Research Design and Analysis

The Center for Computational Biomedicine

2023-06-16

Table of contents

Overview

Welcome!

In this module of the course, we will be learning how to use R to analyze data. This workbook will contain pre-class readings, in-class materials, and links to additional resources.

Most of this workbook has been adapted from The Epidemiologist R Handbook. You can find more details at the bottom of this page.

Why learn R?

As stated on the R project website, R is a programming language and environment for statistical computing and graphics. It is highly versatile, extendable, and community-driven.

R does not involve lots of pointing and clicking, and that's a good thing

The learning curve might be steeper than with other software, but with R, the results of your analysis do not rely on remembering a succession of pointing and clicking, but instead on a series of written commands, and that's a good thing! So, if you want to redo your analysis because you collected more data, you don't have to remember which button you clicked in which order to obtain your results; you just have to run your script again.

Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else who can give you feedback and spot mistakes.

Working with scripts forces you to have a deeper understanding of what you are doing, and facilitates your learning and comprehension of the methods you use.

R code is great for reproducibility

Reproducibility means that someone else (including your future self) can obtain the same results from the same dataset when using the same analysis code.

R integrates with other tools to generate manuscripts or reports from your code. If you collect more data, or fix a mistake in your dataset, the figures and the statistical tests in your manuscript or report are updated automatically.

An increasing number of journals and funding agencies expect analyses to be reproducible, so knowing R will give you an edge with these requirements.

R is interdisciplinary and extensible

With 10000+ packages[1] that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyse your data. For instance, R has packages for image analysis, GIS, time series, population genetics, and a lot more.

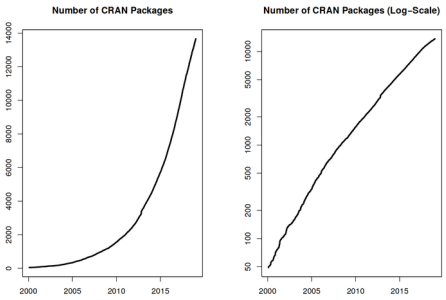


Figure 1: Exponential increase of the number of packages available on CRAN, the Comprehensive R Archive Network. From the R Journal, Volume 10/2, December 2018.

R works on data of all shapes and sizes

The skills you learn with R scale easily with the size of your dataset. Whether your dataset has hundreds or millions of lines, it won't make much difference to you.

R is designed for data analysis. It comes with special data structures and data types that make handling of missing data and statistical factors convenient.

R can connect to spreadsheets, databases, and many other data formats, on your computer or on the web.

R produces high-quality graphics

The plotting functionalities in R are extensive, and allow you to adjust any aspect of your graph to convey most effectively the message from your data.

[1] i.e. add-ons that confer R with new functionality, such as bioinformatics data analysis.

R has a large and welcoming community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow, or on the RStudio community. These broad user communities extend to specialised areas such as bioinformatics. One such subset of the R community is Bioconductor, a scientific project for analysis and comprehension "of data from current and emerging biological assays." Another example is R-Ladies, a worldwide organization whose mission is to promote gender diversity in the R community. It is one of the largest organizations of R users and likely has a chapter near you!

Not only is R free, but it is also open-source and cross-platform

Anyone can inspect the source code to see how R works. Because of this transparency, there is less chance for mistakes, and if you (or someone else) find some, you can report and fix bugs.

Sources and References

Most of the materials in this workbook have been adapted from The Epidemiologist R Handbook with some changes made and materials incorporated from other sources. These additional sources are attributed in the chapters they are a part of.

Part I

Session 0: R basics

1 Installing R and RStudio

1.1 Mac Users

1.1.1 To install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for (Mac) OS X" link at the top of the page.
5. Click on the file containing the latest version of R under "Files."
6. Save the .pkg file, double-click it to open, and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

1.1.2 To install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "DOWNLOAD" in the upper right corner.
3. Download the Free version of RStudio Desktop.
4. Save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

1.2 Windows Users

1.2.1 To install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for Windows" link at the top of the page.
5. Click on the "install R for the first time" link at the top of the page.
6. Click "Download R for Windows" and save the executable file somewhere on your computer. Run the .exe file and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

1.2.2 To install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "DOWNLOAD" in the upper right corner.
3. Download the Free version of RStudio Desktop.
4. Save the executable file. Run the .exe file and follow the installation instructions.

Permissions
Note that you should install R and RStudio to a drive where you have read and write permissions. Otherwise, your ability to install R packages (a frequent occurrence) will be impacted. If you encounter problems, try opening RStudio by right-clicking the icon and selecting "Run as administrator". Other tips can be found in the page [R on network drives].

How to update R and RStudio

Your version of R is printed to the R Console at start-up. You can also run `sessionInfo()`.

To update R, go to the website mentioned above and re-install R. Be aware that the old R version will still exist in your computer. You can temporarily run an older version (older "installation") of R by clicking "Tools" -> "Global Options" in RStudio and choosing an R version. This can be useful if you want to use a package that has not been updated to work on the newest version of R.

To update RStudio, you can go to the website above and re-download RStudio. Another option is to click "Help" -> "Check for Updates" within RStudio, but this may not show the very latest updates.

1.3 Reference

Instructions adapted from guide developed by HMS Research computing and Chapter 3 of the The Epidemiologist R Handbook.

2 Introduction to RStudio

2.1 What is RStudio?

RStudio is freely available open-source Integrated Development Environment (IDE). RStudio provides an environment with many features to make using R easier and is a great alternative to working on R in the terminal.

You can see the complete RStudio user-interface cheatsheet (PDF) here

2.2 Creating a new project directory in RStudio

Let's create a new project directory for the Research and Design course.

1. Open RStudio
2. Go to the `File` menu and select `New Project`.
3. In the `New Project` window, choose `New Directory`. Then, choose `New Project`. Name your new directory whatever you want and then "Create the project as subdirectory of:" the Desktop (or location of your choice).
4. Click on `Create Project`.
5. After your project is completed, if the project does not automatically open in RStudio, then go to the `File` menu, select `Open Project`, and choose `[your project name].Rproj`.
6. When RStudio opens, you will see three panels in the window.
7. Go to the `File` menu and select `New File`, and select `R Script`. The RStudio interface should now look like the screenshot below.

TIP: If your RStudio displays only one left pane it is because you have no scripts open yet.

2.2.1 What is a project in RStudio?

It is simply a directory that contains everything related your analyses for a specific project. RStudio projects are useful when you are working on context- specific analyses and you wish to keep them separate. When creating a project in RStudio you associate it with a working directory of your choice (either an existing one, or a new one). A `. RProj file` is created

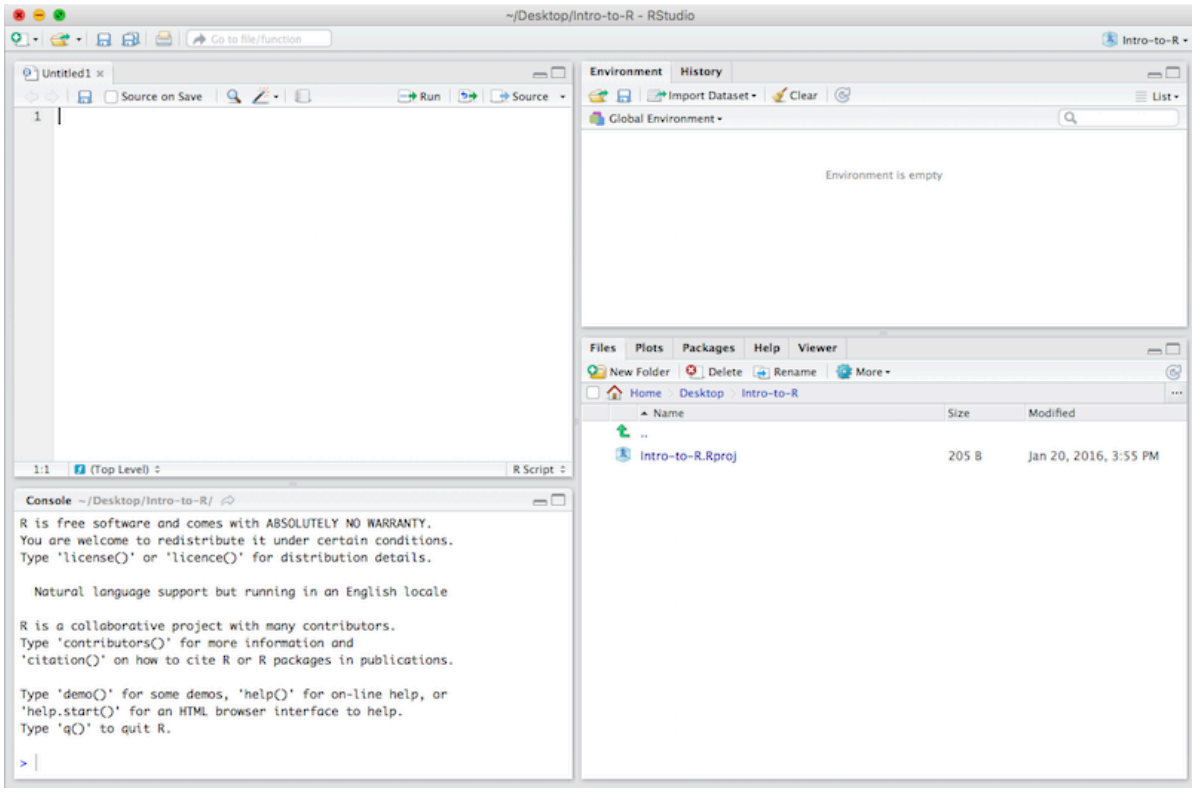


Figure 2.1: RStudio interface

within that directory and that keeps track of your command history and variables in the environment. The `. RProj file` can be used to open the project in its current state but at a later date.

When a project is **(re) opened** within RStudio the following actions are taken:

- A new R session (process) is started
- The .RData file in the project's main directory is loaded, populating the environment with any objects that were present when the project was closed.
- The .Rhistory file in the project's main directory is loaded into the RStudio History pane (and used for Console Up/Down arrow command history).
- The current working directory is set to the project directory.
- Previously edited source documents are restored into editor tabs
- Other RStudio settings (e.g. active tabs, splitter positions, etc.) are restored to where they were the last time the project was closed.

Information adapted from RStudio Support Site

2.3 RStudio Interface

The RStudio interface has four main panels:

1. **Console**: where you can type commands and see output. *The console is all you would see if you ran R in the command line without RStudio.*
2. **Script editor**: where you can type out commands and save to file. You can also submit the commands to run in the console.
3. **Environment/History**: environment shows all active objects and history keeps track of all commands run in console
4. **Files/Plots/Packages/Help**

2.4 Organizing your working directory & setting up

2.4.1 Viewing your working directory

Before we organize our working directory, let's check to see where our current working directory is located by typing into the console:

```
getwd()
```

Your working directory should be the `Intro-to-R` folder constructed when you created the project. The working directory is where RStudio will automatically look for any files you bring in and where it will automatically save any files you create, unless otherwise specified.

You can visualize your working directory by selecting the `Files` tab from the **Files/Plots/Packages/Help** window.

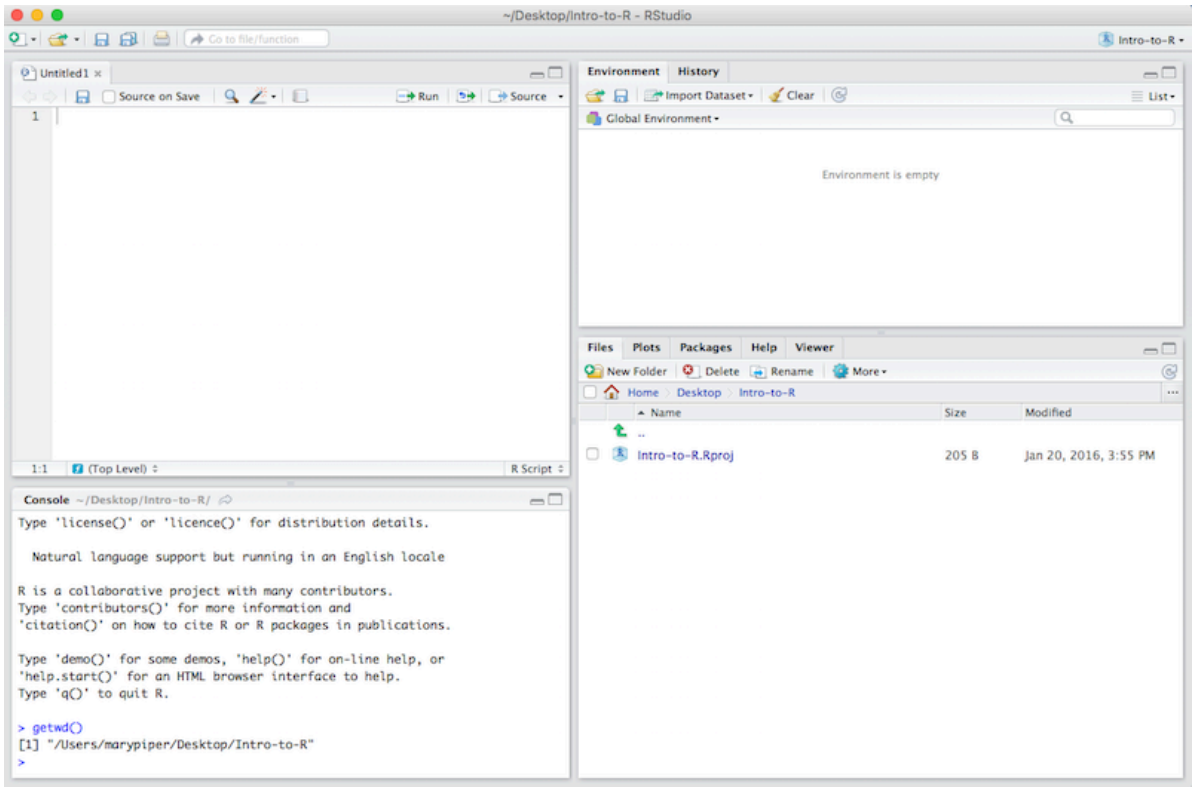


Figure 2.2: Viewing your working directory

If you wanted to choose a different directory to be your working directory, you could navigate to a different folder in the `Files` tab, then, click on the `More` dropdown menu and select `Set As Working Directory`.

2.4.2 Structuring your working directory

To organize your working directory for a particular analysis, you typically want to separate the original data (raw data) from intermediate datasets. For instance, you may want to create a `data/` directory within your working directory that stores the raw data, and have a `results/` directory for intermediate datasets and a `figures/` directory for the plots you will generate.

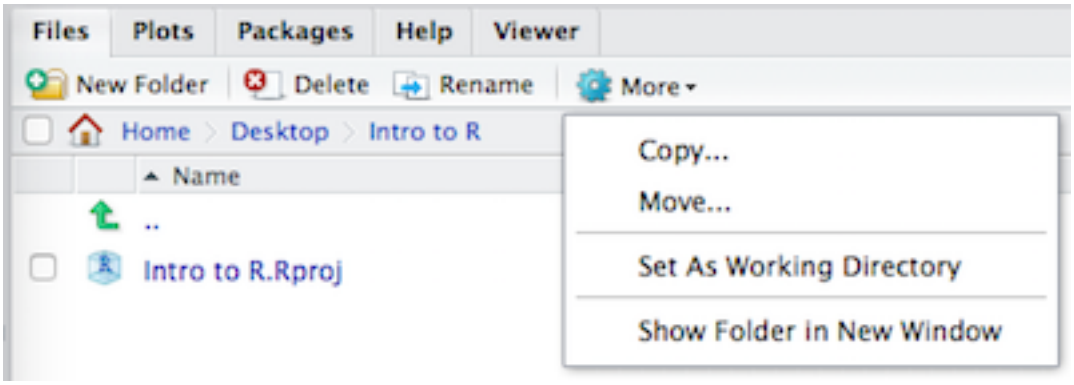


Figure 2.3: Setting your working directory

Let's create these three directories within your working directory by clicking on `New Folder` within the `Files` tab.

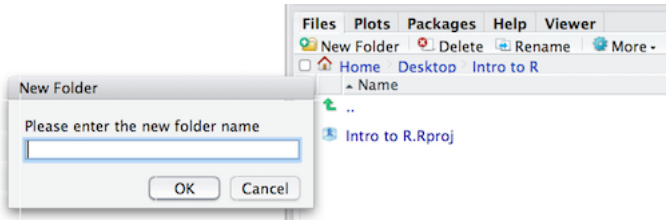


Figure 2.4: Structuring your working directory

When finished, your working directory should look like:

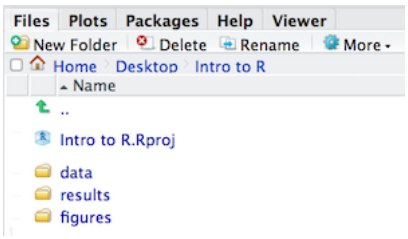


Figure 2.5: Your organized working directory

2.4.3 Setting up

This is more of a housekeeping task. We will be writing long lines of code in our script editor and want to make sure that the lines "wrap" and you don't have to scroll back and forth to look at your long line of code.

