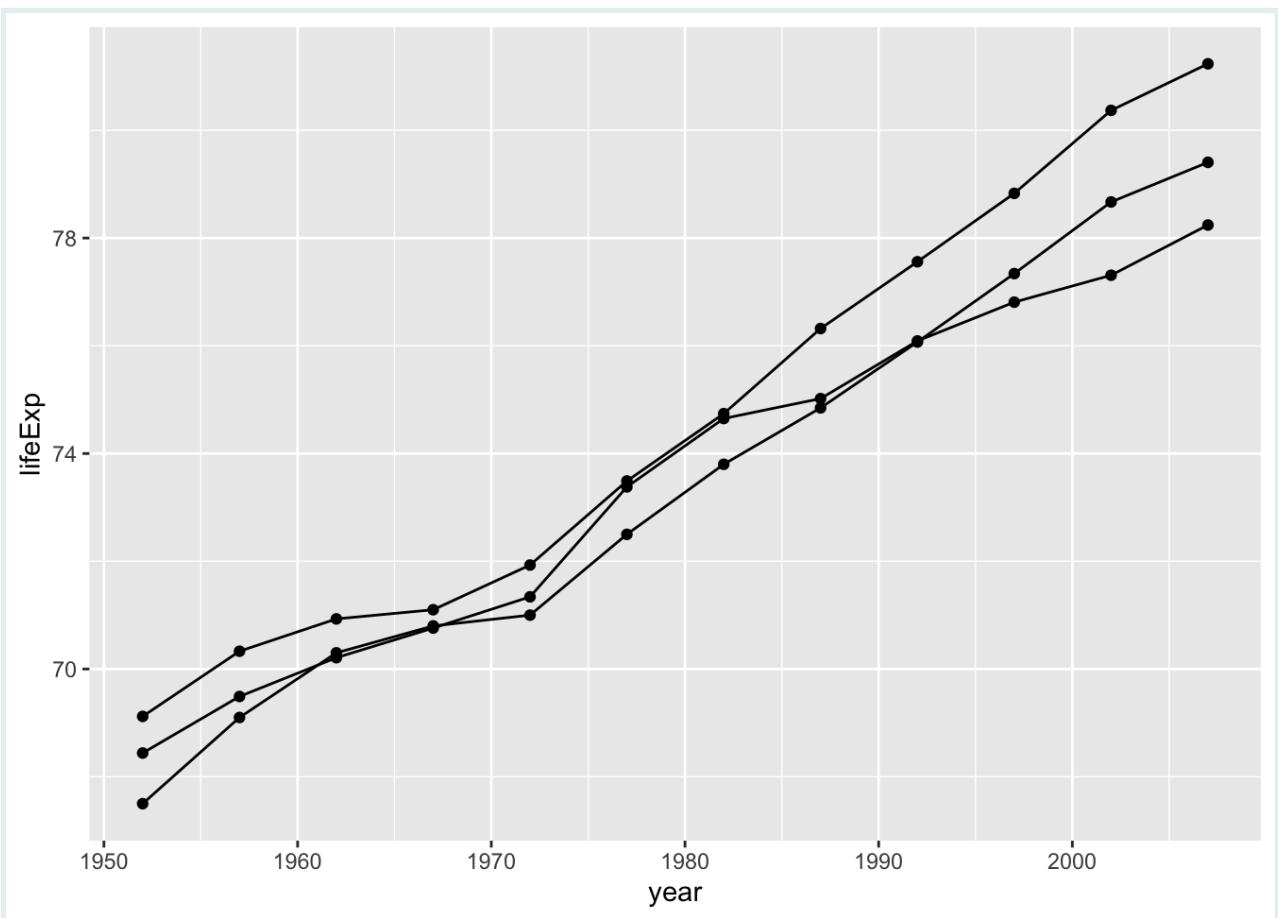
```
# Line graph with no grouping aesthetic
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                      x = year)) +
  geom_line() +
  geom_point()
```



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:

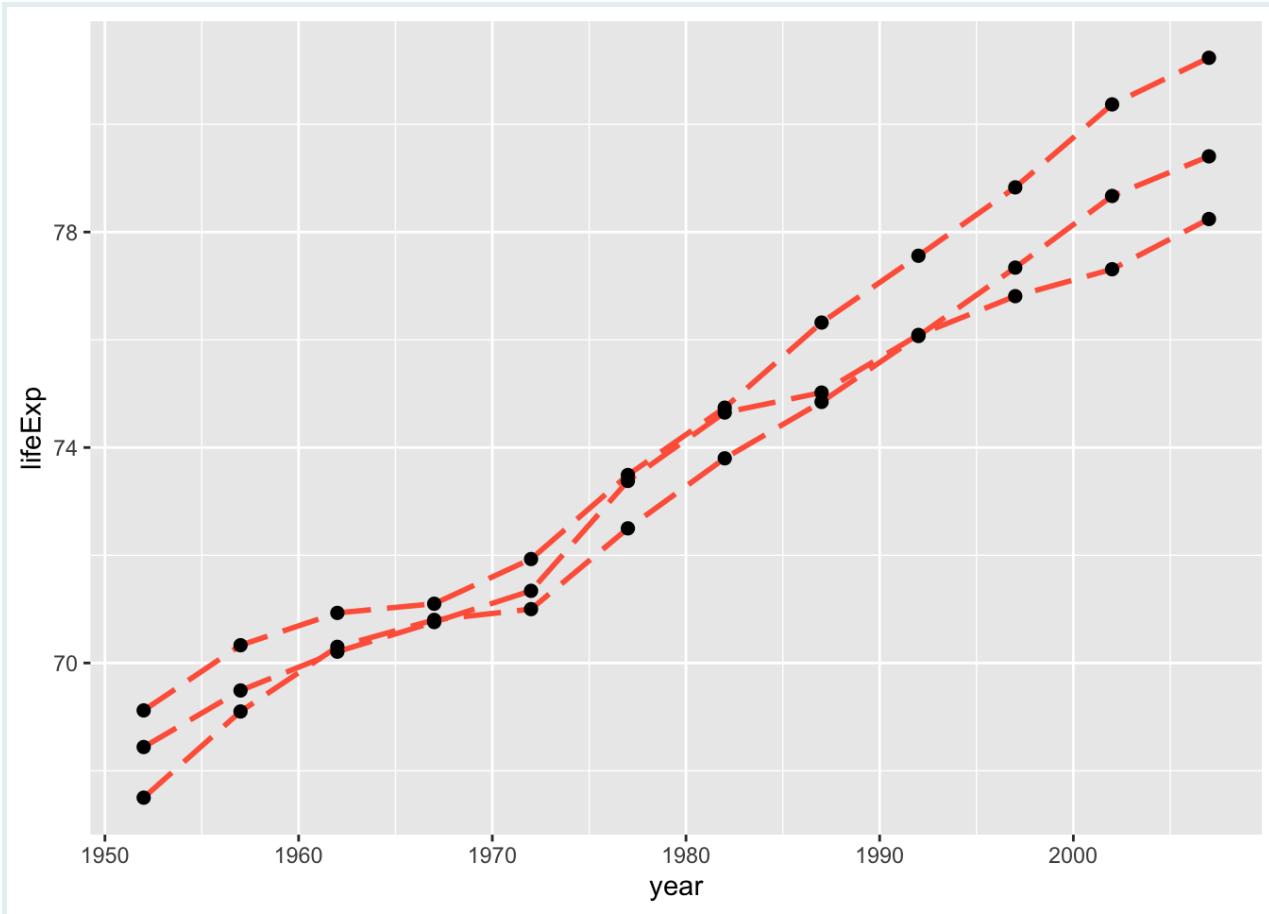
```
# Line graph with grouping by a categorical variable
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     group = country)) +
  geom_line() +
  geom_point()
```



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.

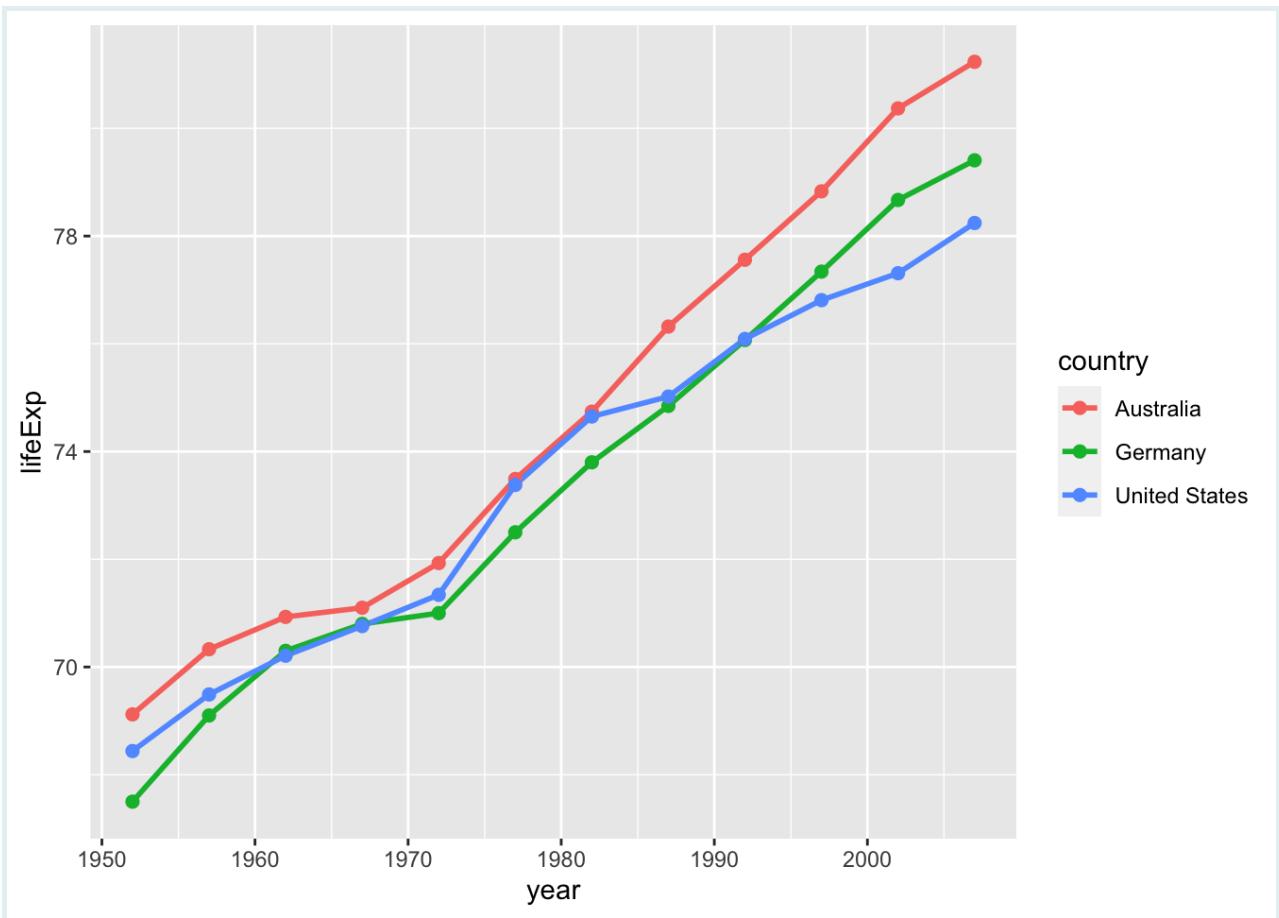
```
# Applying fixed aesthetics to multiple lines
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp,
                     x = year,
                     group = country)) +
  geom_line(linetype="longdash",           # set line type
            color="tomato",             # set line color
            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.

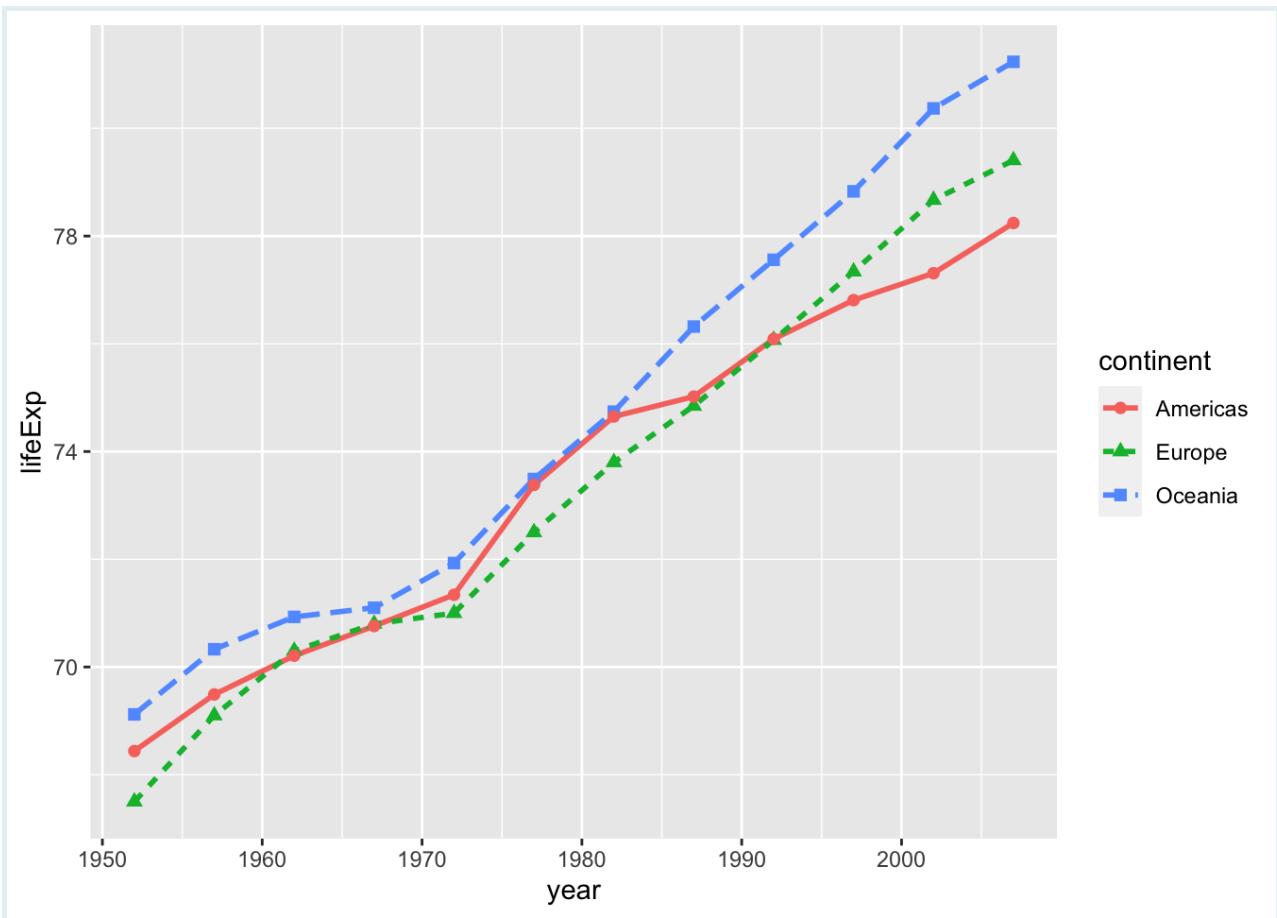
```
# Map country to color
ggplot(data = gap_mini,
       mapping = aes(y = lifeExp, x = year,
                      group = country,
                      color = country)) +
  geom_line(size = 1) +
  geom_point(size = 2)
```



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

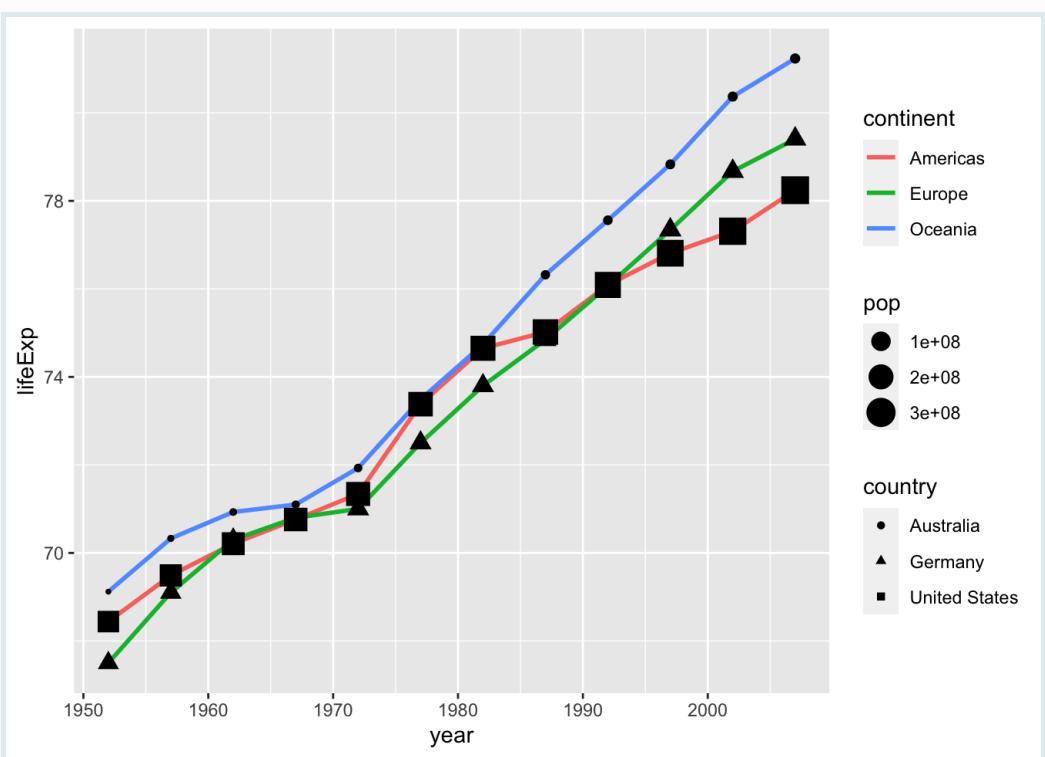
Mappings can either go in the `ggplot()` function or in `geom_*` layer.

CHALLENGE



For example, aesthetic mappings can go in `geom_line()` and will only be applied to that layer:

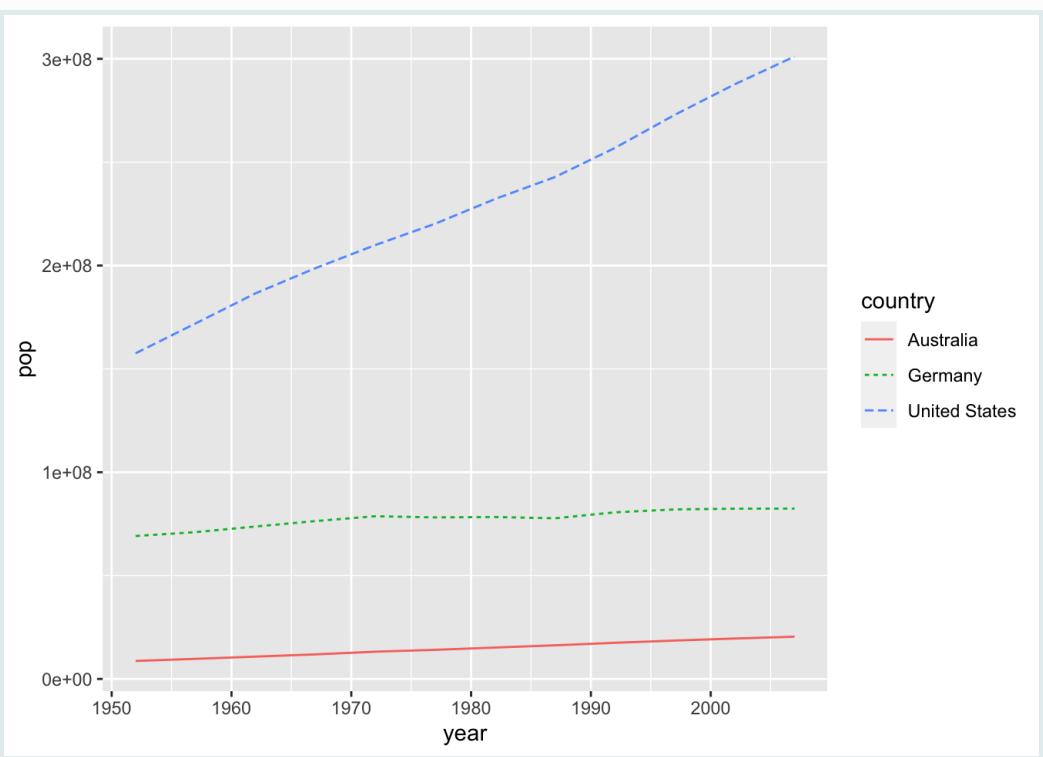
```
ggplot(data = gap_mini,
       mapping = aes(x = year,
                     y = lifeExp)) +
  geom_line(size = 1, mapping = aes(color = continent)) +
  geom_point(mapping = aes(shape = country,
                           size = pop))
```



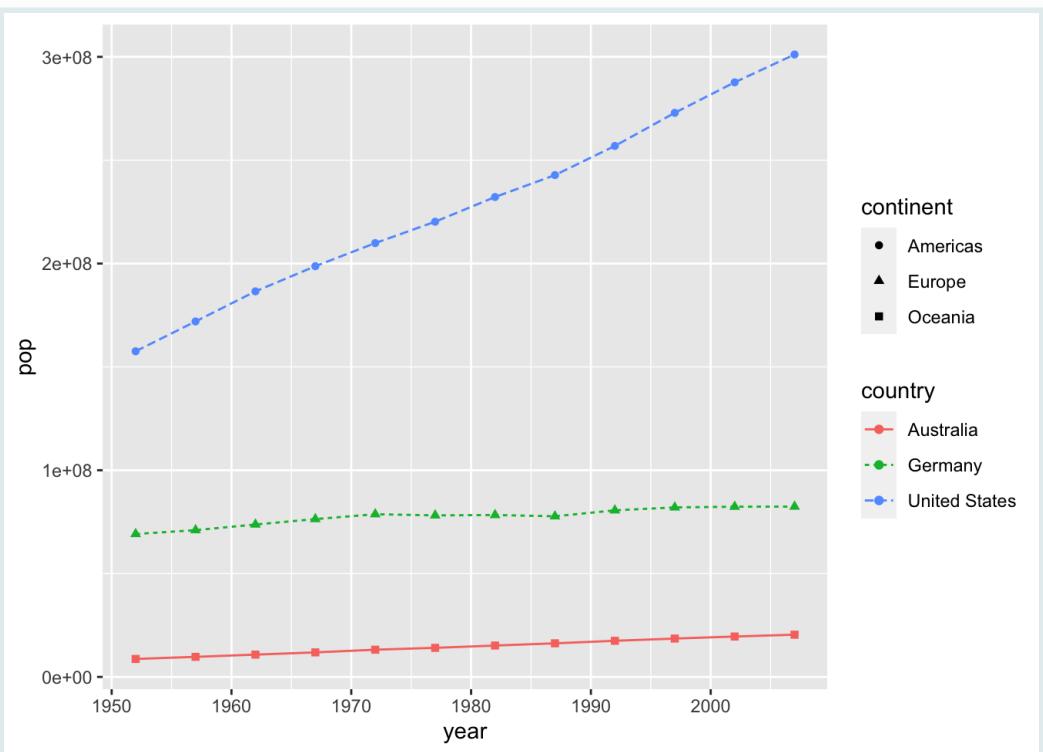
Try adding `mapping = aes()` in `geom_point()` and map continent to any valid aesthetic!



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:

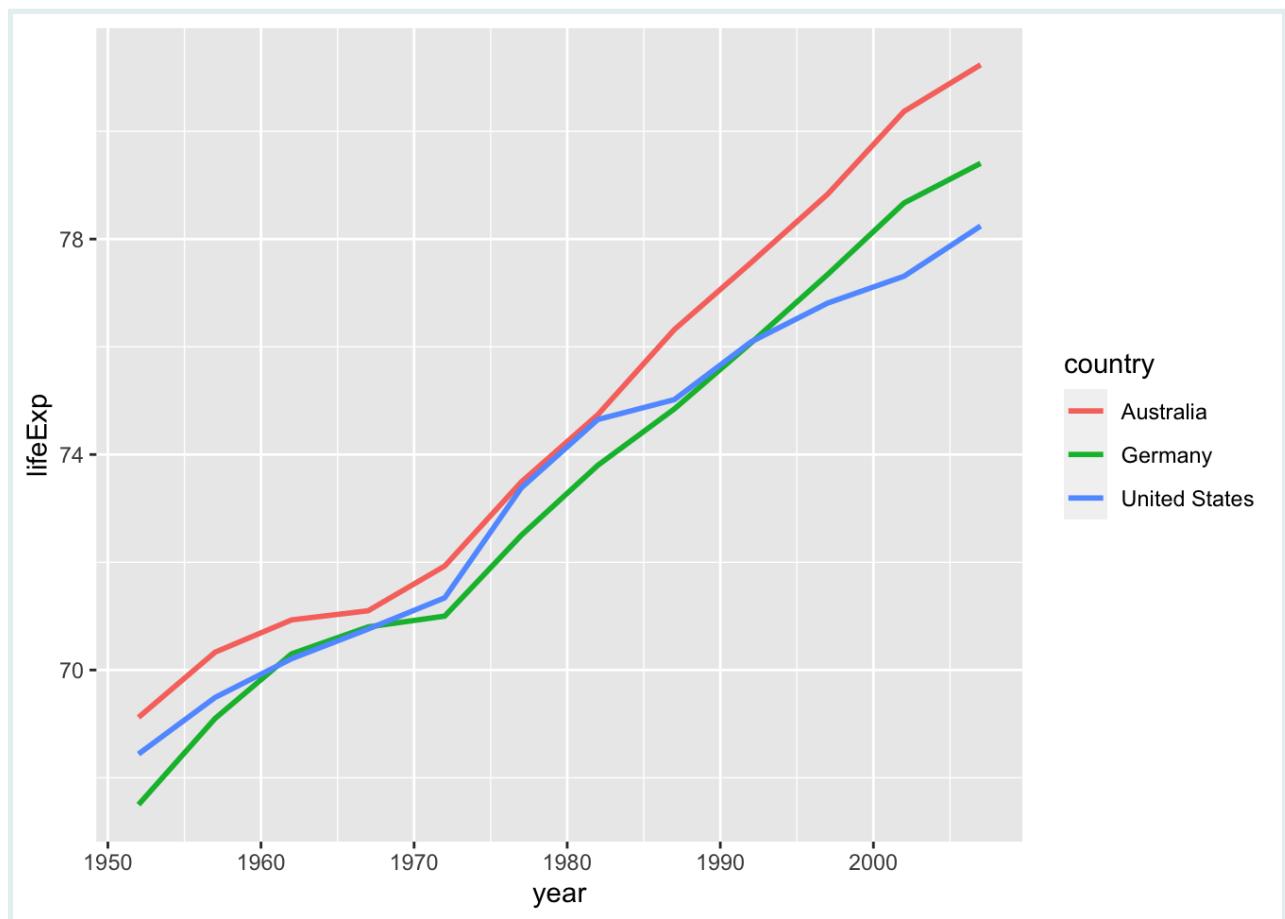


Don't worry about any fixed aesthetics, just make sure the mapping of data variables is the same.

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 we might want to transform the axis scaling for better visualization. We can customize these scales with the `scale_*` family of functions.