Click on "Tools" at the top of your RStudio screen and click on "Global Options" in the pull down menu.

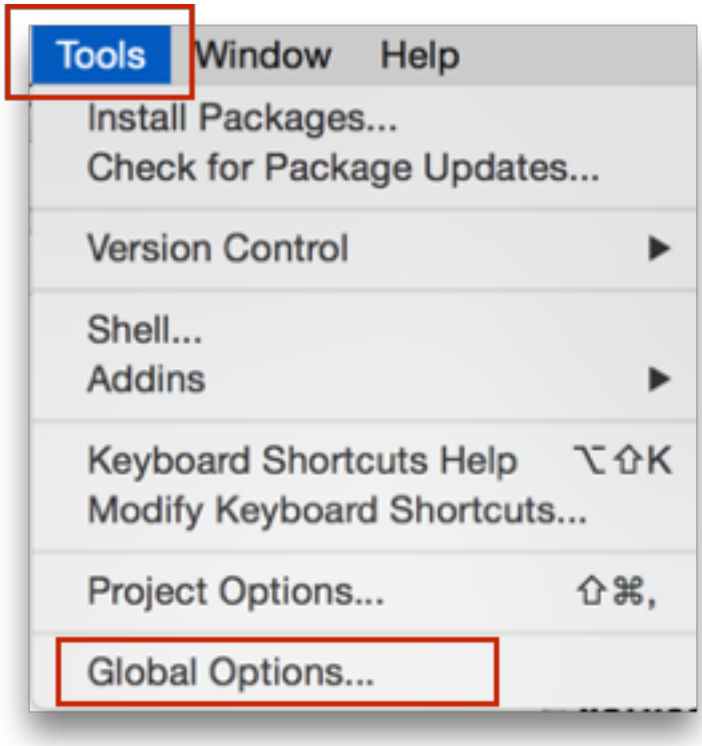


Figure 2.6: options

On the left, select "Code" and put a check against "Soft-wrap R source files". Make sure you click the "Apply" button at the bottom of the Window before saying "OK".

2.5 Interacting with R

Now that we have our interface and directory structure set up, let's start playing with R! There are **two main ways** of interacting with R in RStudio: using the **console** or by using **script editor** (plain text files that contain your code).

2.5.1 Console window

The **console window** (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do, and where it will show the results of a command. You can type commands directly into the console, but they will be forgotten when you close the session.

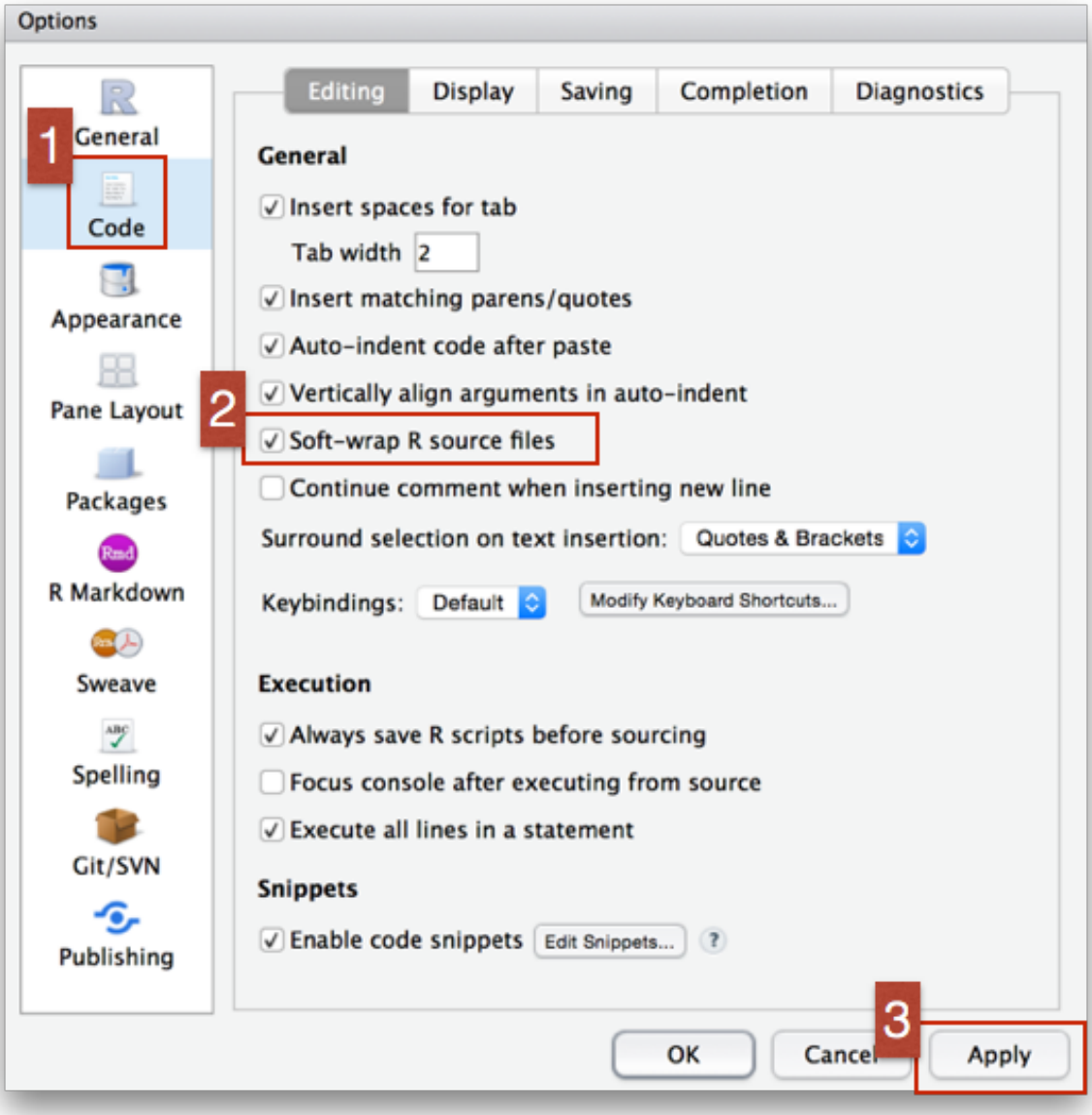


Figure 2.7: wrap_options

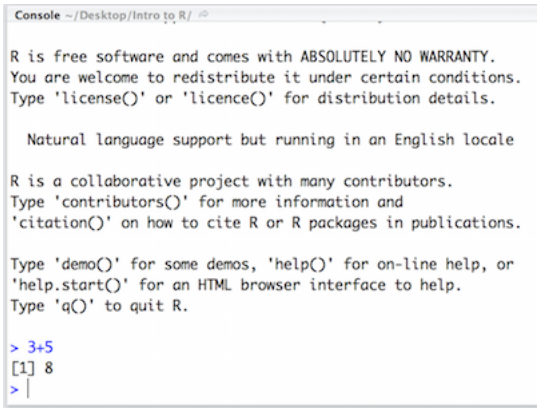


Figure 2.8: Running in the console

2.5.2 Script editor

Best practice is to enter the commands in the **script editor**, and save the script. You are encouraged to comment liberally to describe the commands you are running using `#`. This way, you have a complete record of what you did, you can easily show others how you did it and you can do it again later on if needed.

The Rstudio script editor allows you to 'send' the current line or the currently highlighted text to the R console by clicking on the `Run` button in the upper-right hand corner of the script editor. Alternatively, you can run by simply pressing the `Ctrl` and `Enter` keys at the same time as a shortcut.

Now let's try entering commands to the **script editor** and using the comments character `#` to add descriptions and highlighting the text to run:

```
# Session 1
# Feb 3, 2023

# Interacting with R

# I am adding 3 and 5.
3+5
```

You should see the command run in the console and output the result.

What happens if we do that same command without the comment symbol `#`? Re-run the command after removing the # sign in the front:

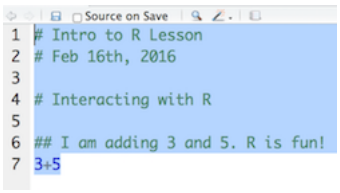


Figure 2.9: Running in the script editor

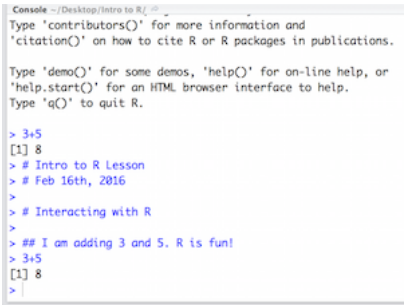


Figure 2.10: Script editor output

```
I am adding 3 and 5. R is fun!
3+5
```

Now R is trying to run that sentence as a command, and it doesn't work. We get an error in the console *"Error: unexpected symbol in"I am" means that the R interpreter did not know what to do with that command."*

2.5.3 Console command prompt

Interpreting the command prompt can help understand when R is ready to accept commands. Below lists the different states of the command prompt and how you can exit a command:

Console is ready to accept commands: `>`.

If R is ready to accept commands, the R console shows a `>` prompt.

When the console receives a command (by directly typing into the console or running from the script editor (`Ctrl-Enter`), R will try to execute it.

After running, the console will show the results and come back with a new `>` prompt to wait for new commands.

Console is waiting for you to enter more data: `+`.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. Often this can be due to you having not 'closed' a parenthesis or quotation.

Escaping a command and getting a new prompt: `esc`

If you're in Rstudio and you can't figure out why your command isn't running, you can click inside the console window and press `esc` to escape the command and bring back a new prompt `>`.

2.5.4 Keyboard shortcuts in RStudio

In addition to some of the shortcuts described earlier in this lesson, we have listed a few more that can be helpful as you work in RStudio.

Shortcuts Table

Some very useful keyboard shortcuts are below. See all the keyboard shortcuts for Windows, Max, and Linux in the second page of this RStudio user interface cheatsheet.

Windows/Linux	Mac	Action
Esc	Esc	Interrupt current command (useful if you accidentally ran an incomplete command and cannot escape seeing "+" in the R console)
Ctrl+s	Cmd+s	Save (script)
Tab	Tab	Auto-complete
Ctrl + Enter	Cmd + Enter	Run current line(s)/selection of code
Ctrl + Shift + C	Cmd + Shift + c	comment/uncomment the highlighted lines
Alt + -	Option + -	Insert `<-`
Ctrl + Shift + m	Cmd + Shift + m	Insert `%>%`
Ctrl + l	Cmd + l	Clear the R console
Ctrl + Alt + b	Cmd + Option + b	Run from start to current line
Ctrl + Alt + t	Cmd + Option + t	Run the current code section (R Markdown)
Ctrl + Alt + i	Cmd + Shift + r	Insert code chunk (into R Markdown)

Ctrl + Alt + c	Cmd + Option + c	Run current code chunk (R Markdown)
up/down arrows in R console	Same	Toggle through recently run commands
Shift + up/down arrows in script	Same	Select multiple code lines
Ctrl + f	Cmd + f	Find and replace in current script
Ctrl + Shift + f	Cmd + Shift + f	Find in files (search/replace across many scripts)
Alt + l	Cmd + Option + l	Fold selected code
Shift + Alt + l	Cmd + Shift + Option+l	Unfold selected code

Take advantage of auto-complete

Use your Tab key when typing to engage RStudio's auto-complete functionality. This can prevent spelling errors. Press Tab while typing to produce a drop-down menu of likely functions and objects, based on what you have typed so far.

2.6 R syntax

Now that we know how to talk with R via the script editor or the console, we want to use R for something more than adding numbers. To do this, we need to know more about the R syntax.

The main "parts of speech" in R (syntax) include:

- the **comments #** and how they are used to document function and its content
- **variables** and **functions**
- the **assignment operator <-**
- the **=** for **arguments** in functions

We will go through each of these "parts of speech" in more detail, starting with the assignment operator.

2.7 Assignment operator

To do useful and interesting things in R, we need to assign *values* to *variables* using the assignment operator, `<-`. For example, we can use the assignment operator to assign the value of `3` to `x` by executing:

```
x <- 3
```

The assignment operator (`<-`) assigns **values on the right** to **variables on the left**.

In RStudio, typing `Alt + -` (push `Alt` at the same time as the `-` key, on Mac type `option + -`) will write `<-` in a single keystroke.

2.8 Variables

A variable is a symbolic name for (or reference to) information. Variables in computer programming are analogous to "buckets", where information can be maintained and referenced. On the outside of the bucket is a name. When referring to the bucket, we use the name of the bucket, not the data stored in the bucket.

In the example above, we created a variable or a 'bucket' called `x`. Inside we put a value, `3`.

Let's create another variable called `y` and give it a value of 5.

```
y <- 5
```

When assigning a value to an variable, R does not print anything to the console. You can force to print the value by using parentheses or by typing the variable name.

```
y
```

You can also view information on the variable by looking in your `Environment` window in the upper right-hand corner of the RStudio interface.

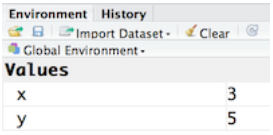


Figure 2.11: Viewing your environment

Now we can reference these buckets by name to perform mathematical operations on the values contained within. What do you get in the console for the following operation:

```
x + y
```

Try assigning the results of this operation to another variable called `number`.

```
number <- x + y
```

---

3 R Basics

> Extended Materials
>
> You can find the original, extended version of this chapter here.

3.1 Functions

Functions are at the core of using R. Functions are how you perform tasks and operations. Many functions come installed with R, many more are available for download in *packages* (explained in the packages section), and you can even write your own custom functions!

This basics section on functions explains:

- What a function is and how they work
- What function *arguments* are
- How to get help understanding a function

> A quick note on syntax
>
> In this workbook, functions are written in code-text with open parentheses, like this: `filter()`. As explained in the packages section, functions are downloaded within *packages*. In this handbook, package names are written in **bold**, like **dplyr**. Sometimes in example code you may see the function name linked explicitly to the name of its package with two colons (`::`) like this: `dplyr::filter()`. The purpose of this linkage is explained in the packages section.

Simple functions

A function is like a machine that receives inputs, does some action with those inputs, and produces an output. What the output is depends on the function.

Functions typically operate upon some object placed within the function's parentheses. For example, the function `sqrt()` calculates the square root of a number:

```
sqrt(49)
```

```
[1] 7
```

The object provided to a function also can be a column in a dataset (see the Objects section for detail on all the kinds of objects). Because R can store multiple datasets, you will need to specify both the dataset and the column. One way to do this is using the `$` notation to link the name of the dataset and the name of the column (`dataset$column`). In the example below, the function `summary()` is applied to the numeric column `age` in the dataset `linelist`, and the output is a summary of the column's numeric and missing values.

```
# Print summary statistics of column 'age' in the dataset 'linelist'
summary(linelist$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    6.00   13.00   16.07   23.00   84.00      86
```

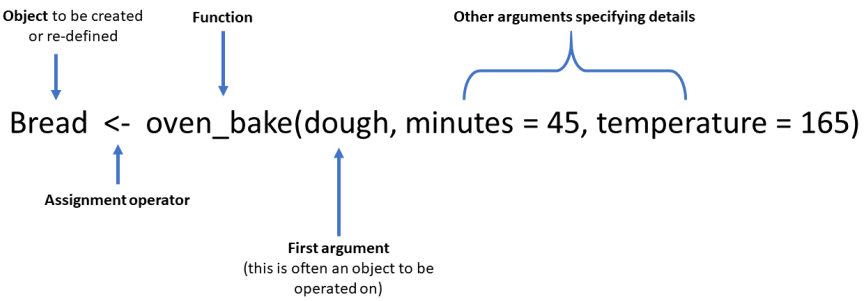
> **Note**
>
> Behind the scenes, a function represents complex additional code that has been wrapped up for the user into one easy command.

Functions with multiple arguments

Functions often ask for several inputs, called ***arguments***, located within the parentheses of the function, usually separated by commas.