```

ggplot(data = <DATAFRAME>,
       mapping = aes(<VARS TO MAP>)) +
<GEOM_FUNCTION>() +
stat = <STAT>, position = <POSITION>) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>

```

Required

Not required, sensible defaults supplied

`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))`
- 
- `p + scale_radius(range = c(1,6))`
- `p + scale_size_area(max_size = 6)`

Scale breaks

Let's create a new subset of countries from `gapminder`, and this time we will plot changes in GDP over time.

```

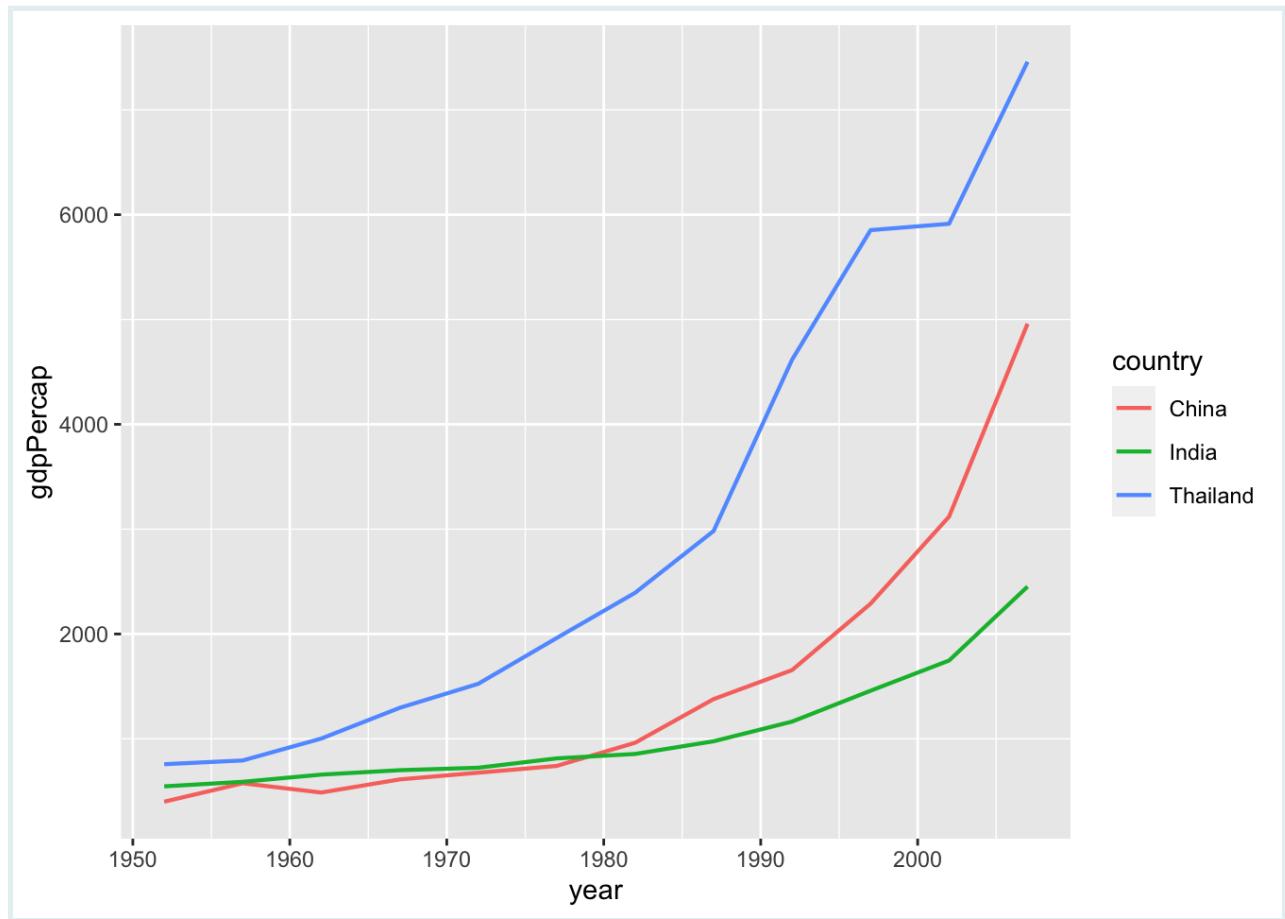
# Data subset to include India, China, and Thailand
gap_mini2 <- filter(gapminder,

```

```
"Thailand"))gap_mini2
```

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)
```

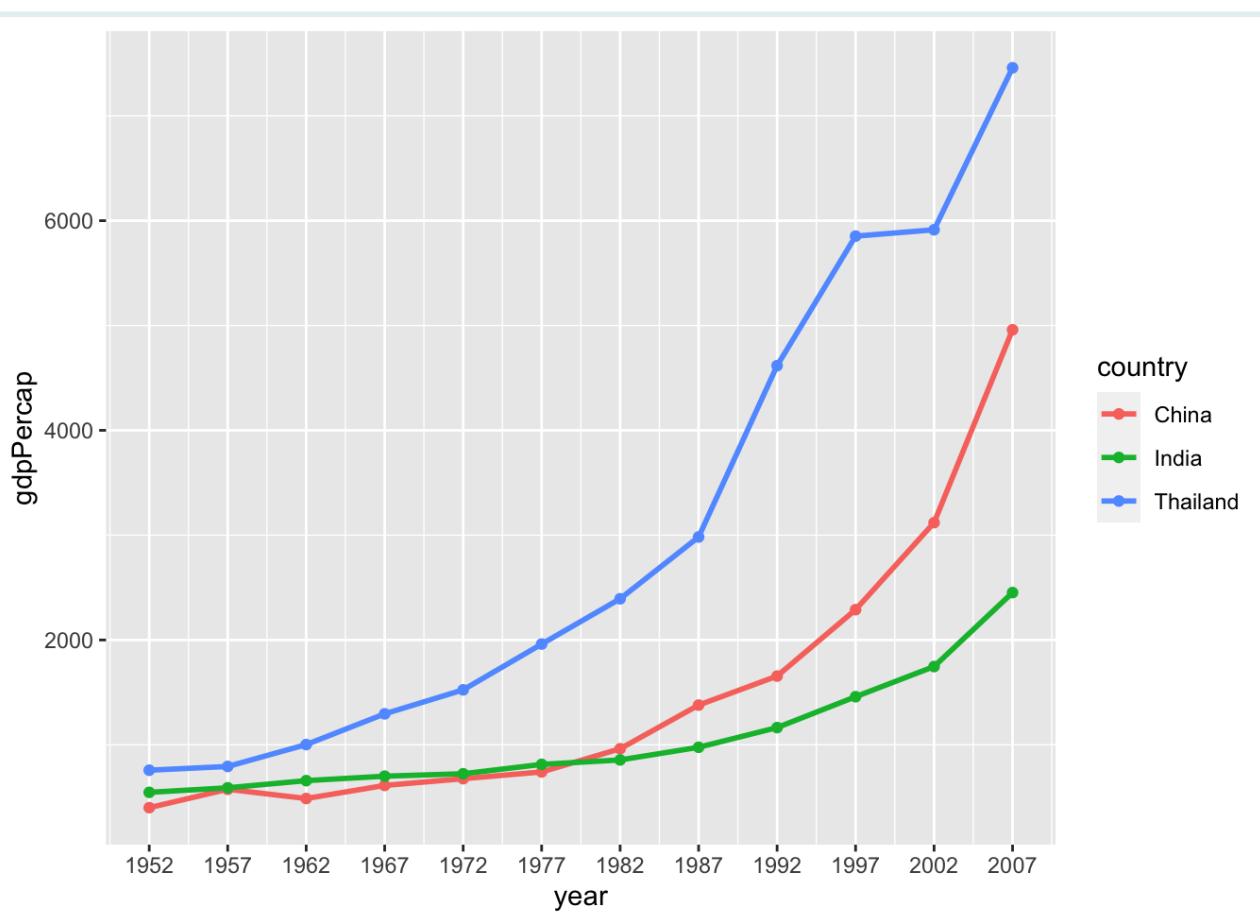
```
## [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

```
# 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:

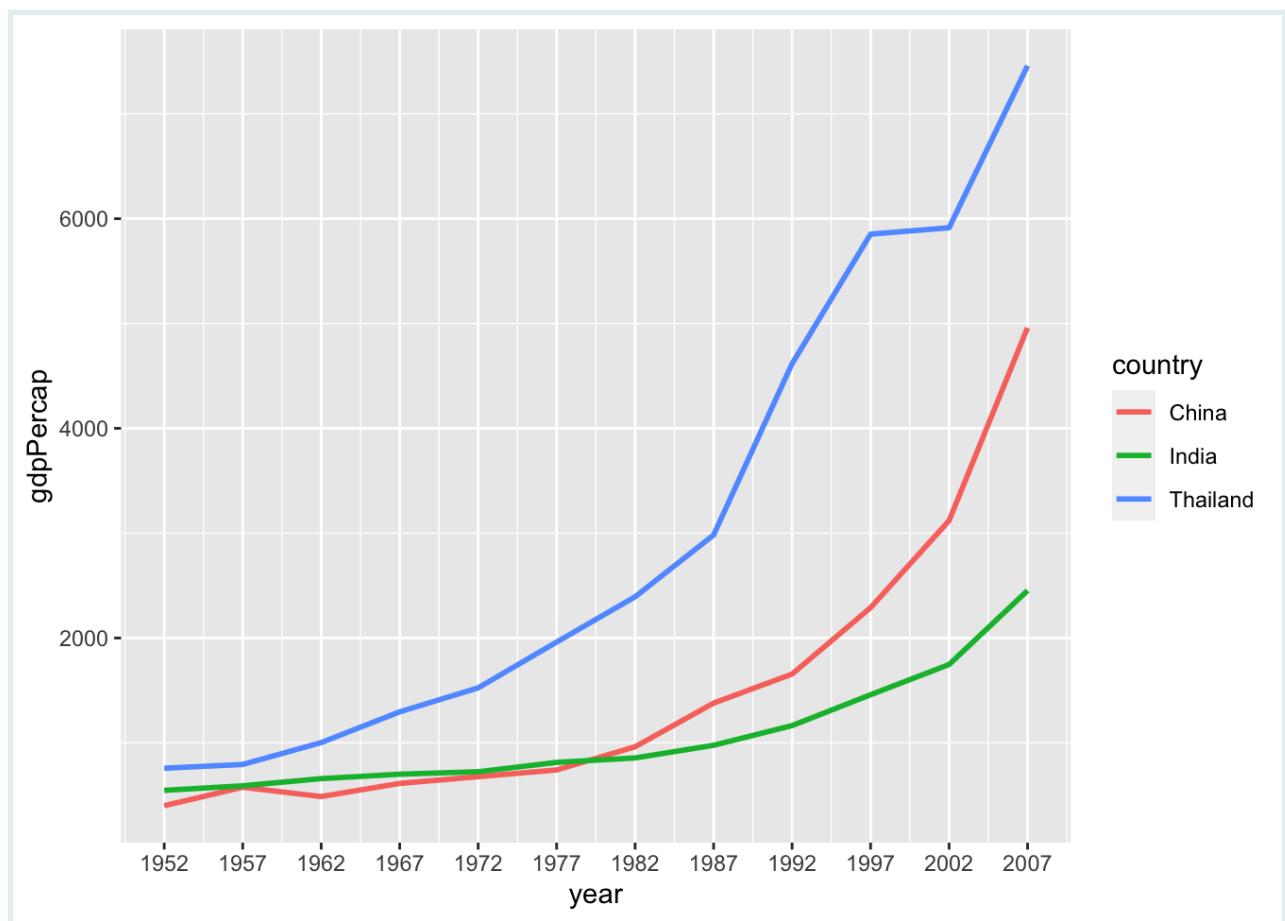
```
# Customize x-axis breaks with `scale_x_continuous(breaks = VECTOR)`  
ggplot(data = gap_mini2,  
       mapping = aes(x = year,  
                      y = gdpPerCap,  
                      color = country)) +  
  geom_line(size = 1) +  
  scale_x_continuous(breaks = seq(from = 1952, to = 2007, 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)
```



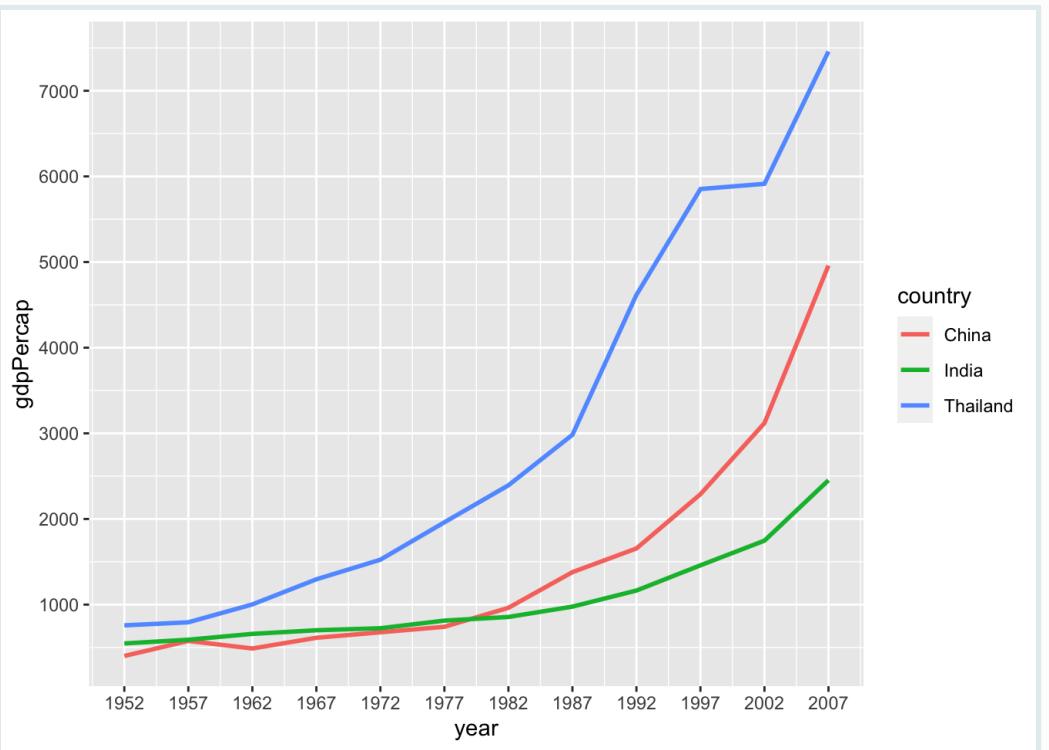
PRACTICE



(in RMD)