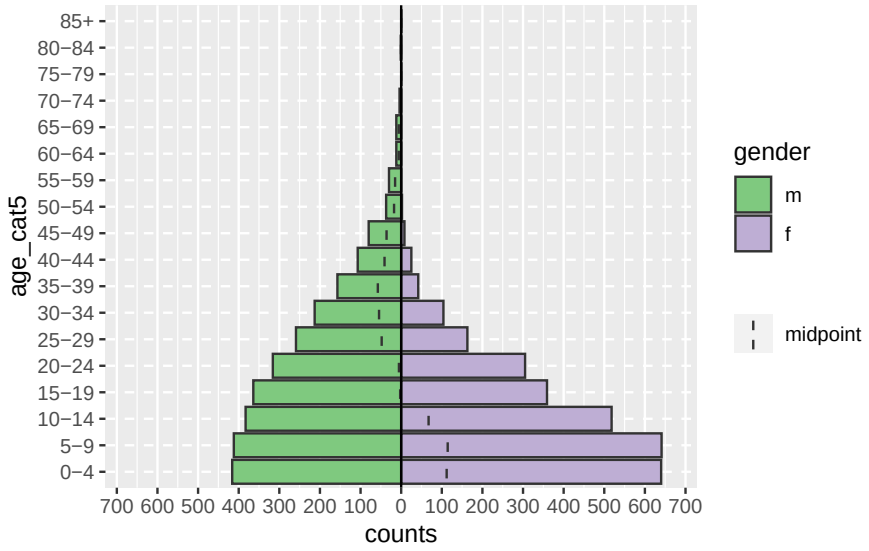
- Some arguments are required for the function to work correctly, others are optional
- Optional arguments have default settings
- Arguments can take character, numeric, logical (TRUE/FALSE), and other inputs

Here is a fun fictional function, called `oven_bake()`, as an example of a typical function. It takes an input object (e.g. a dataset, or in this example "dough") and performs operations on it as specified by additional arguments (`minutes =` and `temperature =`). The output can be printed to the console, or saved as an object using the assignment operator `<-`.



In a more realistic example, the `age_pyramid()` command below produces an age pyramid plot based on defined age groups and a binary split column, such as `gender`. The function is given three arguments within the parentheses, separated by commas. The values supplied to the arguments establish `linelist` as the dataframe to use, `age_cat5` as the column to count, and `gender` as the binary column to use for splitting the pyramid by color.

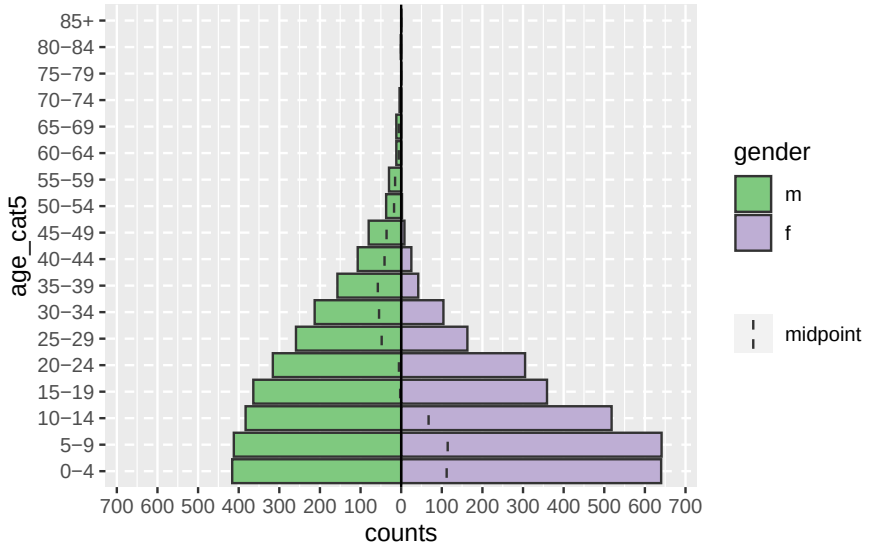
```
# Create an age pyramid
age_pyramid(data = linelist, age_group = "age_cat5", split_by = "gender")
```



The above command can be equivalently written as below, in a longer style with a new line

for each argument. This style can be easier to read, and easier to write "comments" with `#` to explain each part (commenting extensively is good practice!). To run this longer command you can highlight the entire command and click "Run", or just place your cursor in the first line and then press the Ctrl and Enter keys simultaneously.

```
# Create an age pyramid
age_pyramid(
  data = linelist,         # use case linelist
  age_group = "age_cat5", # provide age group column
  split_by = "gender"      # use gender column for two sides of pyramid
  )
```



The first half of an argument assignment (e.g. `data =`) does not need to be specified if the arguments are written in a specific order (specified in the function's documentation). The below code produces the exact same pyramid as above, because the function expects the argument order: data frame, `age_group` variable, `split_by` variable.

```
# This command will produce the exact same graphic as above
age_pyramid(linelist, "age_cat5", "gender")
```

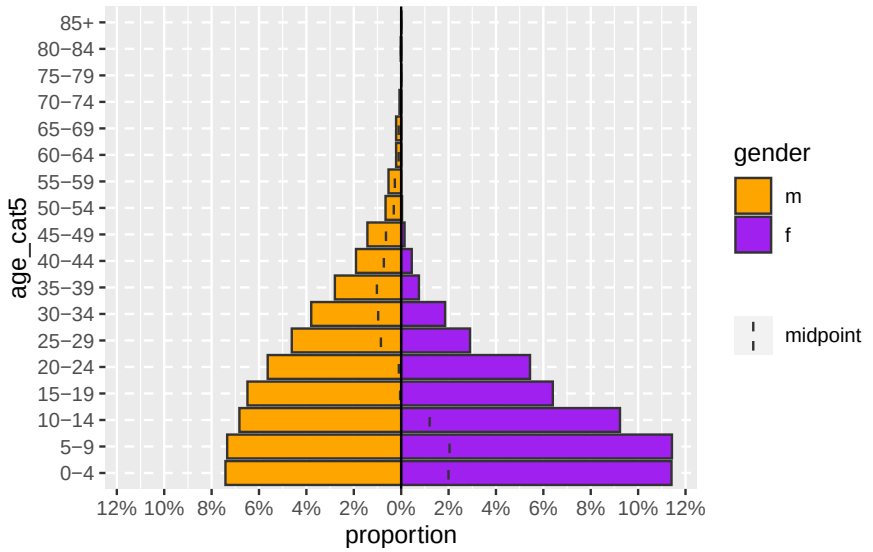
A more complex `age_pyramid()` command might include the *optional* arguments to:

- Show proportions instead of counts (set `proportional = TRUE` when the default is `FALSE`)

- Specify the two colors to use (`pal =` is short for "palette" and is supplied with a vector of two color names. See the objects page for how the function `c()` makes a vector)

> **Note**
>
> For arguments that you specify with both parts of the argument (e.g. `proportional = TRUE`), their order among all the arguments does not matter.

```
age_pyramid(
  linelist,                     # use case linelist
  "age_cat5",                   # age group column
  "gender",                     # split by gender
  proportional = TRUE,          # percents instead of counts
  pal = c("orange", "purple")   # colors
  )
```



3.2 Packages

Packages contain functions.

An R package is a shareable bundle of code and documentation that contains pre-defined functions. Users in the R community develop packages all the time catered to specific problems, it is likely that one can help with your work! You will install and use hundreds of packages in your use of R.

On installation, R contains **"base"** packages and functions that perform common elementary tasks. But many R users create specialized functions, which are verified by the R community and which you can download as a **package** for your own use. In this handbook, package names are written in **bold**. One of the more challenging aspects of R is that there are often many functions or packages to choose from to complete a given task.

Install and load

Functions are contained within **packages** which can be downloaded ("installed") to your computer from the internet. Once a package is downloaded, it is stored in your "library". You can then access the functions it contains during your current R session by "loading" the package.

Think of R as your personal library: When you download a package, your library gains a new book of functions, but each time you want to use a function in that book, you must borrow ("load") that book from your library.

In summary: to use the functions available in an R package, 2 steps must be implemented:

1) The package must be **installed** (once), *and*

2) The package must be **loaded** (each R session)

Your library

Your "library" is actually a folder on your computer, containing a folder for each package that has been installed. Find out where R is installed in your computer, and look for a folder called "win-library". For example: `R\win-library\4.0` (the 4.0 is the R version - you'll have a different library for each R version you've downloaded).

You can print the file path to your library by entering `.libPaths()` (empty parentheses).

Install from CRAN

Most often, R users download packages from CRAN. CRAN (Comprehensive R Archive Network) is an online public warehouse of R packages that have been published by R community members.

How to install and load

The **base** R function for installing a package is `install.packages()`. The name of the package to install must be provided in the parentheses *in quotes*. If you want to install multiple packages in one command, they must be listed within a character vector `c()`.

> ⚠ Common Mistake
>
> This command *installs* a package, but does *not* load it for use in the current session.

```
# install a single package with base R
install.packages("tidyverse")

# install multiple packages with base R
install.packages(c("tidyverse", "rio", "here"))
```

Installation can also be accomplished point-and-click by going to the RStudio "Packages" pane and clicking "Install" and searching for the desired package name.

The **base** R function to **load** a package for use (after it has been installed) is `library()`. It can load only one package at a time (another reason to use `p_load()`). You can provide the package name with or without quotes.

```
# load packages for use, with base R
library(tidyverse)
library(rio)
library(here)
```

To check whether a package in installed and/or loaded, you can view the Packages pane in RStudio. If the package is installed, it is shown there with version number. If its box is checked, it is loaded for the current session.

Code syntax

For clarity in this handbook, functions are sometimes preceded by the name of their package using the `::` symbol in the following way: `package_name::function_name()`

Once a package is loaded for a session, this explicit style is not necessary. One can just use `function_name()`. However writing the package name is useful when a function name is common and may exist in multiple packages (e.g. `plot()`). Writing the package name will also load the package if it is not already loaded.

```r
# This command uses the package "rio" and its function "import()" to import a dataset
linelist <- rio::import("linelist.xlsx", which = "Sheet1")
```

Function help

To read more about a function, you can search for it in the Help tab of the lower-right RStudio. You can also run a command like `?thefunctionname` (put the name of the function after a question mark) and the Help page will appear in the Help pane. Finally, try searching online for resources.

Update packages

You can update packages by re-installing them. You can also click the green "Update" button in your RStudio Packages pane to see which packages have new versions Wto install. Be aware that your old code may need to be updated if there is a major revision to how a function works!

Delete packages

Use `remove.packages()`. Alternatively, go find the folder which contains your library and manually delete the folder.

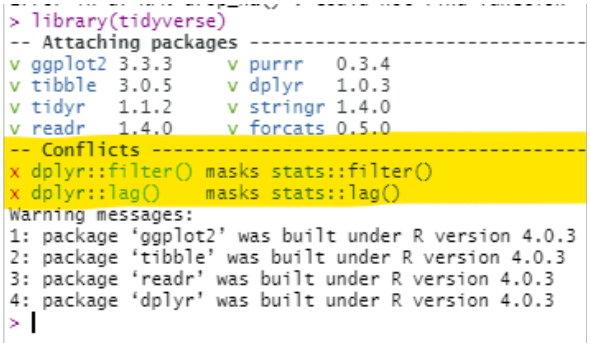
Dependencies

Packages often depend on other packages to work. These are called dependencies. If a dependency fails to install, then the package depending on it may also fail to install.

Masked functions

It is not uncommon that two or more packages contain the same function name. For example, the package **dplyr** has a `filter()` function, but so does the package **stats**. The default `filter()` function depends on the order these packages are first loaded in the R session - the later one will be the default for the command `filter()`.

You can check the order in your Environment pane of R Studio - click the drop-down for "Global Environment" and see the order of the packages. Functions from packages *lower* on that drop-down list will mask functions of the same name in packages that appear higher in the drop-down list. When first loading a package, R will warn you in the console if masking is occurring, but this can be easy to miss.



Here are ways you can fix masking:

1) Specify the package name in the command. For example, use `dplyr::filter()`

2) Re-arrange the order in which the packages are loaded and **start a new R session**

Install older version

See this guide to install an older version of a particular package.

3.3 Scripts

Scripts are a fundamental part of programming. They are documents that hold your commands (e.g. functions to create and modify datasets, print visualizations, etc). You can save a script and run it again later. There are many advantages to storing and running your commands from a script (vs. typing commands one-by-one into the R console "command line"):

- Portability - you can share your work with others by sending them your scripts
- Reproducibility - so that you and others know exactly what you did
- Version control - so you can track changes made by yourself or colleagues
- Commenting/annotation - to explain to your colleagues what you have done

Commenting

In a script you can also annotate ("comment") around your R code. Commenting is helpful to explain to yourself and other readers what you are doing. You can add a comment by typing

the hash symbol (#) and writing your comment after it. The commented text will appear in a different color than the R code.

Any code written after the # will not be run. Therefore, placing a # before code is also a useful way to temporarily block a line of code ("comment out") if you do not want to delete it). You can comment out/in multiple lines at once by highlighting them and pressing Ctrl+Shift+c (Cmd+Shift+c in Mac).

```
# A comment can be on a line by itself
# import data
linelist <- import("linelist_raw.xlsx") %>%   # a comment can also come after code
# filter(age > 50)                           # It can also be used to deactivate / remove a
  count()
```

- Comment on *what* you are doing *and on* ***why*** *you are doing it.*
- Break your code into logical sections
- Accompany your code with a text step-by-step description of what you are doing (e.g. numbered steps)

Style

It is important to be conscious of your coding style - especially if working on a team. Some nice R styles guide are the **tidyverse** style guide, Hadley Wickham's style guide, and Google's. There are also packages such as **styler** and **lintr** which help you conform to a style.

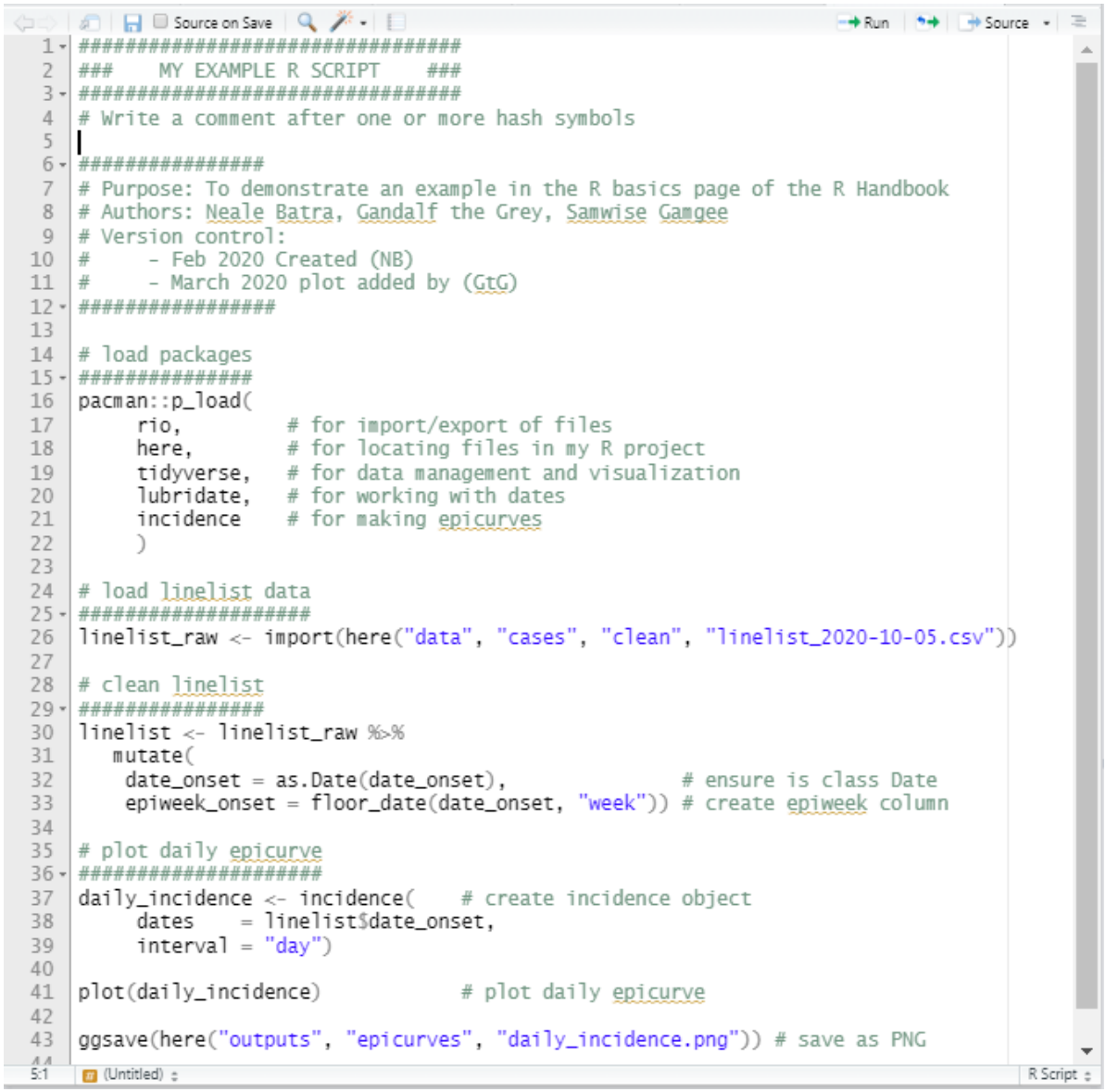
A few very basic points to make your code readable to others:
* When naming objects, use only lowercase letters, numbers, and underscores `_`, e.g. `my_data`
* Use frequent spaces, including around operators, e.g. `n = 1` and `age_new <- age_old + 3`

Example Script

Below is an example of a short R script. Remember, the better you succinctly explain your code in comments, the more your colleagues will like you!