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:



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`:

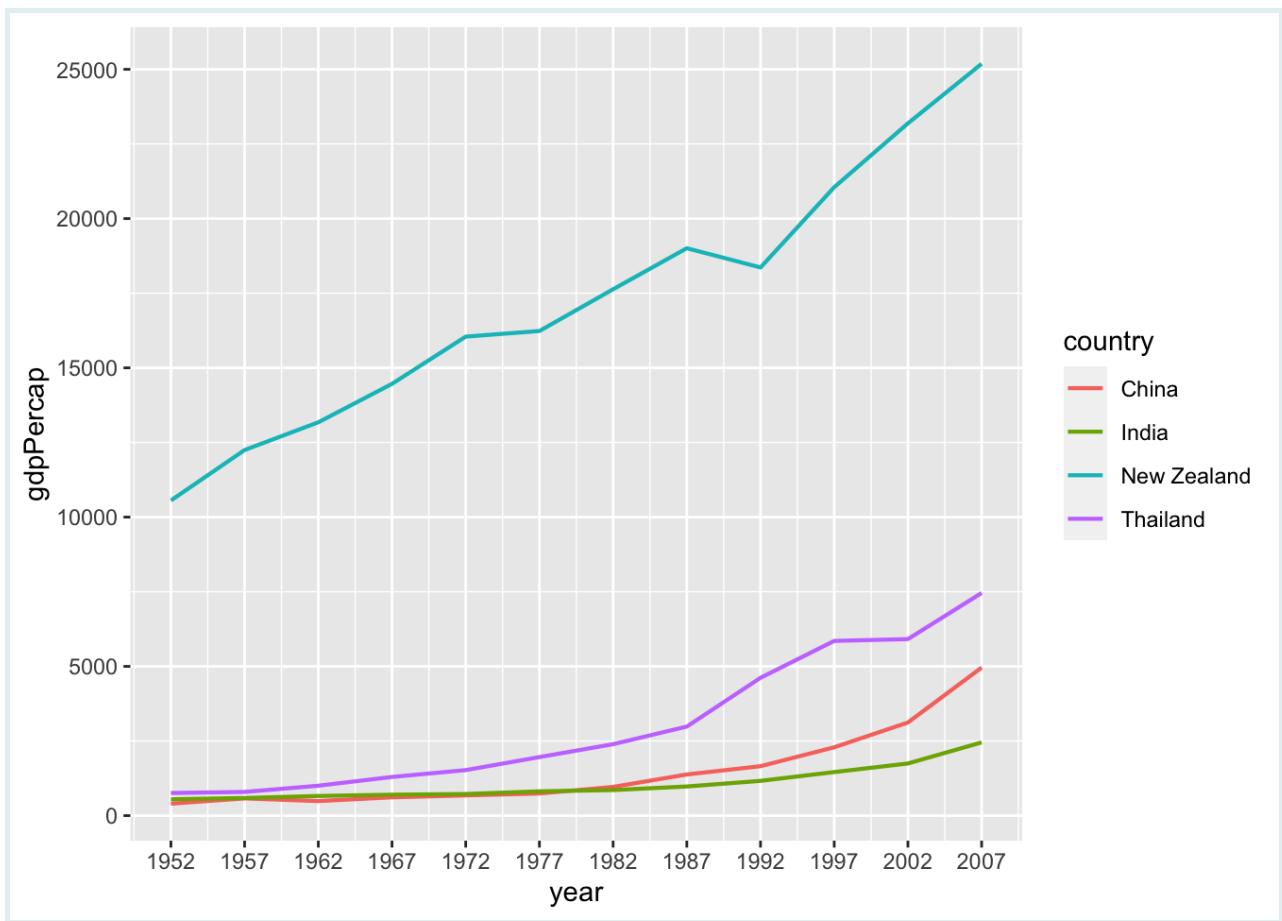
```
# Data subset to include India, China, Thailand, and New Zealand
gap_mini3 <- filter(gapminder,
  country %in% c("India",
    "China",
    "Thailand",
    "New Zealand"))

gap_mini3
```

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,
```

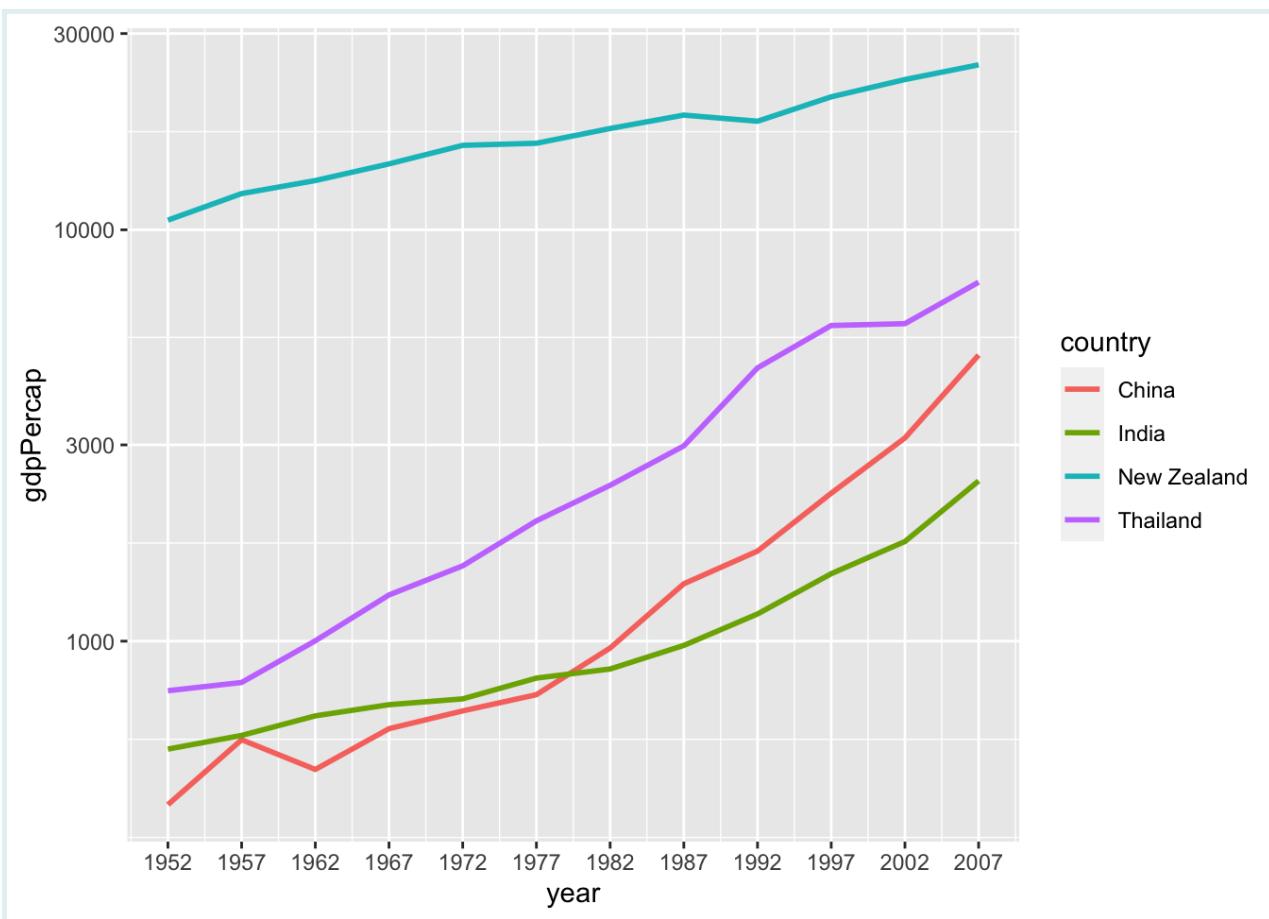
```
color = country)) + geom_line(size = 0.75) +
scale_x_continuous(breaks = gap_years)
```



The curves for India and China show an exponential increase in GDP per capita. However, the y-axes 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:

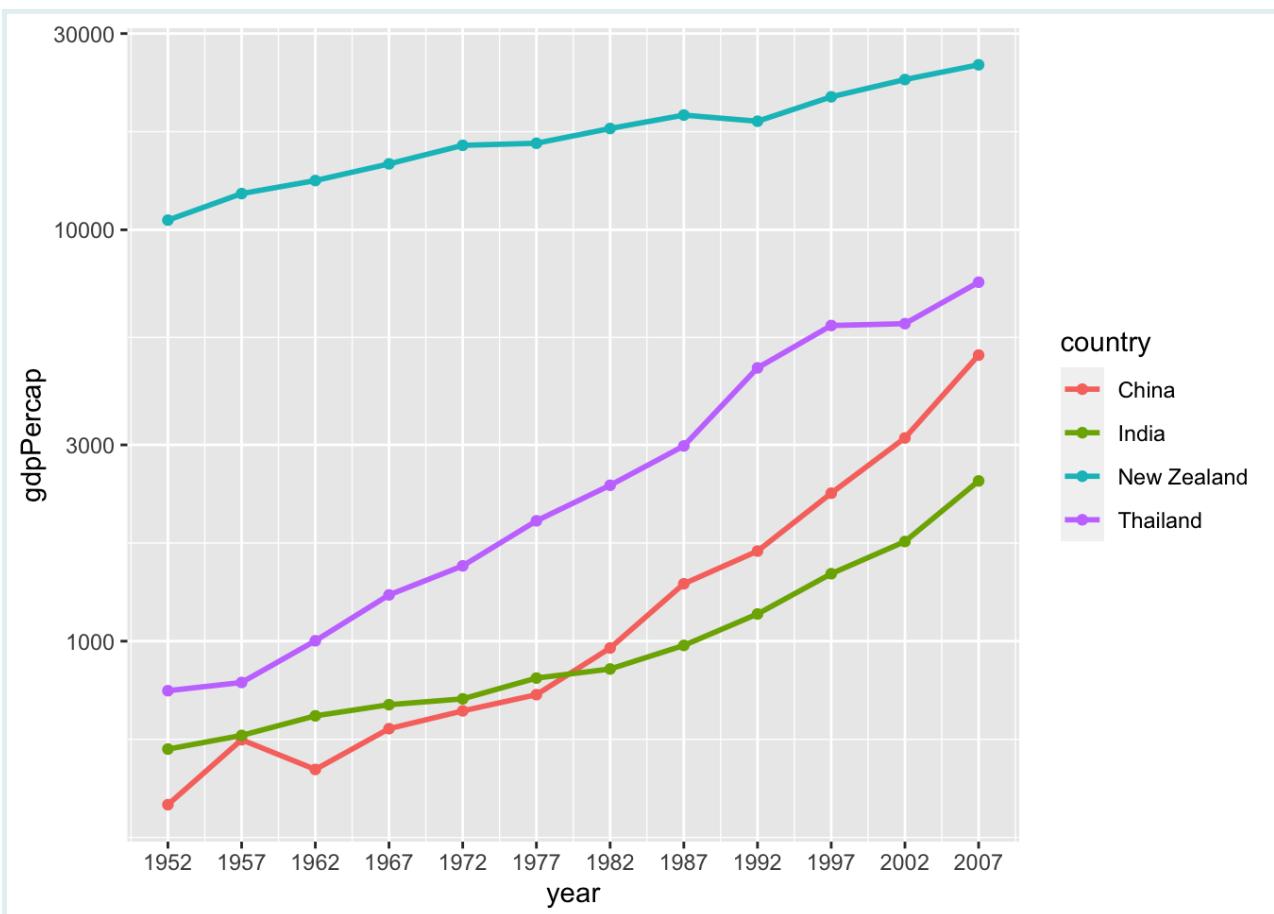
```
# Add scale_y_log10()
ggplot(data = gap_mini3,
       mapping = aes(x = year,
                      y = gdpPercap,
                      color = country)) +
  geom_line(size = 1) +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10()
```



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.

Labeling with `labs()`

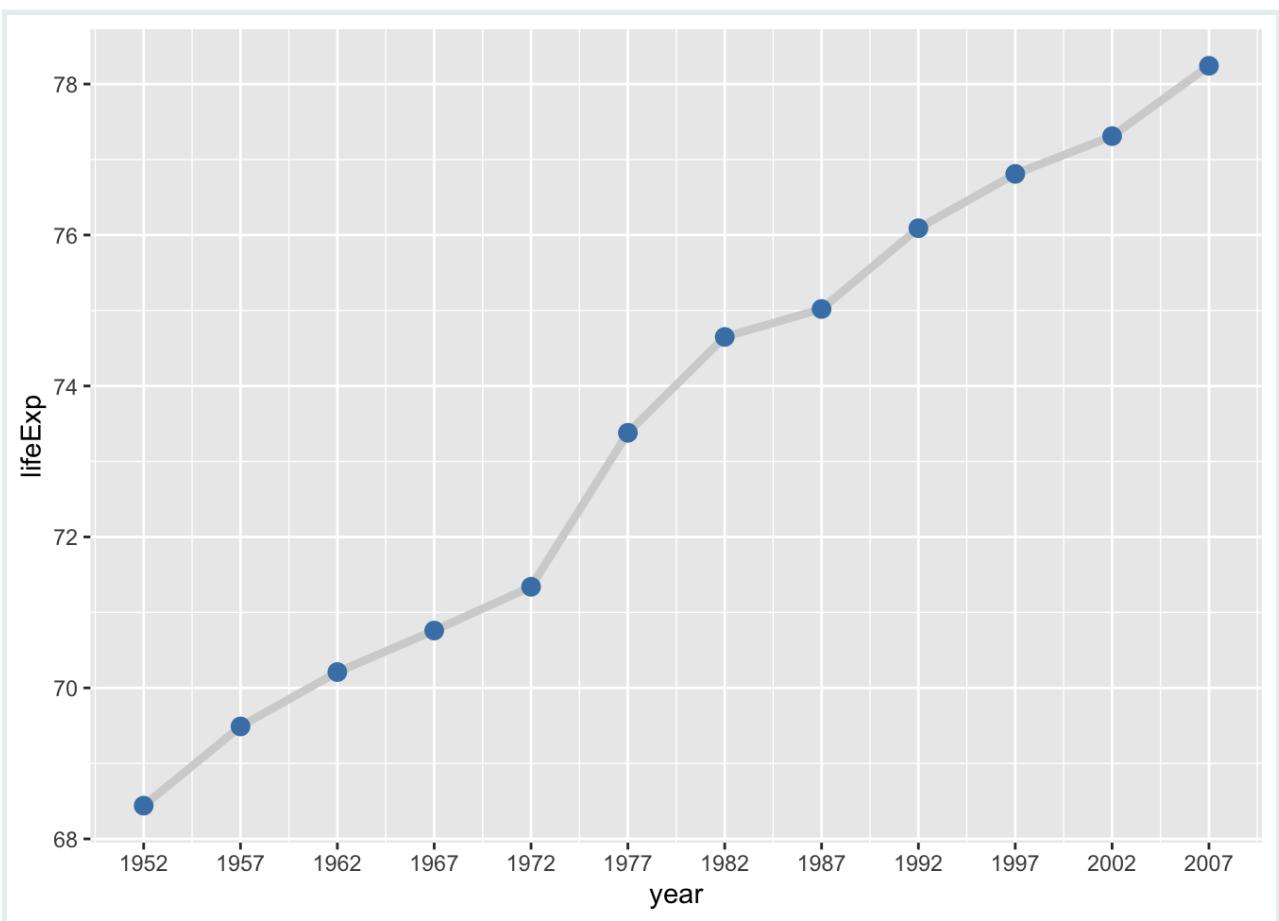
You can add labels to a plot with the `labs()` function. Arguments we can specify with the `labs()` function include:

- `title`: Change or add a title

- `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.

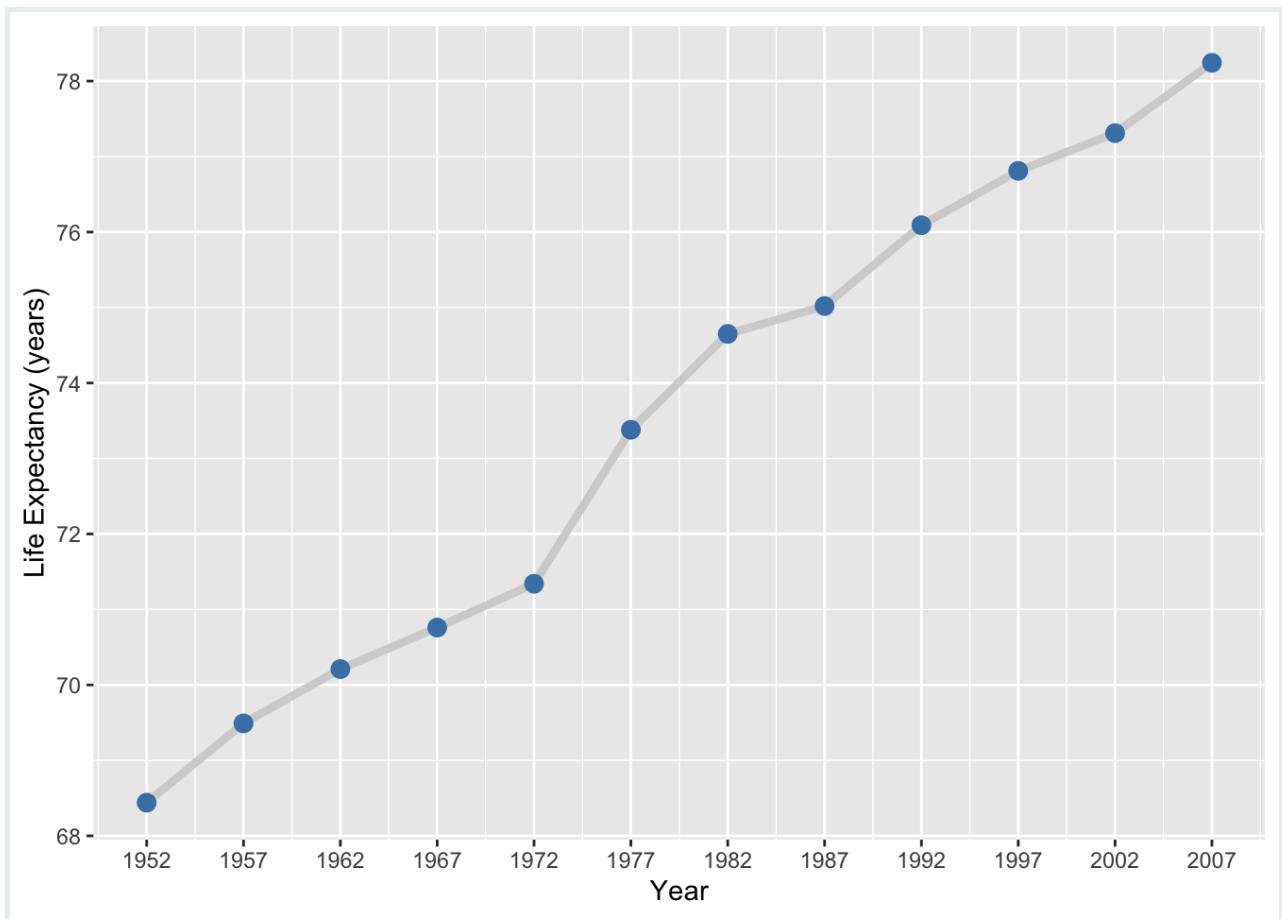
First we will add the `x` and `y` arguments to `labs()`, and change the axis titles from the default (variable name) to something more informative.