R markdown

An R markdown script is a type of R script in which the script itself *becomes* an output document (PDF, Word, HTML, Powerpoint, etc.). These are incredibly useful and versatile



```r
################################
###    MY EXAMPLE R SCRIPT    ###
################################
# Write a comment after one or more hash symbols

################
# Purpose: To demonstrate an example in the R basics page of the R Handbook
# Authors: Neale Batra, Gandalf the Grey, Samwise Gamgee
# Version control:
#     - Feb 2020 Created (NB)
#     - March 2020 plot added by (GtG)
#################

# load packages
###############
pacman::p_load(
     rio,          # for import/export of files
     here,         # for locating files in my R project
     tidyverse,    # for data management and visualization
     lubridate,    # for working with dates
     incidence     # for making epicurves
     )

# load linelist data
####################
linelist_raw <- import(here("data", "cases", "clean", "linelist_2020-10-05.csv"))

# clean linelist
################
linelist <- linelist_raw %>%
   mutate(
    date_onset = as.Date(date_onset),               # ensure is class Date
    epiweek_onset = floor_date(date_onset, "week")) # create epiweek column

# plot daily epicurve
####################
daily_incidence <- incidence(    # create incidence object
     dates    = linelist$date_onset,
     interval = "day")

plot(daily_incidence)            # plot daily epicurve

ggsave(here("outputs", "epicurves", "daily_incidence.png")) # save as PNG
```

tools often used to create dynamic and automated reports. Even this website and handbook is produced with R markdown scripts!

It is worth noting that beginner R users can also use R Markdown - do not be intimidated! To learn more, see the handbook page on [Reports with R Markdown] documents.

3.4 Objects

Everything in R is an object, and R is an "object-oriented" language. These sections will explain:

- How to create objects (`<-`)
- Types of objects (e.g. data frames, vectors..)
- How to access subparts of objects (e.g. variables in a dataset)
- Classes of objects (e.g. numeric, logical, integer, double, character, factor)

Everything is an object

Everything you store in R - datasets, variables, a list of village names, a total population number, even outputs such as graphs - are **objects** which are **assigned a name** and **can be referenced** in later commands.

An object exists when you have assigned it a value (see the assignment section below). When it is assigned a value, the object appears in the Environment (see the upper right pane of RStudio). It can then be operated upon, manipulated, changed, and re-defined.

Defining objects (<-)

Create objects *by assigning them a value* with the <- operator.
You can think of the assignment operator `<-` as the words "is defined as". Assignment commands generally follow a standard order:

object_name <- **value** (or process/calculation that produce a value)

For example, you may want to record the current epidemiological reporting week as an object for reference in later code. In this example, the object `current_week` is created when it is assigned the value `"2018-W10"` (the quote marks make this a character value). The object `current_week` will then appear in the RStudio Environment pane (upper-right) and can be referenced in later commands.

See the R commands and their output in the boxes below.

```r
current_week <- "2018-W10"   # this command creates the object current_week by assigning i
current_week                 # this command prints the current value of current_week objec
```

```
[1] "2018-W10"
```

> **Note**
>
> Note the `[1]` in the R console output is simply indicating that you are viewing the first item of the output

> **Common Mistake**
>
> **An object's value can be over-written** at any time by running an assignment command to re-define its value. Thus, the **order of the commands run is very important**.
> For instance, the following command will re-define the value of `current_week`:
>
> ```r
> current_week <- "2018-W51"   # assigns a NEW value to the object current_week
> current_week                 # prints the current value of current_week in the console
> ```
>
> ```
> [1] "2018-W51"
> ```

Equals signs =

You will also see equals signs in R code:

- A double equals sign `==` between two objects or values asks a logical *question*: "is this equal to that?".
- You will also see equals signs within functions used to specify values of function arguments (read about these in sections below), for example `max(age, na.rm = TRUE)`.
- You *can* use a single equals sign `=` in place of `<-` to create and define objects, but this is discouraged. You can read about why this is discouraged here.

Datasets

Datasets are also objects (typically "dataframes") and must be assigned names when they are imported. In the code below, the object `linelist` is created and assigned the value of a CSV file imported with the **rio** package and its `import()` function.

```r
# linelist is created and assigned the value of the imported CSV file
linelist <- import("my_linelist.csv")
```

You can read more about importing and exporting datasets with the section on [Import and export].

> **Naming Objects**
>
> - Object names must not contain spaces, but you should use underscore (_) or a period (.) instead of a space.
> - Object names are case-sensitive (meaning that Dataset_A is different from dataset_A).
> - Object names must begin with a letter (cannot begin with a number like 1, 2 or 3).

Outputs

Outputs like tables and plots provide an example of how outputs can be saved as objects, or just be printed without being saved. A cross-tabulation of gender and outcome using the **base** R function `table()` can be printed directly to the R console (*without* being saved).

```r
# printed to R console only
table(linelist$gender, linelist$outcome)
```

```
  Death Recover
f  1227     953
m  1228     950
```

But the same table can be saved as a named object. Then, optionally, it can be printed.

```r
# save
gen_out_table <- table(linelist$gender, linelist$outcome)

# print
gen_out_table
```

```
  Death Recover
f  1227     953
m  1228     950
```

Columns

Columns in a dataset are also objects and can be defined, over-written, and created as described below in the section on Columns.

You can use the assignment operator from **base** R to create a new column. Below, the new column `bmi` (Body Mass Index) is created, and for each row the new value is result of a mathematical operation on the row's value in the `wt_kg` and `ht_cm` columns.

```
# create new "bmi" column using base R syntax
linelist$bmi <- linelist$wt_kg / (linelist$ht_cm/100)^2
```

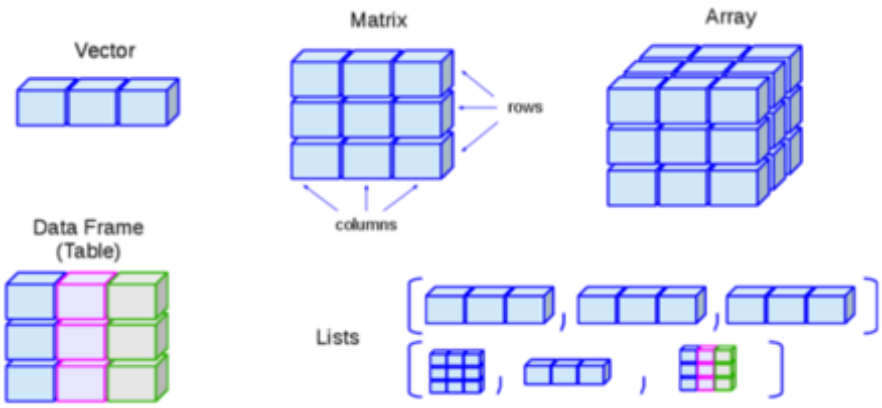
However, in this handbook, we emphasize a different approach to defining columns, which uses the function `mutate()` from the **dplyr** package and *piping* with the pipe operator (`%>%`). The syntax is easier to read and there are other advantages explained in the page on [Cleaning data and core functions]. You can read more about *piping* in the Piping section below.

```
# create new "bmi" column using dplyr syntax
linelist <- linelist %>%
  mutate(bmi = wt_kg / (ht_cm/100)^2)
```

Object structure

Objects can be a single piece of data (e.g. `my_number <- 24`), or they can consist of structured data.

The graphic below is borrowed from this online R tutorial. It shows some common data structures and their names. Not included in this image is spatial data, which is discussed in the [GIS basics] page.



In epidemiology (and particularly field epidemiology), you will *most commonly* encounter data frames and vectors:

Common structure	Explanation	Example
Vectors	A container for a sequence of singular objects, all of the same class (e.g. numeric, character).	**"Variables" (columns) in data frames are vectors** (e.g. the column `age_years`).
Data Frames	Vectors (e.g. columns) that are bound together that all have the same number of rows.	`linelist` is a data frame.

Note that to create a vector that "stands alone" (is not part of a data frame) the function `c()` is used to combine the different elements. For example, if creating a vector of colors plot's color scale: `vector_of_colors <- c("blue", "red2", "orange", "grey")`

Object classes

All the objects stored in R have a *class* which tells R how to handle the object. There are many possible classes, but common ones include:

Class	Explanation	Examples
Character	These are text/words/sentences **"within quotation marks"**. Math cannot be done on these objects.	"Character objects are in quotation marks"
Integer	Numbers that are **whole only** (no decimals)	-5, 14, or 2000
Numeric	These are numbers and **can include decimals**. If within quotation marks they will be considered character class.	23.1 or 14
Factor	These are vectors that have a **specified order** or hierarchy of values	An variable of economic status with ordered values
Date	**Once R is told that certain data are Dates**, these data can be manipulated and displayed in special ways. See the page on [Working with dates] for more information.	2018-04-12 or 15/3/1954 or Wed 4 Jan 1980
Logical	Values must be one of the two special values TRUE or FALSE (note these are **not** "TRUE" and "FALSE" in quotation marks)	TRUE or FALSE

Class	Explanation	Examples
data.frame	A data frame is how R stores a **typical dataset**. It consists of vectors (columns) of data bound together, that all have the same number of observations (rows).	The example AJS dataset named `linelist_raw` contains 68 variables with 300 observations (rows) each.
tibble	tibbles are a variation on data frame, the main operational difference being that they print more nicely to the console (display first 10 rows and only columns that fit on the screen)	Any data frame, list, or matrix can be converted to a tibble with `as_tibble()`
list	A list is like vector, but holds other objects that can be other different classes	A list could hold a single number, and a dataframe, and a vector, and even another list within it!

You can test the class of an object by providing its name to the function `class()`. You can reference a specific column within a dataset using the `$` notation to separate the name of the dataset and the name of the column.

```
class(linelist)          # class should be a data frame or tibble
```

```
[1] "data.frame"
```

```
class(linelist$age)      # class should be numeric
```

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

```
class(linelist$gender)  # class should be character
```

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

Sometimes, a column will be converted to a different class automatically by R. Watch out for this! For example, if you have a vector or column of numbers, but a character value is inserted… the entire column will change to class character.

```r
num_vector <- c(1,2,3,4,5) # define vector as all numbers
class(num_vector)          # vector is numeric class
```

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

```r
num_vector[3] <- "three"   # convert the third element to a character
class(num_vector)          # vector is now character class
```

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

One common example of this is when manipulating a data frame in order to print a table - if you make a total row and try to paste/glue together percents in the same cell as numbers (e.g. `23 (40%)`), the entire numeric column above will convert to character and can no longer be used for mathematical calculations.**Sometimes, you will need to convert objects or columns to another class.**

Function	Action
`as.character()`	Converts to character class
`as.numeric()`	Converts to numeric class
`as.integer()`	Converts to integer class
`as.Date()`	Converts to Date class
`factor()`	Converts to factor

Likewise, there are **base** R functions to check whether an object IS of a specific class, such as `is.numeric()`, `is.character()`, `is.double()`, `is.factor()`, `is.integer()`

Here is more online material on classes and data structures in R.

Columns/Variables ($)

A column in a data frame is technically a "vector" (see table above) - a series of values that must all be the same class (either character, numeric, logical, etc).

A vector can exist independent of a data frame, for example a vector of column names that you want to include as explanatory variables in a model. To create a "stand alone" vector, use the `c()` function as below:

```r
# define the stand-alone vector of character values
explanatory_vars <- c("gender", "fever", "chills", "cough", "aches", "vomit")
```

```r
# print the values in this named vector
explanatory_vars
```

```
[1] "gender" "fever"  "chills" "cough"  "aches"  "vomit"
```

Columns in a data frame are also vectors and can be called, referenced, extracted, or created using the $ symbol. The $ symbol connects the name of the column to the name of its data frame. In this handbook, we try to use the word "column" instead of "variable".

```r
# Retrieve the length of the vector age_years
length(linelist$age) # (age is a column in the linelist data frame)
```

Access/index with brackets ([ ])

You may need to view parts of objects, also called "indexing", which is often done using the square brackets [ ]. Using $ on a dataframe to access a column is also a type of indexing.

```r
my_vector <- c("a", "b", "c", "d", "e", "f")  # define the vector
my_vector[5]                                  # print the 5th element
```

```
[1] "e"
```

Square brackets also work to return specific parts of an returned output, such as the output of a `summary()` function:

```r
# All of the summary
summary(linelist$age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    6.00   13.00   16.07   23.00   84.00      86
```

```r
# Just the second element of the summary, with name (using only single brackets)
summary(linelist$age)[2]
```

```
1st Qu.
      6
```

```r
# Just the second element, without name (using double brackets)
summary(linelist$age)[[2]]
```

```
[1] 6
```

```r
# Extract an element by name, without showing the name
summary(linelist$age)[["Median"]]
```

```
[1] 13
```

Brackets also work on data frames to view specific rows and columns. You can do this using the syntax `dataframe[rows, columns]`:

```r
# View a specific row (2) from dataset, with all columns (don't forget the comma!)
linelist[2,]

# View all rows, but just one column
linelist[, "date_onset"]

# View values from row 2 and columns 5 through 10
linelist[2, 5:10]

# View values from row 2 and columns 5 through 10 and 18
linelist[2, c(5:10, 18)]

# View rows 2 through 20, and specific columns
linelist[2:20, c("date_onset", "outcome", "age")]

# View rows and columns based on criteria
# *** Note the dataframe must still be named in the criteria!
linelist[linelist$age > 25 , c("date_onset", "outcome", "age")]

# Use View() to see the outputs in the RStudio Viewer pane (easier to read)
# *** Note the capital "V" in View() function
View(linelist[2:20, "date_onset"])

# Save as a new object
new_table <- linelist[2:20, c("date_onset")]
```

Using Tidyverse Instead

In a future session, we will learn how to perform many of these operations using **dplyr** syntax (functions `filter()` for rows, and `select()` for columns) as opposed to the base R syntax shown here.
To filter based on "row number", you can use the **dplyr** function `row_number()` with open parentheses as part of a logical filtering statement. Often you will use the `%in%` operator and a range of numbers as part of that logical statement, as shown below. To see the *first* N rows, you can also use the special **dplyr** function `head()`.

```
# View first 100 rows
linelist %>% head(100)

# Show row 5 only
linelist %>% filter(row_number() == 5)

# View rows 2 through 20, and three specific columns (note no quotes necessary on column
linelist %>% filter(row_number() %in% 2:20) %>% select(date_onset, outcome, age)
```

When indexing an object of class **list**, single brackets always return with class list, even if only a single object is returned. Double brackets, however, can be used to access a single element and return a different class than list.
Brackets can also be written after one another, as demonstrated below.

This visual explanation of lists indexing, with pepper shakers is humorous and helpful.

```
# define demo list
my_list <- list(
  # First element in the list is a character vector
  hospitals = c("Central", "Empire", "Santa Anna"),

  # second element in the list is a data frame of addresses
  addresses   = data.frame(
    street = c("145 Medical Way", "1048 Brown Ave", "999 El Camino"),
    city   = c("Andover", "Hamilton", "El Paso")
    )
  )
```

Here is how the list looks when printed to the console. See how there are two named elements:

- `hospitals`, a character vector
- `addresses`, a data frame of addresses

```
  my_list
```

```
$hospitals
[1] "Central"    "Empire"     "Santa Anna"

$addresses
           street     city
1 145 Medical Way  Andover
2  1048 Brown Ave Hamilton
3   999 El Camino  El Paso
```

Now we extract, using various methods:

```
  my_list[1] # this returns the element in class "list" - the element name is still displaye
```

```
$hospitals
[1] "Central"    "Empire"     "Santa Anna"
```

```
  my_list[[1]] # this returns only the (unnamed) character vector
```

```
[1] "Central"    "Empire"     "Santa Anna"
```

```
  my_list[["hospitals"]] # you can also index by name of the list element
```

```
[1] "Central"    "Empire"     "Santa Anna"
```

```
  my_list[[1]][3] # this returns the third element of the "hospitals" character vector
```

```
[1] "Santa Anna"
```

```
  my_list[[2]][1] # This returns the first column ("street") of the address data frame
```

```
           street
1 145 Medical Way
2  1048 Brown Ave
3   999 El Camino
```

Remove objects

You can remove individual objects from your R environment by putting the name in the `rm()` function (no quote marks):

```
rm(object_name)
```

You can remove all objects (clear your workspace) by running:

```
rm(list = ls(all = TRUE))
```

3.4.1 Categorical data: factors

Since factors are special vectors, the same rules for selecting values using indices apply.

```
expression <- c("high","low","low","medium","high","medium","medium","low","low","low")
```

The elements of this expression factor created previously has following categories or levels: low, medium, and high.