```
# 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)")

```

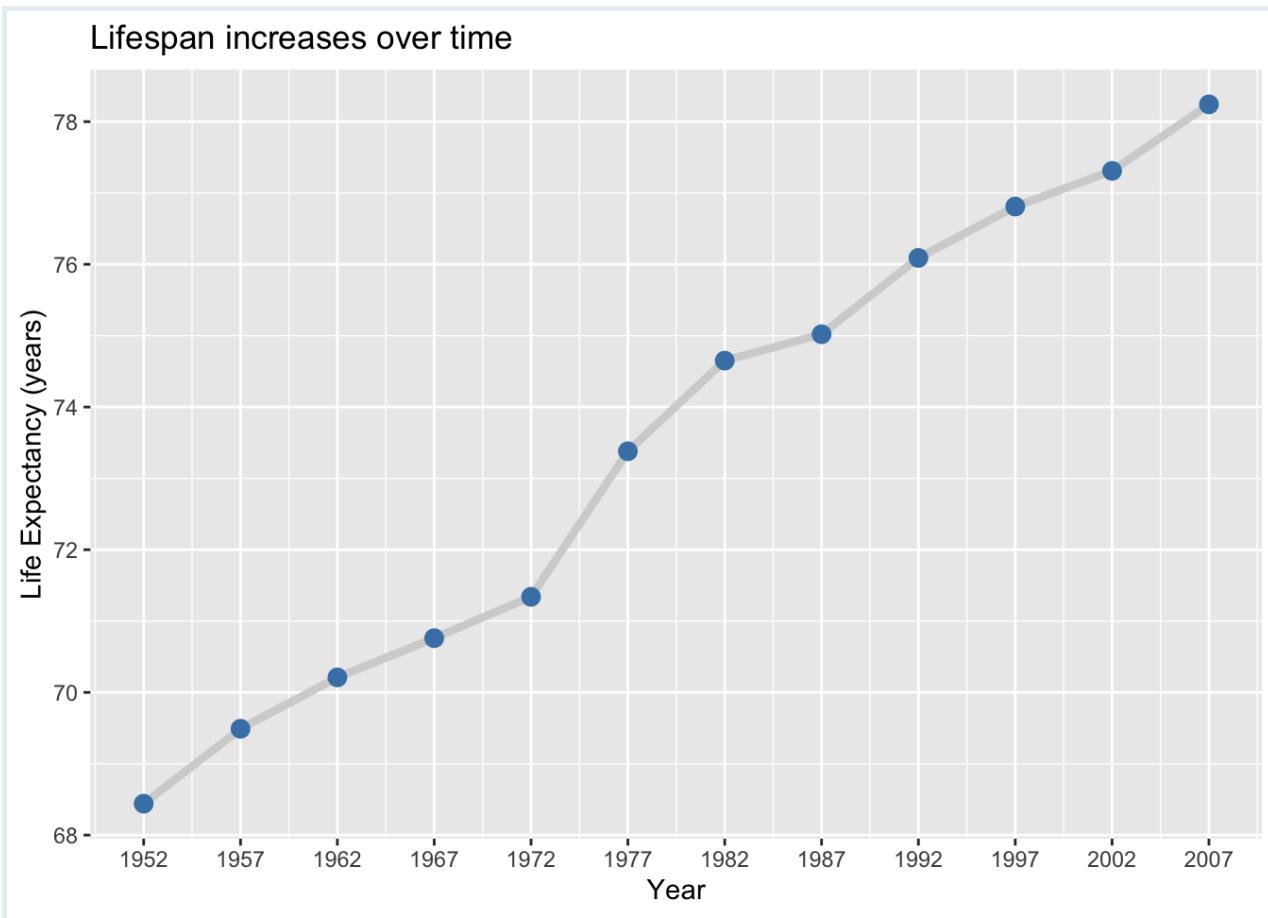


Next we supply a character string to the `title` argument to add large text above the plot.

```

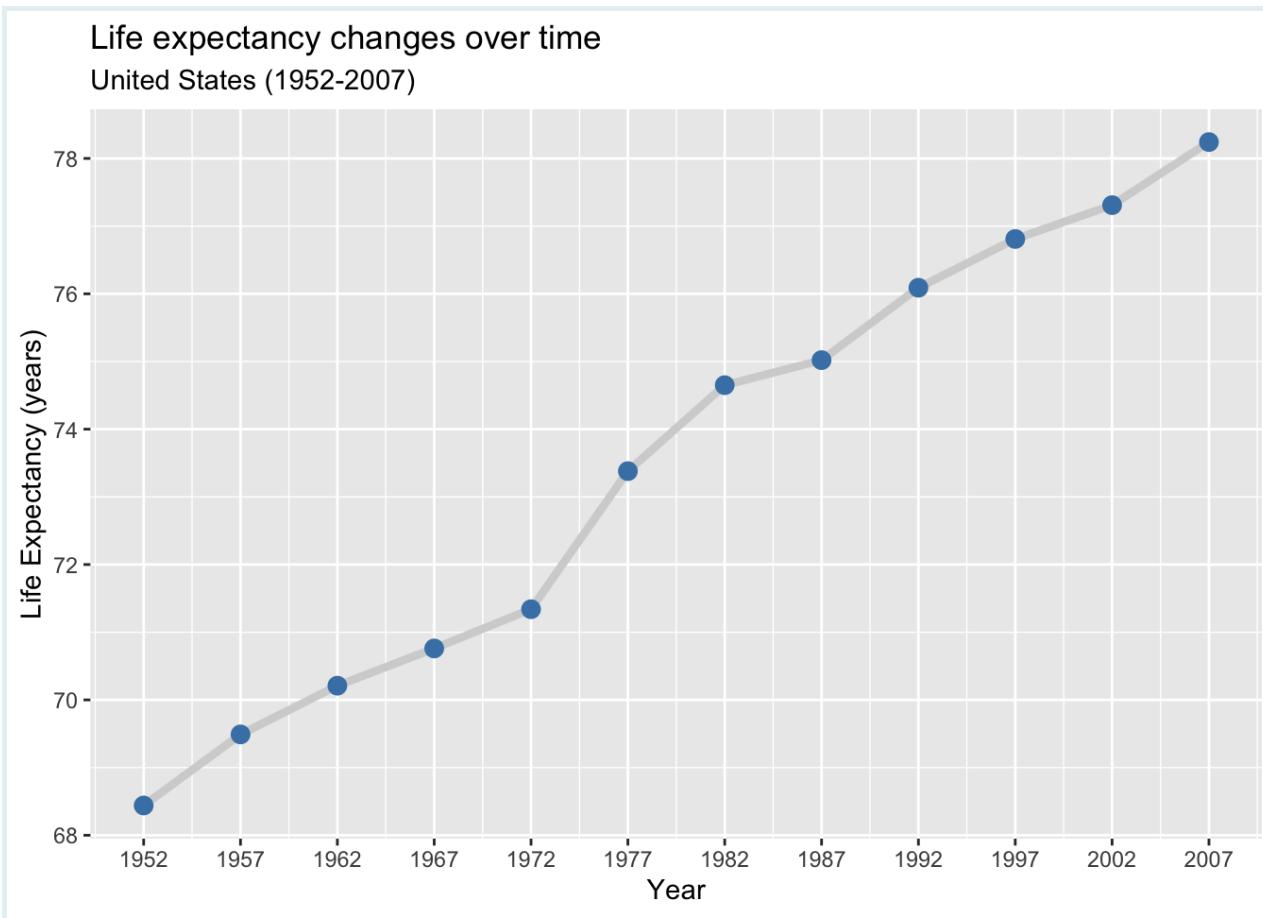
# Add main title: "Lifespan increases over time"
ggplot(data = gap_US,
       mapping = aes(x = year,
                      y = lifeExp)) +
  geom_line(size = 1.5,
            color = "lightgrey") +
  geom_point(size = 3,
            color = "steelblue") +
  scale_x_continuous(breaks = gap_years) +
  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)")
```

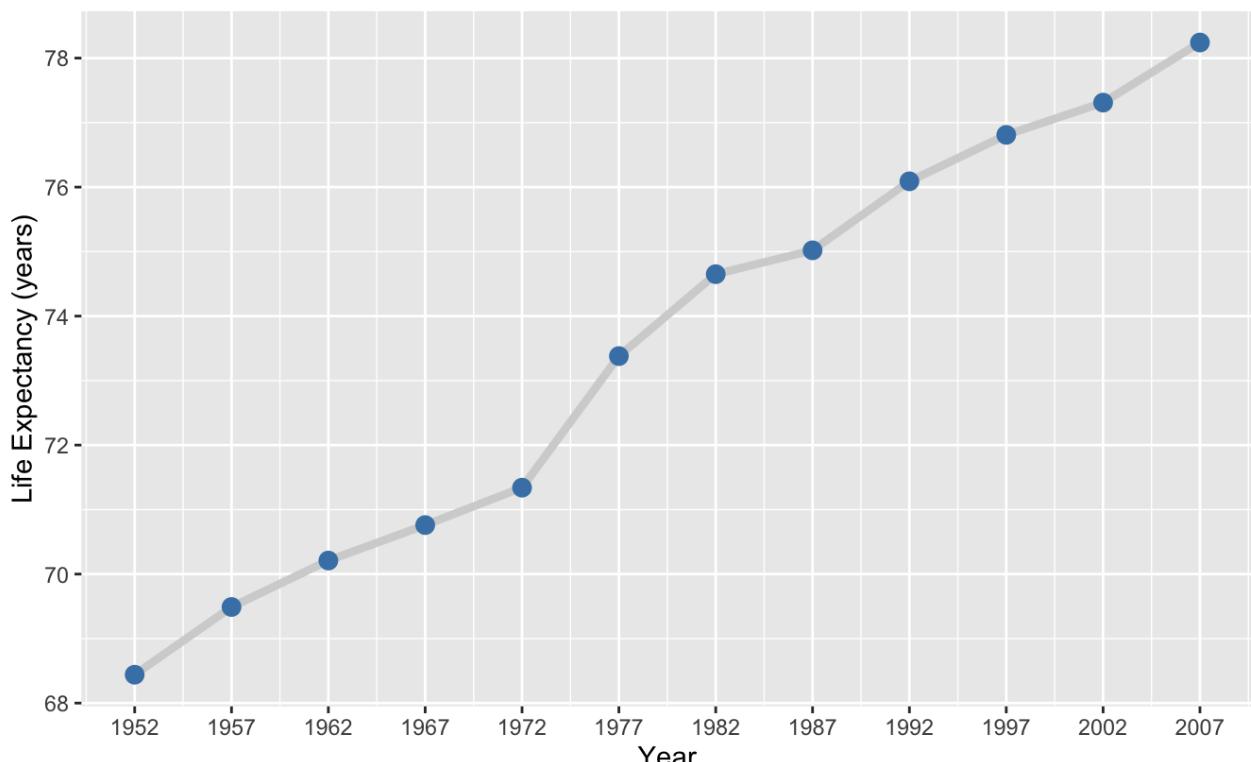


Finally, we can supply the caption argument to add small text to the bottom-right corner below the plot.