Let's extract the values of the factor with high expression, and let's using nesting here:

```
expression[expression == "high"]    ## This will only return those elements in the factor
```

```
[1] "high" "high"
```

> **Nesting note**:
>
> The piece of code above was more efficient with nesting; we used a single step instead of two steps as shown below:
>
> Step1 (no nesting): `idx <- expression == "high"`
>
> Step2 (no nesting): `expression[idx]`

3.4.1.1 Releveling factors

We have briefly talked about factors, but this data type only becomes more intuitive once you've had a chance to work with it. Let's take a slight detour and learn about how to **relevel categories within a factor**.

To view the integer assignments under the hood you can use `str()`:

```
expression
```

```
[1] "high"   "low"    "low"    "medium" "high"   "medium" "medium" "low"
[9] "low"    "low"
```

The categories are referred to as "factor levels". As we learned earlier, the levels in the `expression` factor were assigned integers alphabetically, with high=1, low=2, medium=3. However, it makes more sense for us if low=1, medium=2 and high=3, i.e. it makes sense for us to "relevel" the categories in this factor.

To relevel the categories, you can add the `levels` argument to the `factor()` function, and give it a vector with the categories listed in the required order:

```
expression <- factor(expression, levels=c("low", "medium", "high"))     # you can re-facto
```

Now we have a releveled factor with low as the lowest or first category, medium as the second and high as the third. This is reflected in the way they are listed in the output of `str()`, as well as in the numbering of which category is where in the factor.

> Note: Releveling becomes necessary when you need a specific category in a factor to be the "base" category, i.e. category that is equal to 1. One example would be if you need the "control" to be the "base" in a given RNA-seq experiment.

3.5 Piping (%>%)

Two general approaches to working with objects are:

1) **Pipes/tidyverse** - pipes send an object from function to function - emphasis is on the *action*, not the object

2) **Define intermediate objects** - an object is re-defined again and again - emphasis is on the object

Pipes

Simply explained, the pipe operator (%>%) passes an intermediate output from one function to the next.
You can think of it as saying "then". Many functions can be linked together with %>%.

- **Piping emphasizes a sequence of actions, not the object the actions are being performed on**

- Pipes are best when a sequence of actions must be performed on one object

- Pipes come from the package **magrittr**, which is automatically included in packages **dplyr** and **tidyverse**
- Pipes can make code more clean and easier to read, more intuitive

Read more on this approach in the tidyverse style guide

Here is a fake example for comparison, using fictional functions to "bake a cake". First, the pipe method:

```
# A fake example of how to bake a cake using piping syntax

cake <- flour %>%       # to define cake, start with flour, and then...
  add(eggs) %>%   # add eggs
  add(oil) %>%    # add oil
  add(water) %>%  # add water
  mix_together(         # mix together
    utensil = spoon,
    minutes = 2) %>%
  bake(degrees = 350,   # bake
       system = "fahrenheit",
       minutes = 35) %>%
  let_cool()            # let it cool down
```

Here is another link describing the utility of pipes.

Piping is not a **base** function. To use piping, the **magrittr** package must be installed and loaded (this is typically done by loading **tidyverse** or **dplyr** package which include it). You can read more about piping in the magrittr documentation.

Note that just like other R commands, pipes can be used to just display the result, or to save/re-save an object, depending on whether the assignment operator <- is involved. See both below:

```
# Create or overwrite object, defining as aggregate counts by age category (not printed)
linelist_summary <- linelist %>%
  count(age_cat)
```

```r
# Print the table of counts in the console, but don't save it
linelist %>%
  count(age_cat)
```

```
  age_cat    n
1     0-4 1095
2     5-9 1095
3   10-14  941
4   15-19  743
5   20-29 1073
6   30-49  754
7   50-69   95
8     70+    6
9    <NA>   86
```

%<>%
This is an "assignment pipe" from the **magrittr** package, which *pipes an object forward and also re-defines the object*. It must be the first pipe operator in the chain. It is shorthand. The below two commands are equivalent:

```r
linelist <- linelist %>%
  filter(age > 50)

linelist %<>% filter(age > 50)
```

Define intermediate objects

This approach to changing objects/dataframes may be better if:

- You need to manipulate multiple objects
- There are intermediate steps that are meaningful and deserve separate object names

Risks:

- Creating new objects for each step means creating lots of objects. If you use the wrong one you might not realize it!
- Naming all the objects can be confusing
- Errors may not be easily detectable

Either name each intermediate object, or overwrite the original, or combine all the functions together. All come with their own risks.

Below is the same fake "cake" example as above, but using this style:

```
# a fake example of how to bake a cake using this method (defining intermediate objects)
batter_1 <- left_join(flour, eggs)
batter_2 <- left_join(batter_1, oil)
batter_3 <- left_join(batter_2, water)

batter_4 <- mix_together(object = batter_3, utensil = spoon, minutes = 2)

cake <- bake(batter_4, degrees = 350, system = "fahrenheit", minutes = 35)

cake <- let_cool(cake)
```

Combine all functions together - this is difficult to read:

```
# an example of combining/nesting mutliple functions together - difficult to read
cake <- let_cool(bake(mix_together(batter_3, utensil = spoon, minutes = 2), degrees = 350,
```

3.6 Errors vs. warnings and debugging tips

Error versus Warning

When a command is run, the R Console may show you warning or error messages in red text.

- A **warning** means that R has completed your command, but had to take additional steps or produced unusual output that you should be aware of.
- An **error** means that R was not able to complete your command.

Look for clues:

- The error/warning message will often include a line number for the problem.
- If an object "is unknown" or "not found", perhaps you spelled it incorrectly, forgot to call a package with library(), or forgot to re-run your script after making changes.

If all else fails, copy the error message into Google along with some key terms - chances are that someone else has worked through this already!

General syntax tips

A few things to remember when writing commands in R, to avoid errors and warnings:

- Always close parentheses - tip: count the number of opening "(" and closing parentheses ")" for each code chunk
- Avoid spaces in column and object names. Use underscore (_) or periods (.) instead
- Keep track of and remember to separate a function's arguments with commas

Code assists

Any script (RMarkdown or otherwise) will give clues when you have made a mistake. For example, if you forgot to write a comma where it is needed, or to close a parentheses, RStudio will raise a flag on that line, on the right side of the script, to warn you.

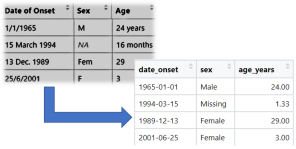
Part II

Session 1: Starting with Data

4 Getting Started with Data

Extended Materials

You can find the original, extended version of this chapter here.



Date of Onset	Sex	Age
1/1/1965	M	24 years
15 March 1994	*NA*	16 months
13 Dec. 1989	Fem	29
25/6/2001	F	3

date_onset	sex	age_years
1965-01-01	Male	24.00
1994-03-15	Missing	1.33
1989-12-13	Female	29.00
2001-06-25	Female	3.00

This week we are going to learn how to use R to manipulate data. This will include learning about core functions for manipulating and summarizing data, as well as using conditional statements to create subsets.

4.1 Key operators, functions, and constants

An **operators** is a symbol or set of symbols representing some mathematical or logical operation. They are essentially equivalent to functions. R has a number of built-in operators, and libraries may add additional operators (such as the `%>%` operator used in Tidyverse packages). Some examples of operators are:

- Definitional operators
- Relational operators (less than, equal too..)
- Logical operators (and, or…)
- Handling missing values
- Mathematical operators and functions (+/-, >, sum(), median(), …)
- The `%in%` operator

R also has some built-in **constants**, which have the same meaning in programming as in mathematics and statistics. Examples of constants in R include:

- `pi` which in base R equals `3.141593`
- `Inf` and `-Inf` for positive and negative infinity
- `NaN` for not a number (such as the result of `0/0`)
- `NA` for missing data

Assignment operators

`<-`

The basic assignment operator in R is `<-`. Such that `object_name <- value`.
This assignment operator can also be written as `=`. We advise use of `<-` for general R use.
We also advise surrounding such operators with spaces, for readability.

Relational and logical operators

Relational operators compare values and are often used when defining new variables and subsets of datasets. Here are the common relational operators in R:

Meaning	Operator	Example	Example Result
Equal to	`==`	`"A" == "a"`	`FALSE` (because R is case sensitive) *Note that == (double equals) is different from = (single equals), which acts like the assignment operator* `<-`
Not equal to	`!=`	`2 != 0`	`TRUE`
Greater than	`>`	`4 > 2`	`TRUE`
Less than	`<`	`4 < 2`	`FALSE`
Greater than or equal to	`>=`	`6 >= 4`	`TRUE`
Less than or equal to	`<=`	`6 <= 4`	`FALSE`
Value is missing	`is.na()`	`is.na(7)`	`FALSE` (see page on [Missing data])
Value is not missing	`!is.na()`	`!is.na(7)`	`TRUE`

Logical operators, such as AND and OR, are often used to connect relational operators and create more complicated criteria. Complex statements might require parentheses ( ) for grouping and order of application.

Meaning	Operator
AND	`&`
OR	`|` (vertical bar)
Parentheses	`( )` Used to group criteria together and clarify order of operations

For example, below, we have a linelist with two variables we want to use to create our case definition, `hep_e_rdt`, a test result and `other_cases_in_hh`, which will tell us if there are other cases in the household. The command below uses the function `case_when()` to create the new variable `case_def` such that:

```
linelist_cleaned <- linelist %>%
  mutate(case_def = case_when(
    is.na(rdt_result) & is.na(other_case_in_home)            ~ NA_character_,
    rdt_result == "Positive"                                 ~ "Confirmed",
    rdt_result != "Positive" & other_cases_in_home == "Yes"  ~ "Probable",
    TRUE                                                     ~ "Suspected"
  ))
```

Criteria in example above	Resulting value in new variable "case_def"
If the value for variables `rdt_result` and `other_cases_in_home` are missing	`NA` (missing)
If the value in `rdt_result` is "Positive"	"Confirmed"
If the value in `rdt_result` is NOT "Positive" AND the value in `other_cases_in_home` is "Yes"	"Probable"
If one of the above criteria are not met	"Suspected"

Note that R is case-sensitive, so "Positive" is different than "positive"…

Missing values

In R, missing values are represented by the special value `NA` (a "reserved" value) (capital letters N and A - not in quotation marks). **To test whether a value is `NA`, use the special function `is.na()`**, which returns `TRUE` or `FALSE`.

```r
rdt_result <- c("Positive", "Suspected", "Positive", NA)   # two positive cases, one suspe
is.na(rdt_result)  # Tests whether the value of rdt_result is NA
```

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

We will be learning more about how to deal with missing data in future weeks.

Mathematics and statistics

All the operators and functions in this page are automatically available using **base** R.

Mathematical operators

These are often used to perform addition, division, to create new columns, etc. Below are common mathematical operators in R. Whether you put spaces around the operators is not important.

Purpose	Example in R
addition	2 + 3
subtraction	2 - 3
multiplication	2 * 3
division	30 / 5
exponent	2^3
order of operations	( )

Mathematical functions

Purpose	Function
rounding	round(x, digits = n)
rounding	janitor::round_half_up(x, digits = n)
ceiling (round up)	ceiling(x)
floor (round down)	floor(x)
absolute value	abs(x)
square root	sqrt(x)
exponent	exponent(x)
natural logarithm	log(x)
log base 10	log10(x)

Purpose	Function
log base 2	log2(x)

Note: for `round()` the `digits =` specifies the number of decimal placed. Use `signif()` to round to a number of significant figures.

Statistical functions

> ⚠ Warning
>
> The functions below will by default include missing values in calculations. Missing values will result in an output of `NA`, unless the argument `na.rm = TRUE` is specified. This can be written shorthand as `na.rm = T`.

Objective	Function
mean (average)	mean(x, na.rm=T)
median	median(x, na.rm=T)
standard deviation	sd(x, na.rm=T)
quantiles*	quantile(x, probs)
sum	sum(x, na.rm=T)
minimum value	min(x, na.rm=T)
maximum value	max(x, na.rm=T)
range of numeric values	range(x, na.rm=T)
summary**	summary(x)

Notes:

- `*quantile()`: `x` is the numeric vector to examine, and `probs =` is a numeric vector with probabilities within 0 and 1.0, e.g `c(0.5, 0.8, 0.85)`
- `**summary()`: gives a summary on a numeric vector including mean, median, and common percentiles

> ⚠ Warning
>
> If providing a vector of numbers to one of the above functions, be sure to wrap the numbers within `c()` .}

```
# If supplying raw numbers to a function, wrap them in c()
mean(1, 6, 12, 10, 5, 0)    # !!! INCORRECT !!!
```

```
[1] 1
```

```
mean(c(1, 6, 12, 10, 5, 0)) # CORRECT
```

```
[1] 5.666667
```

Other useful functions

Objective	Function	Example
create a sequence	seq(from, to, by)	`seq(1, 10, 2)`
repeat x, n times	rep(x, ntimes)	`rep(1:3, 2)` or `rep(c("a", "b", "c"), 3)`
subdivide a numeric vector	cut(x, n)	`cut(linelist$age, 5)`
take a random sample	sample(x, size)	`sample(linelist$id, size = 5, replace = TRUE)`

%in%

A very useful operator for matching values, and for quickly assessing if a value is within a vector or dataframe.

```
my_vector <- c("a", "b", "c", "d")
```

```
"a" %in% my_vector
```

```
[1] TRUE
```

```
"h" %in% my_vector
```

```
[1] FALSE
```

To ask if a value is **not** **%in%** a vector, put an exclamation mark (!) **in front** of the logic statement:

```r
# to negate, put an exclamation in front
!"a" %in% my_vector
```

```
[1] FALSE
```

```r
!"h" %in% my_vector
```

```
[1] TRUE
```

`%in%` is very useful when using the **dplyr** function `case_when()`. You can define a vector previously, and then reference it later. For example:

```r
affirmative <- c("1", "Yes", "YES", "yes", "y", "Y", "oui", "Oui", "Si")

linelist <- linelist %>%
  mutate(child_hospitaled = case_when(
    hospitalized %in% affirmative & age < 18 ~ "Hospitalized Child",
    TRUE                                     ~ "Not"))
```

Core functions

We will be emphasizing use of the functions from the **tidyverse** family of R packages. The functions we will be learning about are listed below.

Many of these functions belong to the **dplyr** R package, which provides "verb" functions to solve data manipulation challenges (the name is a reference to a "data frame-plier. **dplyr** is part of the **tidyverse** family of R packages (which also includes **ggplot2**, **tidyr**, **stringr**, **tibble**, **purrr**, **magrittr**, and **forcats** among others).

Function	Utility	Package
`%>%`	"pipe" (pass) data from one function to the next	**magrittr**
`mutate()`	create, transform, and re-define columns	**dplyr**
`select()`	keep, remove, select, or re-name columns	**dplyr**
`rename()`	rename columns	**dplyr**

Function	Utility	Package
`clean_names()`	standardize the syntax of column names	**janitor**
`as.character()`, `as.numeric()`, `as.Date()`, etc.	convert the class of a column	**base** R
`across()`	transform multiple columns at one time	**dplyr**
tidyselect functions	use logic to select columns	**tidyselect**
`filter()`	keep certain rows	**dplyr**
`distinct()`	de-duplicate rows	**dplyr**
`rowwise()`	operations by/within each row	**dplyr**
`add_row()`	add rows manually	**tibble**
`arrange()`	sort rows	**dplyr**
`recode()`	re-code values in a column	**dplyr**
`case_when()`	re-code values in a column using more complex logical criteria	**dplyr**
`replace_na()`, `na_if()`, `coalesce()`	special functions for re-coding	**tidyr**
`age_categories()` and `cut()`	create categorical groups from a numeric column	**epikit** and **base** R
`match_df()`	re-code/clean values using a data dictionary	**matchmaker**
`which()`	apply logical criteria; return indices	**base** R

4.2 Data

We will continue to use the same data we looked at last week. This is a fictional Ebola outbreak, expanded from the ebola_sim practice dataset in the `outbreaks` package.

```
linelist <- import("linelist_cleaned.rds")
```

The first 50 rows of `linelist`:

```
   case_id generation date_infection date_onset date_hospitalisation
1   5fe599          4     2014-05-08 2014-05-13           2014-05-15
2   8689b7          4           <NA> 2014-05-13           2014-05-14
3   11f8ea          2           <NA> 2014-05-16           2014-05-18
4   b8812a          3     2014-05-04 2014-05-18           2014-05-20
5   893f25          3     2014-05-18 2014-05-21           2014-05-22
```

6	be99c8	3	2014-05-03	2014-05-22	2014-05-23
7	07e3e8	4	2014-05-22	2014-05-27	2014-05-29
8	369449	4	2014-05-28	2014-06-02	2014-06-03
9	f393b4	4	<NA>	2014-06-05	2014-06-06
10	1389ca	4	<NA>	2014-06-05	2014-06-07
11	2978ac	4	2014-05-30	2014-06-06	2014-06-08
12	57a565	4	2014-05-28	2014-06-13	2014-06-15
13	fc15ef	6	2014-06-14	2014-06-16	2014-06-17
14	2eaa9a	5	2014-06-07	2014-06-17	2014-06-17
15	bbfa93	6	2014-06-09	2014-06-18	2014-06-20
16	c97dd9	9	<NA>	2014-06-19	2014-06-19
17	f50e8a	10	<NA>	2014-06-22	2014-06-23
18	3a7673	8	<NA>	2014-06-23	2014-06-24
19	7f5a01	7	2014-06-23	2014-06-25	2014-06-27
20	ddddee	6	2014-06-18	2014-06-26	2014-06-28
21	99e8fa	7	2014-06-24	2014-06-28	2014-06-29
22	567136	6	<NA>	2014-07-02	2014-07-03
23	9371a9	8	<NA>	2014-07-08	2014-07-09
24	bc2adf	6	2014-07-03	2014-07-09	2014-07-09
25	403057	10	<NA>	2014-07-09	2014-07-11
26	8bd1e8	8	2014-07-10	2014-07-10	2014-07-11
27	f327be	6	2014-06-14	2014-07-12	2014-07-13
28	42e1a9	12	<NA>	2014-07-12	2014-07-14
29	90e5fe	5	2014-06-18	2014-07-13	2014-07-14
30	959170	8	2014-06-29	2014-07-13	2014-07-13
31	8ebf6e	7	2014-07-02	2014-07-14	2014-07-14
32	e56412	9	2014-07-12	2014-07-15	2014-07-17
33	6d788e	11	2014-07-12	2014-07-16	2014-07-17
34	a47529	5	2014-06-13	2014-07-17	2014-07-18
35	67be4e	8	2014-07-15	2014-07-17	2014-07-19
36	da8ecb	5	2014-06-20	2014-07-18	2014-07-20
37	148f18	6	<NA>	2014-07-19	2014-07-20
38	2cb9a5	11	<NA>	2014-07-22	2014-07-22
39	f5c142	7	2014-07-20	2014-07-22	2014-07-24
40	70a9fe	9	<NA>	2014-07-24	2014-07-26
41	3ad520	7	2014-07-12	2014-07-24	2014-07-24
42	062638	8	2014-07-19	2014-07-25	2014-07-27
43	c76676	9	2014-07-18	2014-07-25	2014-07-25
44	baacc1	12	2014-07-18	2014-07-27	2014-07-27
45	497372	13	2014-07-27	2014-07-29	2014-07-31
46	23e499	9	<NA>	2014-07-30	2014-08-01
47	38cc4a	8	2014-07-19	<NA>	2014-08-03
48	3789ee	10	2014-07-26	2014-08-01	2014-08-02

```
49  c71dcd          8     2014-07-24 2014-08-02           2014-08-02
50  6b70f0          7           <NA> 2014-08-03           2014-08-04
   date_outcome outcome gender age age_unit age_years age_cat age_cat5
1          <NA>    <NA>      m   2    years         2     0-4      0-4
2    2014-05-18 Recover      f   3    years         3     0-4      0-4
3    2014-05-30 Recover      m  56    years        56   50-69    55-59
4          <NA>    <NA>      f  18    years        18   15-19    15-19
5    2014-05-29 Recover      m   3    years         3     0-4      0-4
6    2014-05-24 Recover      f  16    years        16   15-19    15-19
7    2014-06-01 Recover      f  16    years        16   15-19    15-19
8    2014-06-07   Death      f   0    years         0     0-4      0-4
9    2014-06-18 Recover      m  61    years        61   50-69    60-64
10   2014-06-09   Death      f  27    years        27   20-29    25-29
11   2014-06-15   Death      m  12    years        12   10-14    10-14
12         <NA>   Death      m  42    years        42   30-49    40-44
13   2014-07-09 Recover      m  19    years        19   15-19    15-19
14         <NA> Recover      f   7    years         7     5-9      5-9
15   2014-06-30    <NA>      f   7    years         7     5-9      5-9
16   2014-07-11 Recover      m  13    years        13   10-14    10-14
17   2014-07-01    <NA>      f  35    years        35   30-49    35-39
18   2014-06-25    <NA>      f  17    years        17   15-19    15-19
19   2014-07-06   Death      f  11    years        11   10-14    10-14
20   2014-07-02   Death      f  11    years        11   10-14    10-14
21   2014-07-09 Recover      m  19    years        19   15-19    15-19
22   2014-07-07    <NA>      m  54    years        54   50-69    50-54
23   2014-07-20    <NA>      f  14    years        14   10-14    10-14
24         <NA>    <NA>      m  28    years        28   20-29    25-29
25   2014-07-22   Death      f   6    years         6     5-9      5-9
26   2014-07-16    <NA>      m   3    years         3     0-4      0-4
27   2014-07-14   Death      m  31    years        31   30-49    30-34
28   2014-07-20   Death      f   6    years         6     5-9      5-9
29   2014-07-16    <NA>      m  67    years        67   50-69    65-69
30   2014-07-19   Death      f  14    years        14   10-14    10-14
31   2014-07-27 Recover      f  10    years        10   10-14    10-14
32   2014-07-19   Death      f  21    years        21   20-29    20-24
33         <NA> Recover      m  20    years        20   20-29    20-24
34   2014-07-26   Death      m  45    years        45   30-49    45-49
35   2014-08-14 Recover      f   1    years         1     0-4      0-4
36   2014-08-01    <NA>      m  12    years        12   10-14    10-14
37   2014-07-23   Death      f   3    years         3     0-4      0-4
38   2014-08-28 Recover      f  15    years        15   15-19    15-19
39   2014-07-28 Recover      f  20    years        20   20-29    20-24
40   2014-07-19   Death      m  36    years        36   30-49    35-39
```

```
41         <NA>    <NA>      f   7    years         7     5-9      5-9
42   2014-08-03    <NA>      m  13    years        13   10-14    10-14
43         <NA>   Death      f  14    years        14   10-14    10-14
44         <NA>   Death      m   3    years         3     0-4      0-4
45         <NA>   Death      m  10    years        10   10-14    10-14
46   2014-08-06   Death      f   1    years         1     0-4      0-4
47   2014-08-21 Recover      m   0    years         0     0-4      0-4
48   2014-09-13    <NA>      f  20    years        20   20-29    20-24
49   2014-08-04   Death      m  26    years        26   20-29    25-29
50         <NA>   Death      m  14    years        14   10-14    10-14
                              hospital       lon      lat infector  source
1                                Other -13.21574 8.468973   f547d6   other
2                              Missing -13.21523 8.451719     <NA>    <NA>
3  St. Mark's Maternity Hospital (SMMH) -13.21291 8.464817     <NA>    <NA>
4                         Port Hospital -13.23637 8.475476   f90f5f   other
5                     Military Hospital -13.22286 8.460824   11f8ea   other
6                         Port Hospital -13.22263 8.461831   aec8ec   other
7                              Missing -13.23315 8.462729   893f25   other
8                              Missing -13.23210 8.461444   133ee7   other
9                              Missing -13.22255 8.461913     <NA>    <NA>
10                             Missing -13.25722 8.472923     <NA>    <NA>
11                        Port Hospital -13.22063 8.484016   996f3a   other
12                    Military Hospital -13.25399 8.458371   133ee7   other
13                             Missing -13.23851 8.477617   37a6f6   other
14                             Missing -13.20939 8.475702   9f6884   other
15                               Other -13.21573 8.477799   4802b1   other
16                        Port Hospital -13.22434 8.471451     <NA>    <NA>
17                        Port Hospital -13.23361 8.478048     <NA>    <NA>
18                        Port Hospital -13.21422 8.485280     <NA>    <NA>
19                             Missing -13.23397 8.469575   a75c7f   other
20                               Other -13.25356 8.459574   8e104d   other
21                        Port Hospital -13.22501 8.474049   ab634e   other
22                        Port Hospital -13.21607 8.488029     <NA>    <NA>
23 St. Mark's Maternity Hospital (SMMH) -13.26807 8.473437     <NA>    <NA>
24                             Missing -13.22667 8.484083   b799eb   other
25                               Other -13.21602 8.462422     <NA>    <NA>
26                             Missing -13.24826 8.470268   5d9e4d   other
27 St. Mark's Maternity Hospital (SMMH) -13.21563 8.463984   a15e13   other
28                    Military Hospital -13.21424 8.464135     <NA>    <NA>
29                        Port Hospital -13.26149 8.456231   ea3740   other
30                     Central Hospital -13.24530 8.483346   beb26e funeral
31                    Military Hospital -13.26306 8.474940   567136   other
32                     Central Hospital -13.23433 8.478321   894024 funeral
```

```
33                                Missing -13.21991 8.469393   36e2e7   other
34                     Military Hospital -13.22273 8.484806   a2086d   other
35                                 Other -13.23431 8.471212   7baf73   other
36                                Missing -13.21878 8.484384   eb2277 funeral
37                                Missing -13.24837 8.484662     <NA>    <NA>
38                           Port Hospital -13.20975 8.477142     <NA>    <NA>
39                           Port Hospital -13.26809 8.462381   d6584f   other
40                           Port Hospital -13.25875 8.455686     <NA>    <NA>
41                                Missing -13.26264 8.463288   312ecf   other
42                      Central Hospital -13.26972 8.479407   52ea64   other
43                     Military Hospital -13.22090 8.463539   cfd79c   other
44                                 Other -13.23307 8.461790   d145b7   other
45                                 Other -13.26809 8.475087   174288   other
46                                 Other -13.25472 8.458258     <NA>    <NA>
47                                Missing -13.25737 8.453257   53608c funeral
48 St. Mark's Maternity Hospital (SMMH) -13.21374 8.473257   3b096b   other
49 St. Mark's Maternity Hospital (SMMH) -13.21760 8.479116   f5c142   other
50                                Missing -13.24864 8.484803     <NA>    <NA>
   wt_kg ht_cm ct_blood fever chills cough aches vomit temp time_admission
1     27    48       22    no     no   yes    no   yes 36.8          <NA>
2     25    59       22  <NA>   <NA>  <NA>  <NA>  <NA> 36.9         09:36
3     91   238       21  <NA>   <NA>  <NA>  <NA>  <NA> 36.9         16:48
4     41   135       23    no     no    no    no    no 36.8         11:22
5     36    71       23    no     no   yes    no   yes 36.9         12:60
6     56   116       21    no     no   yes    no   yes 37.6         14:13
7     47    87       21  <NA>   <NA>  <NA>  <NA>  <NA> 37.3         14:33
8      0    11       22    no     no   yes    no   yes 37.0         09:25
9     86   226       22    no     no   yes    no   yes 36.4         11:16
10    69   174       22    no     no   yes    no    no 35.9         10:55
11    67   112       22    no     no   yes    no   yes 36.5         16:03
12    84   186       22    no     no   yes    no    no 36.9         11:14
13    68   174       22    no     no   yes    no    no 36.5         12:42
14    44    90       21    no     no   yes    no    no 37.1         11:06
15    34    91       23    no     no   yes    no   yes 36.5         09:10
16    66   152       22    no     no   yes   yes    no 37.3         08:45
17    78   214       23    no    yes   yes    no    no 37.0          <NA>
18    47   137       21    no     no   yes    no    no 38.0         15:41
19    53   117       22  <NA>   <NA>  <NA>  <NA>  <NA> 38.0         13:34
20    47   131       23    no     no   yes    no    no 36.0         18:58
21    71   150       21    no     no   yes    no   yes 37.0         12:43
22    86   241       23    no     no   yes    no    no 36.7         16:33
23    53   131       21    no    yes   yes    no    no 36.9         14:29
24    69   161       24    no     no   yes    no    no 36.5         07:18
```

```
25    38    80       23  <NA>   <NA>  <NA>  <NA>  <NA> 37.0          08:11
26    46    69       22    no     no   yes    no    no 36.5          16:32
27    68   188       24    no     no   yes    no    no 37.6          16:17
28    37    66       23    no    yes   yes    no    no 36.6          07:32
29   100   233       20  <NA>   <NA>  <NA>  <NA>  <NA> 36.6          17:45
30    56   142       24  <NA>   <NA>  <NA>  <NA>  <NA> 36.2           <NA>
31    50   110       24    no     no   yes    no    no 36.4          13:24
32    57   182       20    no     no   yes    no   yes 37.1          14:43
33    65   164       24  <NA>   <NA>  <NA>  <NA>  <NA> 37.5          02:33
34    72   214       21    no     no   yes    no   yes 37.5          11:36
35    29    26       22    no     no   yes    no   yes 37.4          17:28
36    69   157       21  <NA>   <NA>  <NA>  <NA>  <NA> 36.9          16:27
37    37    39       23  <NA>   <NA>  <NA>  <NA>  <NA> 36.4           <NA>
38    48   154       22    no     no   yes   yes   yes 37.3          20:49
39    54   133       23    no     no   yes   yes   yes 37.0           <NA>
40    71   168       23  <NA>   <NA>  <NA>  <NA>  <NA> 37.8          11:38
41    47   100       23    no     no   yes    no   yes 36.5          14:25
42    61   125       22    no     no   yes    no   yes 37.5          13:42
43    47   123       23  <NA>   <NA>  <NA>  <NA>  <NA> 36.7          21:22
44    35    67       22    no     no   yes    no   yes 37.0          13:33
45    53   134       22    no    yes   yes    no   yes 37.3          19:06
46    16    31       22    no     no   yes    no    no 36.6          17:14
47    13    36       23    no     no   yes    no   yes 36.5          20:09
48    59   125       22    no     no   yes    no   yes 36.6           <NA>
49    69   183       22    no     no    no    no   yes 37.6          10:23
50    67   169       22  <NA>   <NA>  <NA>  <NA>  <NA> 36.8          09:09
         bmi days_onset_hosp
1  117.18750               2
2   71.81844               1
3   16.06525               2
4   22.49657               2
5   71.41440               1
6   41.61712               1
7   62.09539               2
8    0.00000               1
9   16.83765               1
10  22.79033               2
11  53.41199               2
12  24.28026               2
13  22.46003               1
14  54.32099               0
15  41.05784               2
16  28.56648               0
```

```
17  17.03206              1
18  25.04129              1
19  38.71722              2
20  27.38768              2
21  31.55556              1
22  14.80691              1
23  30.88398              1
24  26.61934              0
25  59.37500              2
26  96.61836              1
27  19.23947              1
28  84.94031              2
29  18.41994              1
30  27.77227              0
31  41.32231              0
32  17.20807              2
33  24.16716              1
34  15.72190              1
35 428.99408              2
36  27.99302              2
37 243.26101              1
38  20.23950              0
39  30.52745              2
40  25.15590              2
41  47.00000              0
42  39.04000              2
43  31.06616              0
44  77.96837              0
45  29.51660              2
46 166.49324              2
47 100.30864             NA
48  37.76000              1
49  20.60378              0
50  23.45856              1
```

4.3 Select or re-order columns

Use `select()` from **dplyr** to select the columns you want to retain, and to specify their order in the data frame.

Here are ALL the column names in the linelist at this point in the cleaning pipe chain:

```r
names(linelist)
```

```
 [1] "case_id"             "generation"          "date_infection"
 [4] "date_onset"          "date_hospitalisation" "date_outcome"
 [7] "outcome"             "gender"              "age"
[10] "age_unit"            "age_years"           "age_cat"
[13] "age_cat5"            "hospital"            "lon"
[16] "lat"                 "infector"            "source"
[19] "wt_kg"               "ht_cm"               "ct_blood"
[22] "fever"               "chills"              "cough"
[25] "aches"               "vomit"               "temp"
[28] "time_admission"      "bmi"                 "days_onset_hosp"
```

Keep columns

Select only the columns you want to remain

Put their names in the `select()` command, with no quotation marks. They will appear in the data frame in the order you provide. Note that if you include a column that does not exist, R will return an error (see use of `any_of()` below if you want no error in this situation).

```r
# linelist dataset is piped through select() command, and names() prints just the column n
linelist %>%
  select(case_id, date_onset, date_hospitalisation, fever) %>%
  names()  # display the column names
```

```
[1] "case_id"             "date_onset"          "date_hospitalisation"
[4] "fever"
```

Remove columns

Indicate which columns to remove by placing a minus symbol "-" in front of the column name (e.g. `select(-outcome)`), or a vector of column names (as below). All other columns will be retained.

```r
linelist %>%
  select(-c(date_onset, fever:vomit)) %>% # remove date_onset and all columns from fever t
  names()
```

```
 [1] "case_id"              "generation"           "date_infection"
 [4] "date_hospitalisation" "date_outcome"         "outcome"
 [7] "gender"               "age"                  "age_unit"
[10] "age_years"            "age_cat"              "age_cat5"
[13] "hospital"             "lon"                  "lat"
[16] "infector"             "source"               "wt_kg"
[19] "ht_cm"                "ct_blood"             "temp"
[22] "time_admission"       "bmi"                  "days_onset_hosp"
```

You can also remove a column using **base** R syntax, by defining it as `NULL`. For example:

```
linelist$date_onset <- NULL   # deletes column with base R syntax
```

Standalone

`select()` can also be used as an independent command (not in a pipe chain). In this case, the first argument is the original dataframe to be operated upon.

```
# Create a new linelist with id and age-related columns
linelist_age <- select(linelist, case_id, contains("age"))

# display the column names
names(linelist_age)
```

```
[1] "case_id"   "age"       "age_unit"  "age_years" "age_cat"   "age_cat5"
```

4.4 Column creation and transformation

In addition to selecting columns, we can create new columns with `mutate()`. The syntax is: `mutate(new_column_name = value or transformation)`. `mutate()` can also be used to modify an existing column.

New columns

The most basic `mutate()` command to create a new column might look like this. It creates a new column `new_col` where the value in every row is 10.

```r
linelist <- linelist %>%
  mutate(new_col = 10)
```

You can also reference values in other columns, to perform calculations. Below, a new column `bmi` is created to hold the Body Mass Index (BMI) for each case - as calculated using the formula BMI = kg/m^2, using column `ht_cm` and column `wt_kg`.

```r
linelist <- linelist %>%
  mutate(bmi = wt_kg / (ht_cm/100)^2)
```

If creating multiple new columns, separate each with a comma and new line. Below are examples of new columns, including ones that consist of values from other columns combined using `str_glue()` from the **stringr** package (see page on [Characters and strings].

```r
new_col_demo <- linelist %>%
  mutate(
    new_var_dup    = case_id,              # new column = duplicate/copy another existing c
    new_var_static = 7,                    # new column = all values the same
    new_var_static = new_var_static + 5,  # you can overwrite a column, and it can be a ca
    new_var_paste  = stringr::str_glue("{hospital} on ({date_hospitalisation})") # new col
    ) %>%
  select(case_id, hospital, date_hospitalisation, contains("new"))        # show only new
```

Review the new columns. For demonstration purposes, only the new columns and the columns used to create them are shown:

```
   case_id                              hospital date_hospitalisation
1   5fe599                                 Other           2014-05-15
2   8689b7                               Missing           2014-05-14
3   11f8ea St. Mark's Maternity Hospital (SMMH)           2014-05-18
4   b8812a                          Port Hospital          2014-05-20
5   893f25                     Military Hospital           2014-05-22
6   be99c8                          Port Hospital          2014-05-23
7   07e3e8                               Missing           2014-05-29
8   369449                               Missing           2014-06-03
9   f393b4                               Missing           2014-06-06
10  1389ca                               Missing           2014-06-07
11  2978ac                          Port Hospital          2014-06-08
12  57a565                     Military Hospital           2014-06-15
13  fc15ef                               Missing           2014-06-17
14  2eaa9a                               Missing           2014-06-17
15  bbfa93                                 Other           2014-06-20
```

```
16  c97dd9                          Port Hospital          2014-06-19
17  f50e8a                          Port Hospital          2014-06-23
18  3a7673                          Port Hospital          2014-06-24
19  7f5a01                                Missing          2014-06-27
20  ddddee                                  Other          2014-06-28
21  99e8fa                          Port Hospital          2014-06-29
22  567136                          Port Hospital          2014-07-03
23  9371a9 St. Mark's Maternity Hospital (SMMH)          2014-07-09
24  bc2adf                                Missing          2014-07-09
25  403057                                  Other          2014-07-11
26  8bd1e8                                Missing          2014-07-11
27  f327be St. Mark's Maternity Hospital (SMMH)          2014-07-13
28  42e1a9                       Military Hospital          2014-07-14
29  90e5fe                           Port Hospital          2014-07-14
30  959170                        Central Hospital          2014-07-13
31  8ebf6e                       Military Hospital          2014-07-14
32  e56412                        Central Hospital          2014-07-17
33  6d788e                                Missing          2014-07-17
34  a47529                       Military Hospital          2014-07-18
35  67be4e                                  Other          2014-07-19
36  da8ecb                                Missing          2014-07-20
37  148f18                                Missing          2014-07-20
38  2cb9a5                           Port Hospital          2014-07-22
39  f5c142                          Port Hospital          2014-07-24
40  70a9fe                          Port Hospital          2014-07-26
41  3ad520                                Missing          2014-07-24
42  062638                        Central Hospital          2014-07-27
43  c76676                       Military Hospital          2014-07-25
44  baacc1                                  Other          2014-07-27
45  497372                                  Other          2014-07-31
46  23e499                                  Other          2014-08-01
47  38cc4a                                Missing          2014-08-03
48  3789ee St. Mark's Maternity Hospital (SMMH)          2014-08-02
49  c71dcd St. Mark's Maternity Hospital (SMMH)          2014-08-02
50  6b70f0                                Missing          2014-08-04
   new_var_dup new_var_static
1       5fe599             12
2       8689b7             12
3       11f8ea             12
4       b8812a             12
5       893f25             12
6       be99c8             12
7       07e3e8             12
```

8	369449	12
9	f393b4	12
10	1389ca	12
11	2978ac	12
12	57a565	12
13	fc15ef	12
14	2eaa9a	12
15	bbfa93	12
16	c97dd9	12
17	f50e8a	12
18	3a7673	12
19	7f5a01	12
20	ddddee	12
21	99e8fa	12
22	567136	12
23	9371a9	12
24	bc2adf	12
25	403057	12
26	8bd1e8	12
27	f327be	12
28	42e1a9	12
29	90e5fe	12
30	959170	12
31	8ebf6e	12
32	e56412	12
33	6d788e	12
34	a47529	12
35	67be4e	12
36	da8ecb	12
37	148f18	12
38	2cb9a5	12
39	f5c142	12
40	70a9fe	12
41	3ad520	12
42	062638	12
43	c76676	12
44	baacc1	12
45	497372	12
46	23e499	12
47	38cc4a	12
48	3789ee	12
49	c71dcd	12
50	6b70f0	12

```
                                        new_var_paste
1                                   Other on (2014-05-15)
2                                 Missing on (2014-05-14)
3  St. Mark's Maternity Hospital (SMMH) on (2014-05-18)
4                            Port Hospital on (2014-05-20)
5                       Military Hospital on (2014-05-22)
6                            Port Hospital on (2014-05-23)
7                                 Missing on (2014-05-29)
8                                 Missing on (2014-06-03)
9                                 Missing on (2014-06-06)
10                                Missing on (2014-06-07)
11                           Port Hospital on (2014-06-08)
12                      Military Hospital on (2014-06-15)
13                                Missing on (2014-06-17)
14                                Missing on (2014-06-17)
15                                  Other on (2014-06-20)
16                           Port Hospital on (2014-06-19)
17                           Port Hospital on (2014-06-23)
18                           Port Hospital on (2014-06-24)
19                                Missing on (2014-06-27)
20                                  Other on (2014-06-28)
21                           Port Hospital on (2014-06-29)
22                           Port Hospital on (2014-07-03)
23 St. Mark's Maternity Hospital (SMMH) on (2014-07-09)
24                                Missing on (2014-07-09)
25                                  Other on (2014-07-11)
26                                Missing on (2014-07-11)
27 St. Mark's Maternity Hospital (SMMH) on (2014-07-13)
28                      Military Hospital on (2014-07-14)
29                           Port Hospital on (2014-07-14)
30                       Central Hospital on (2014-07-13)
31                      Military Hospital on (2014-07-14)
32                       Central Hospital on (2014-07-17)
33                                Missing on (2014-07-17)
34                      Military Hospital on (2014-07-18)
35                                  Other on (2014-07-19)
36                                Missing on (2014-07-20)
37                                Missing on (2014-07-20)
38                           Port Hospital on (2014-07-22)
39                           Port Hospital on (2014-07-24)
40                           Port Hospital on (2014-07-26)
41                                Missing on (2014-07-24)
42                       Central Hospital on (2014-07-27)
```

```
43                    Military Hospital on (2014-07-25)
44                                Other on (2014-07-27)
45                                Other on (2014-07-31)
46                                Other on (2014-08-01)
47                              Missing on (2014-08-03)
48 St. Mark's Maternity Hospital (SMMH) on (2014-08-02)
49 St. Mark's Maternity Hospital (SMMH) on (2014-08-02)
50                              Missing on (2014-08-04)
```

Transmute

A variation on `mutate()` is the function `transmute()`. This function adds a new column just like `mutate()`, but also drops/removes all other columns that you do not mention within its parentheses.

Convert column class

Columns containing values that are dates, numbers, or logical values (TRUE/FALSE) will only behave as expected if they are correctly classified. There is a difference between "2" of class character and 2 of class numeric!

There are ways to set column class during the import commands, but this is often cumbersome. See the [R Basics] section on object classes to learn more about converting the class of objects and columns.

First, let's run some checks on important columns to see if they are the correct class. We also saw this in the beginning when we ran `skim()`.

Currently, the class of the `age` column is character. To perform quantitative analyses, we need these numbers to be recognized as numeric!

```
class(linelist$age)
```

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

To resolve this, use the ability of `mutate()` to re-define a column with a transformation. We define the column as itself, but converted to a different class. Here is a basic example, converting or ensuring that the column `age` is class Numeric:

```
linelist <- linelist %>%
  mutate(age = as.numeric(age))
```

In a similar way, you can use `as.character()` and `as.logical()`. To convert to class Factor, you can use `factor()`.

4.5 Re-code values

Here are a few scenarios where you need to re-code (change) values:

- to edit one specific value (e.g. one date with an incorrect year or format)
- to reconcile values not spelled the same
- to create a new column of categorical values
- to create a new column of numeric categories (e.g. age categories)

Specific values

To change values manually you can use the `recode()` function within the `mutate()` function.

Imagine there is a nonsensical date in the data (e.g. "2014-14-15"): you could fix the date manually in the raw source data, or, you could write the change into the cleaning pipeline via `mutate()` and `recode()`. The latter is more transparent and reproducible to anyone else seeking to understand or repeat your analysis.

```
# fix incorrect values                    # old value       # new value
linelist <- linelist %>%
  mutate(date_onset = recode(date_onset, "2014-14-15" = "2014-04-15"))
```

The `mutate()` line above can be read as: "mutate the column `date_onset` to equal the column `date_onset` re-coded so that OLD VALUE is changed to NEW VALUE". Note that this pattern (OLD = NEW) for `recode()` is the opposite of most R patterns (new = old). The R development community is working on revising this.

By logic

Below we demonstrate how to re-code values in a column using logic and conditions:

- Using `replace()`, `ifelse()` and `if_else()` for simple logic
- Using `case_when()` for more complex logic

Simple logic

`replace()`

To re-code with simple logical criteria, you can use `replace()` within `mutate()`. `replace()` is a function from **base** R. Use a logic condition to specify the rows to change . The general syntax is:

`mutate(col_to_change = replace(col_to_change, criteria for rows, new value))`.

One common situation to use `replace()` is **changing just one value in one row, using an unique row identifier**. Below, the gender is changed to "Female" in the row where the column `case_id` is "2195".

```
# Example: change gender of one specific observation to "Female"
linelist <- linelist %>%
  mutate(gender = replace(gender, case_id == "2195", "Female"))
```

The equivalent command using **base** R syntax and indexing brackets `[ ]` is below. It reads as "Change the value of the dataframe `linelist`'s column `gender` (for the rows where `linelist`'s column `case_id` has the value '2195') to 'Female' ".

```
linelist$gender[linelist$case_id == "2195"] <- "Female"
```

`ifelse()` and `if_else()`

Another tool for simple logic is `ifelse()` and its partner `if_else()`. However, in most cases for re-coding it is more clear to use `case_when()` (detailed below). These "if else" commands are simplified versions of an `if` and `else` programming statement. The general syntax is: `ifelse(condition, value to return if condition evaluates to TRUE, value to return if condition evaluates to FALSE)`

Below, the column `source_known` is defined. Its value in a given row is set to "known" if the row's value in column `source` is *not* missing. If the value in `source` *is* missing, then the value in `source_known` is set to "unknown".

```
linelist <- linelist %>%
  mutate(source_known = ifelse(!is.na(source), "known", "unknown"))
```

`if_else()` is a special version from **dplyr** that handles dates. Note that if the 'true' value is a date, the 'false' value must also qualify a date, hence using the special value `NA_real_` instead of just `NA`.

```r
# Create a date of death column, which is NA if patient has not died.
linelist <- linelist %>%
  mutate(date_death = if_else(outcome == "Death", date_outcome, NA_real_))
```

Avoid stringing together many ifelse commands... use `case_when()` instead! `case_when()` is much easier to read and you'll make fewer errors.

Outside of the context of a data frame, if you want to have an object used in your code switch its value, consider using `switch()` from **base** R.

Complex logic

Use **dplyr**'s `case_when()` if you are re-coding into many new groups, or if you need to use complex logic statements to re-code values. This function evaluates every row in the data frame, assess whether the rows meets specified criteria, and assigns the correct new value.

`case_when()` commands consist of statements that have a Right-Hand Side (RHS) and a Left-Hand Side (LHS) separated by a "tilde" `~`. The logic criteria are in the left side and the pursuant values are in the right side of each statement. Statements are separated by commas.

For example, here we utilize the columns `age` and `age_unit` to create a column `age_years`:

```r
linelist <- linelist %>%
  mutate(age_years = case_when(
       age_unit == "years"  ~ age,       # if age unit is years
       age_unit == "months" ~ age/12,    # if age unit is months, divide age by 12
       is.na(age_unit)      ~ age))      # if age unit is missing, assume years
                                         # any other circumstance, assign NA (missing)
```

As each row in the data is evaluated, the criteria are applied/evaluated in the order the `case_when()` statements are written - from top-to-bottom. If the top criteria evaluates to `TRUE` for a given row, the RHS value is assigned, and the remaining criteria are not even tested for that row in the data. Thus, it is best to write the most specific criteria first, and the most general last. A data row that does not meet any of the RHS criteria will be assigned `NA`.

Sometimes, you may with to write a final statement that assigns a value for all other scenarios not described by one of the previous lines. To do this, place `TRUE` on the left-side, which will capture any row that did not meet any of the previous criteria. The right-side of this statement could be assigned a value like "check me!" or missing.

Below is another example of `case_when()` used to create a new column with the patient classification, according to a case definition for confirmed and suspect cases:

```
linelist <- linelist %>%
     mutate(case_status = case_when(

          # if patient had lab test and it is positive,
          # then they are marked as a confirmed case
          ct_blood < 20                    ~ "Confirmed",

          # given that a patient does not have a positive lab result,
          # if patient has a "source" (epidemiological link) AND has fever,
          # then they are marked as a suspect case
          !is.na(source) & fever == "yes" ~ "Suspect",

          # any other patient not addressed above
          # is marked for follow up
          TRUE                             ~ "To investigate"))
```

4.6 Numeric categories

Here we describe some special approaches for creating categories from numerical columns. Common examples include age categories, groups of lab values, etc. Here we will discuss:

- `age_categories()`, from the **epikit** package
- `cut()`, from **base** R
- `case_when()`
- quantile breaks with `quantile()` and `ntile()`

Review distribution

For this example we will create an `age_cat` column using the `age_years` column.

```
#check the class of the linelist variable age
class(linelist$age_years)
```

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

`cut()`

The basic syntax within `cut()` is to first provide the numeric column to be cut (`age_years`), and then the *breaks* argument, which is a numeric vector `c()` of break points. Using `cut()`, the resulting column is an ordered factor.

By default, the categorization occurs so that the right/upper side is "open" and inclusive (and the left/lower side is "closed" or exclusive). This is the opposite behavior from the `age_categories()` function. The default labels use the notation "(A, B]", which means A is not included but B is. **Reverse this behavior by providing the `right = TRUE` argument**.

Thus, by default, "0" values are excluded from the lowest group, and categorized as `NA`! "0" values could be infants coded as age 0 so be careful! To change this, add the argument `include.lowest = TRUE` so that any "0" values will be included in the lowest group. The automatically-generated label for the lowest category will then be "[A],B]". Note that if you include the `include.lowest = TRUE` argument **and** `right = TRUE`, the extreme inclusion will now apply to the *highest* break point value and category, not the lowest.

You can provide a vector of customized labels using the `labels =` argument. As these are manually written, be very careful to ensure they are accurate! Check your work using cross-tabulation, as described below.

An example of `cut()` applied to `age_years` to make the new variable `age_cat` is below:

```
# Create new variable, by cutting the numeric age variable
# lower break is excluded but upper break is included in each category
linelist <- linelist %>%
  mutate(
    age_cat = cut(
      age_years,
      breaks = c(0, 5, 10, 15, 20,
                 30, 50, 70, 100),
      include.lowest = TRUE         # include 0 in lowest group
```

```
      ))

# tabulate the number of observations per group
table(linelist$age_cat, useNA = "always")
```

```
 [0,5]   (5,10]  (10,15]  (15,20]  (20,30]  (30,50]  (50,70] (70,100]
  1315     1065      930       696     1013      694       84        5
  <NA>
    86
```

Check your work!!! Verify that each age value was assigned to the correct category by cross-tabulating the numeric and category columns. Examine assignment of boundary values (e.g. 15, if neighboring categories are 10-15 and 16-20).

Quantile breaks

In common understanding, "quantiles" or "percentiles" typically refer to a value below which a proportion of values fall. For example, the 95th percentile of ages in `linelist` would be the age below which 95% of the age fall.

However in common speech, "quartiles" and "deciles" can also refer to the *groups of data* as equally divided into 4, or 10 groups (note there will be one more break point than group).

To get quantile break points, you can use `quantile()` from the **stats** package from **base** R. You provide a numeric vector (e.g. a column in a dataset) and vector of numeric probability values ranging from 0 to 1.0. The break points are returned as a numeric vector. Explore the details of the statistical methodologies by entering `?quantile`.

- If your input numeric vector has any missing values it is best to set `na.rm = TRUE`
- Set `names = FALSE` to get an un-named numeric vector

```
quantile(linelist$age_years,                # specify numeric vector to work on
  probs = c(0, .25, .50, .75, .90, .95),    # specify the percentiles you want
  na.rm = TRUE)                             # ignore missing values
```

```
  0%  25%  50%  75%  90%  95%
 0.0  6.0 13.0 23.0 33.9 41.0
```

You can use the results of `quantile()` as break points in `age_categories()` or `cut()`. Below we create a new column `deciles` using `cut()` where the breaks are defined using `quantiles()` on `age_years`. Below, we display the results using `tabyl()` from **janitor** so you can see the percentages. Note how they are not exactly 10% in each group.

```
linelist %>%                                  # begin with linelist
  mutate(deciles = cut(age_years,             # create new column decile as cut() on column
    breaks = quantile(                        # define cut breaks using quantile()
      age_years,                                 # operate on age_years
      probs = seq(0, 1, by = 0.1),               # 0.0 to 1.0 by 0.1
      na.rm = TRUE),                             # ignore missing values
    include.lowest = TRUE)) %>%               # for cut() include age 0
  janitor::tabyl(deciles)                     # pipe to table to display
```

```
  deciles   n    percent valid_percent
    [0,2] 658 0.11175272    0.11340917
    (2,5] 657 0.11158288    0.11323681
    (5,7] 447 0.07591712    0.07704240
   (7,10] 618 0.10495924    0.10651499
  (10,13] 572 0.09714674    0.09858669
  (13,17] 674 0.11447011    0.11616684
  (17,21] 520 0.08831522    0.08962427
  (21,26] 547 0.09290082    0.09427784
(26,33.9] 528 0.08967391    0.09100310
(33.9,84] 581 0.09867527    0.10013788
     <NA>  86 0.01460598            NA
```

4.7 Filter rows

A typical cleaning step after you have cleaned the columns and re-coded values is to *filter* the data frame for specific rows using the **dplyr** verb `filter()`.

Within `filter()`, specify the logic that must be `TRUE` for a row in the dataset to be kept. Below we show how to filter rows based on simple and complex logical conditions.

Simple filter

This simple example re-defines the dataframe `linelist` as itself, having filtered the rows to meet a logical condition. **Only the rows where the logical statement within the parentheses evaluates to `TRUE` are kept.**

In this example, the logical statement is `gender == "f"`, which is asking whether the value in the column `gender` is equal to "f" (case sensitive).

Before the filter is applied, the number of rows in `linelist` is `nrow(linelist)`.

```
linelist <- linelist %>%
  filter(gender == "f")   # keep only rows where gender is equal to "f"
```

After the filter is applied, the number of rows in `linelist` is `linelist %>% filter(gender == "f") %>% nrow()`.

Complex filter

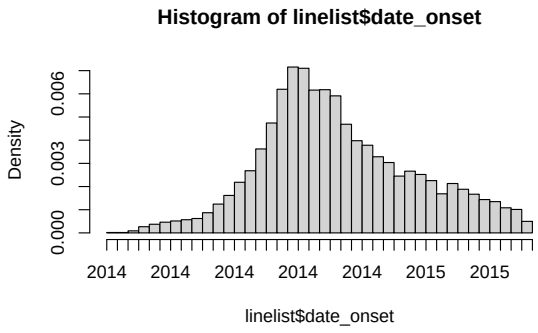
More complex logical statements can be constructed using parentheses `( )`, OR `|`, negate `!`, `%in%`, and AND `&` operators. An example is below:

Note: You can use the `!` operator in front of a logical criteria to negate it. For example, `!is.na(column)` evaluates to true if the column value is *not* missing. Likewise `!column %in% c("a", "b", "c")` evaluates to true if the column value is *not* in the vector.

Examine the data

Below is a simple one-line command to create a histogram of onset dates. See that a second smaller outbreak from 2012-2013 is also included in this raw dataset. **For our analyses, we want to remove entries from this earlier outbreak.**

```
hist(linelist$date_onset, breaks = 50)
```



How filters handle missing numeric and date values

Can we just filter by `date_onset` to rows after June 2013? **Caution! Applying the code `filter(date_onset > as.Date("2013-06-01")))` would remove any rows in the later epidemic with a missing date of onset!**

> ⚠ Conditions with NA
>
> Filtering to greater than (>) or less than (<) a date or number can remove any rows with missing values (`NA`)! This is because `NA` is treated as infinitely large and small.

Standalone

Filtering can also be done as a stand-alone command (not part of a pipe chain). Like other **dplyr** verbs, in this case the first argument must be the dataset itself.

```
# dataframe <- filter(dataframe, condition(s) for rows to keep)

linelist <- filter(linelist, !is.na(case_id))
```

You can also use **base** R to subset using square brackets which reflect the [rows, columns] that you want to retain.

```
# dataframe <- dataframe[row conditions, column conditions] (blank means keep all)

linelist <- linelist[!is.na(case_id), ]
```

4.8 Arrange and sort

Use the **dplyr** function `arrange()` to sort or order the rows by column values.

Simple list the columns in the order they should be sorted on. Specify `.by_group = TRUE` if you want the sorting to to first occur by any *groupings* applied to the data (see page on [Grouping data]).

By default, column will be sorted in "ascending" order (which applies to numeric and also to character columns). You can sort a variable in "descending" order by wrapping it with `desc()`.

Sorting data with `arrange()` is particularly useful when making [Tables for presentation], using `slice()` to take the "top" rows per group, or setting factor level order by order of appearance.

For example, to sort the our linelist rows by `hospital`, then by `date_onset` in descending order, we would use:

```
linelist %>%
   arrange(hospital, desc(date_onset))
```

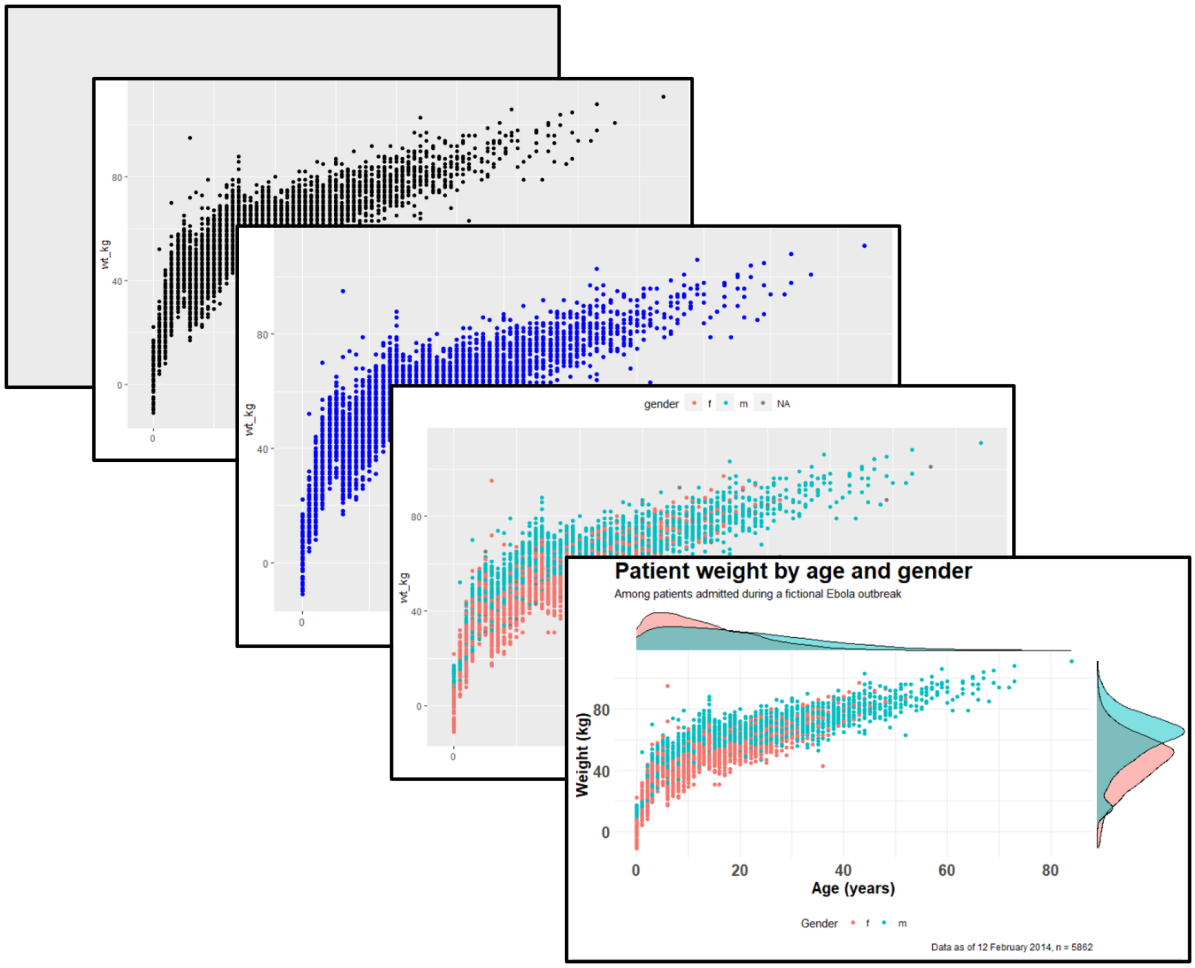
Part III

Session 2: Visualizing Data

5 Data visualization

Extended Materials

You can find the original, extended version of these materials from chapters 30 and 31.



ggplot2 is the most popular data visualisation R package. Its `ggplot()` function is at the core of this package, and this whole approach is colloquially known as *"ggplot"* with the resulting figures sometimes affectionately called "ggplots". The "gg" in these names reflects

the "**g**rammar of **g**raphics" used to construct the figures. **ggplot2** benefits from a wide variety of supplementary R packages that further enhance its functionality.

The data visualization with ggplot cheatsheet from the RStudio website is a great reference to have on-hand when creating pltos. If you want inspiration for ways to creatively visualise your data, we suggest reviewing websites like the R graph gallery and Data-to-viz.

Data Preparation

Import data

We import the dataset of cases from a simulated Ebola epidemic. If you want to follow along, click to download the "clean" linelist (as .rds file).
The first 50 rows of the linelist are displayed below. We will focus on the continuous variables `age`, `wt_kg` (weight in kilos), `ct_blood` (CT values), and `days_onset_hosp` (difference between onset date and hospitalisation).

	case_id	generation	date_infection	date_onset	date_hospitalisation
1	5fe599	4	2014-05-08	2014-05-13	2014-05-15
2	8689b7	4	<NA>	2014-05-13	2014-05-14
3	11f8ea	2	<NA>	2014-05-16	2014-05-18
4	b8812a	3	2014-05-04	2014-05-18	2014-05-20
5	893f25	3	2014-05-18	2014-05-21	2014-05-22
6	be99c8	3	2014-05-03	2014-05-22	2014-05-23
7	07e3e8	4	2014-05-22	2014-05-27	2014-05-29
8	369449	4	2014-05-28	2014-06-02	2014-06-03
9	f393b4	4	<NA>	2014-06-05	2014-06-06
10	1389ca	4	<NA>	2014-06-05	2014-06-07
11	2978ac	4	2014-05-30	2014-06-06	2014-06-08
12	57a565	4	2014-05-28	2014-06-13	2014-06-15
13	fc15ef	6	2014-06-14	2014-06-16	2014-06-17
14	2eaa9a	5	2014-06-07	2014-06-17	2014-06-17
15	bbfa93	6	2014-06-09	2014-06-18	2014-06-20
16	c97dd9	9	<NA>	2014-06-19	2014-06-19
17	f50e8a	10	<NA>	2014-06-22	2014-06-23
18	3a7673	8	<NA>	2014-06-23	2014-06-24
19	7f5a01	7	2014-06-23	2014-06-25	2014-06-27
20	ddddee	6	2014-06-18	2014-06-26	2014-06-28
21	99e8fa	7	2014-06-24	2014-06-28	2014-06-29
22	567136	6	<NA>	2014-07-02	2014-07-03
23	9371a9	8	<NA>	2014-07-08	2014-07-09
24	bc2adf	6	2014-07-03	2014-07-09	2014-07-09
25	403057	10	<NA>	2014-07-09	2014-07-11

```
26  8bd1e8          8     2014-07-10 2014-07-10           2014-07-11
27  f327be          6     2014-06-14 2014-07-12           2014-07-13
28  42e1a9         12           <NA> 2014-07-12           2014-07-14
29  90e5fe          5     2014-06-18 2014-07-13           2014-07-14
30  959170          8     2014-06-29 2014-07-13           2014-07-13
31  8ebf6e          7     2014-07-02 2014-07-14           2014-07-14
32  e56412          9     2014-07-12 2014-07-15           2014-07-17
33  6d788e         11     2014-07-12 2014-07-16           2014-07-17
34  a47529          5     2014-06-13 2014-07-17           2014-07-18
35  67be4e          8     2014-07-15 2014-07-17           2014-07-19
36  da8ecb          5     2014-06-20 2014-07-18           2014-07-20
37  148f18          6           <NA> 2014-07-19           2014-07-20
38  2cb9a5         11           <NA> 2014-07-22           2014-07-22
39  f5c142          7     2014-07-20 2014-07-22           2014-07-24
40  70a9fe          9           <NA> 2014-07-24           2014-07-26
41  3ad520          7     2014-07-12 2014-07-24           2014-07-24
42  062638          8     2014-07-19 2014-07-25           2014-07-27
43  c76676          9     2014-07-18 2014-07-25           2014-07-25
44  baacc1         12     2014-07-18 2014-07-27           2014-07-27
45  497372         13     2014-07-27 2014-07-29           2014-07-31
46  23e499          9           <NA> 2014-07-30           2014-08-01
47  38cc4a          8     2014-07-19       <NA>           2014-08-03
48  3789ee         10     2014-07-26 2014-08-01           2014-08-02
49  c71dcd          8     2014-07-24 2014-08-02           2014-08-02
50  6b70f0          7           <NA> 2014-08-03           2014-08-04
   date_outcome outcome gender age age_unit age_years age_cat age_cat5
1          <NA>    <NA>      m   2    years         2     0-4      0-4
2    2014-05-18 Recover      f   3    years         3     0-4      0-4
3    2014-05-30 Recover      m  56    years        56   50-69    55-59
4          <NA>    <NA>      f  18    years        18   15-19    15-19
5    2014-05-29 Recover      m   3    years         3     0-4      0-4
6    2014-05-24 Recover      f  16    years        16   15-19    15-19
7    2014-06-01 Recover      f  16    years        16   15-19    15-19
8    2014-06-07   Death      f   0    years         0     0-4      0-4
9    2014-06-18 Recover      m  61    years        61   50-69    60-64
10   2014-06-09   Death      f  27    years        27   20-29    25-29
11   2014-06-15   Death      m  12    years        12   10-14    10-14
12         <NA>   Death      m  42    years        42   30-49    40-44
13   2014-07-09 Recover      m  19    years        19   15-19    15-19
14         <NA> Recover      f   7    years         7     5-9      5-9
15   2014-06-30    <NA>      f   7    years         7     5-9      5-9
```

```
16   2014-07-11 Recover      m  13    years        13   10-14    10-14
17   2014-07-01    <NA>      f  35    years        35   30-49    35-39
18   2014-06-25    <NA>      f  17    years        17   15-19    15-19
19   2014-07-06   Death      f  11    years        11   10-14    10-14
20   2014-07-02   Death      f  11    years        11   10-14    10-14
21   2014-07-09 Recover      m  19    years        19   15-19    15-19
22   2014-07-07    <NA>      m  54    years        54   50-69    50-54
23   2014-07-20    <NA>      f  14    years        14   10-14    10-14
24         <NA>    <NA>      m  28    years        28   20-29    25-29
25   2014-07-22   Death      f   6    years         6     5-9      5-9
26   2014-07-16    <NA>      m   3    years         3     0-4      0-4
27   2014-07-14   