```
# 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/")
```

Life expectancy changes over time

United States (1952-2007)



Source: <http://www.gapminder.org/data/>

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.

CHALLENGE

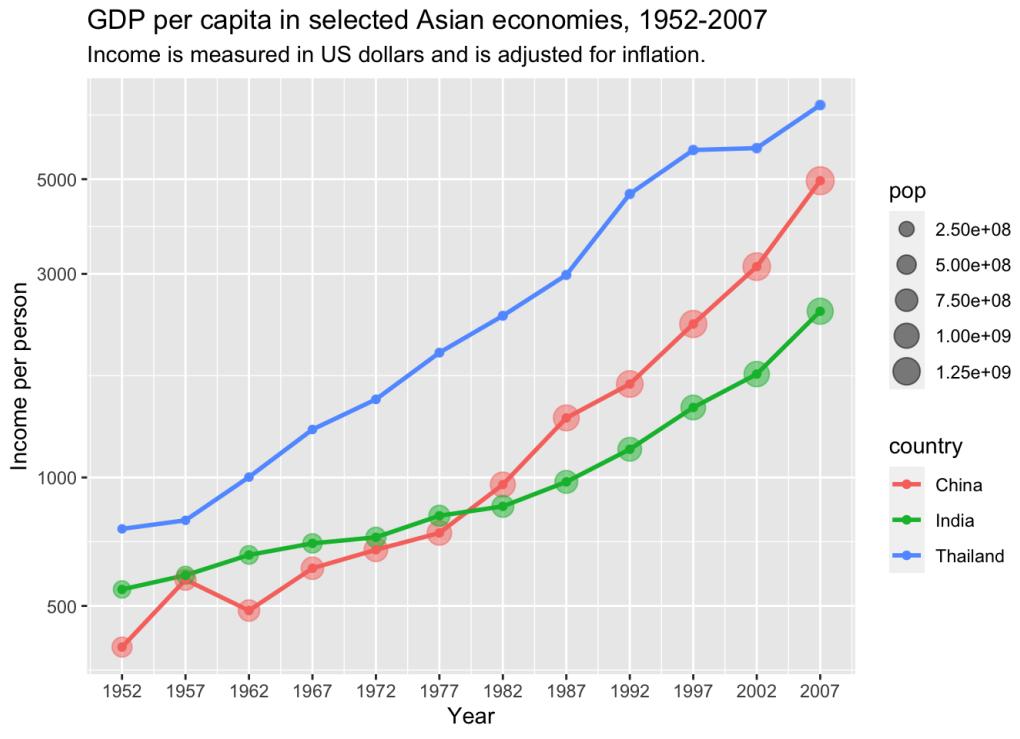


```
ggplot(data = gap_mini2,
       mapping = aes(x = year,
                      y = gdpPercap,
                      color = country)) +
  geom_line(size = 1) +
  geom_point(mapping = aes(size = pop),
             alpha = 0.5) +
  geom_point() +
  scale_x_continuous(breaks = gap_years) +
  scale_y_log10() +
  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.")

```



CHALLENGE



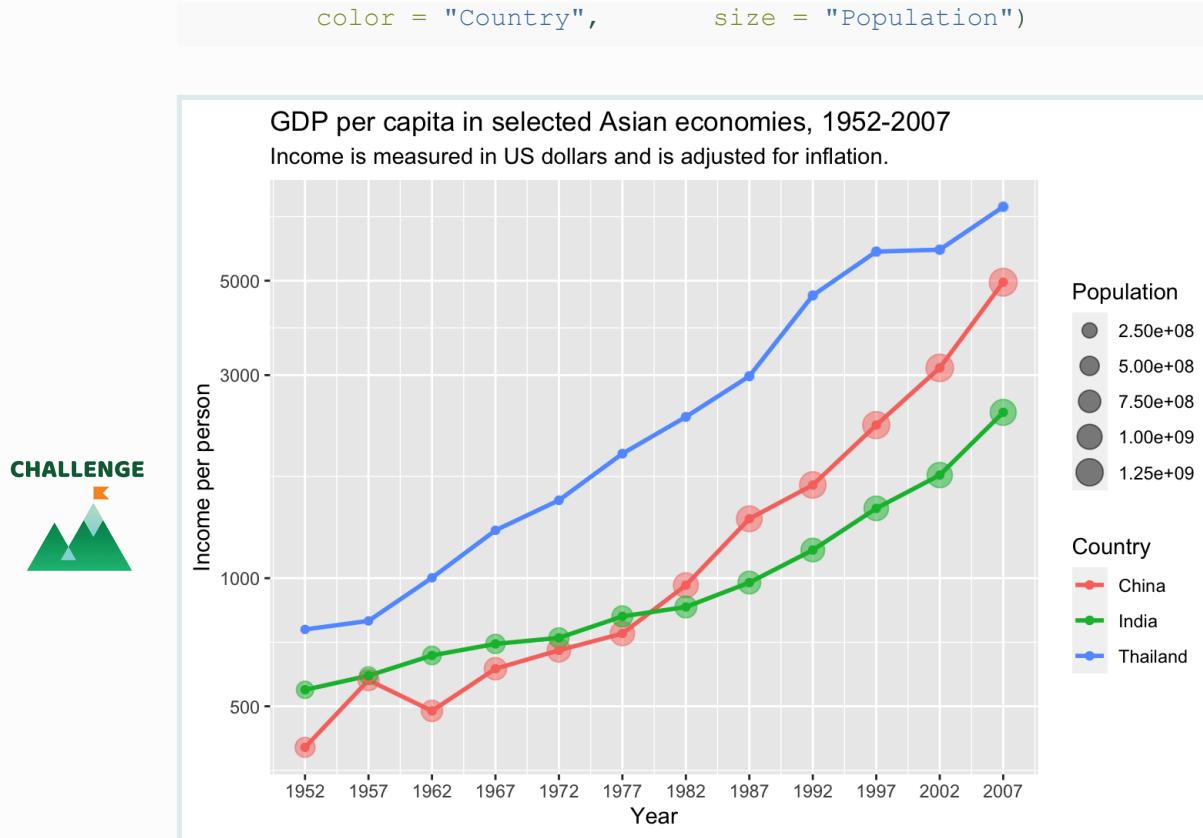
The default title of a legend or key is the name of the data variable it corresponds to. Here the color legend is titled `country`, and the size legend is titled `pop`.

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",
       color = "Country")

```

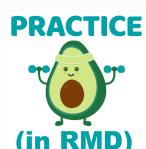


The same syntax can be used to edit legend titles for other aesthetic mappings. A common mistake is to use the variable name instead of the aesthetic name in `labs()`, so watch out for that!

Create a time series plot comparing the trends in GDP per capita 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



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.)

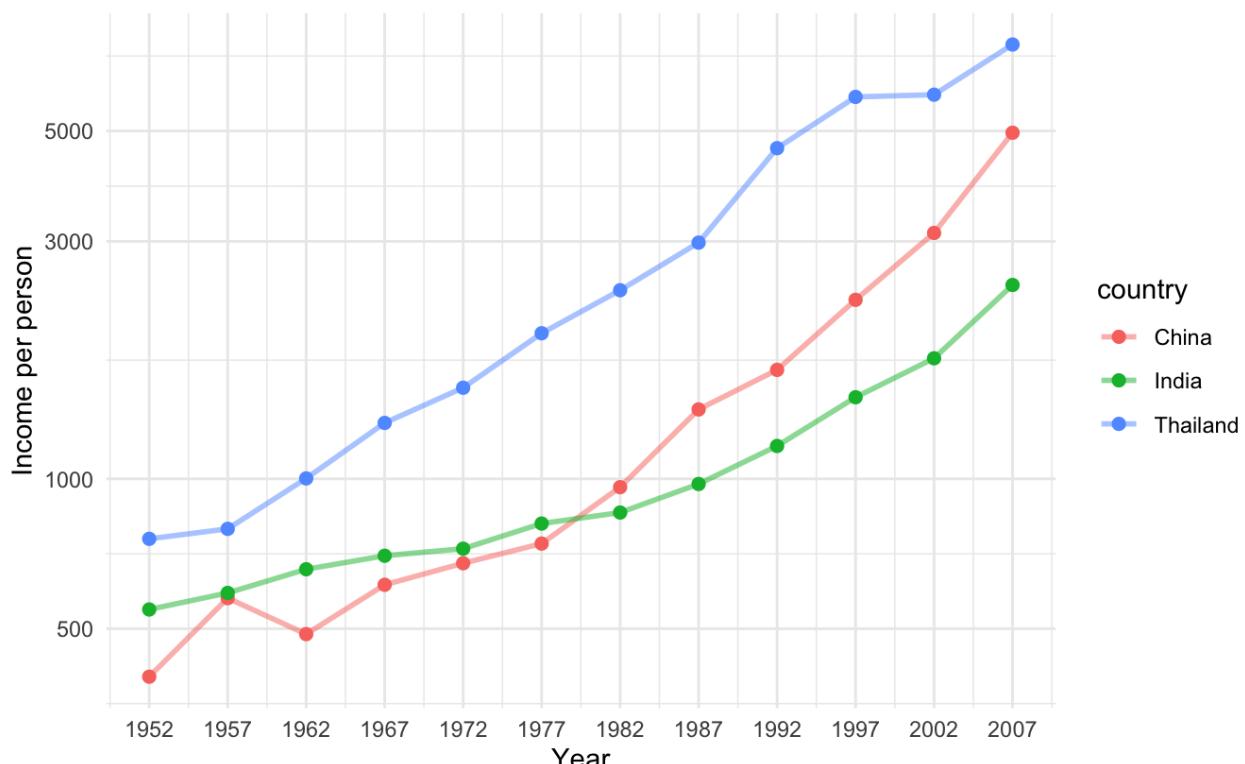
Preview: Themes

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()
```

GDP per capita in selected Asian economies, 1952-2007

Income is measured in US dollars and is adjusted for inflation.



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!

Contributors

The following team members contributed to this lesson:



JOY VAZ

R Developer and Instructor, the GRAPH Network
Loves doing science and teaching science



ADMIN TEAM

GRAPH Courses Administration Team

The GRAPH Courses team is building epidemiological training courses to enhance disease surveillance and data science for public health across the globe

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/

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.

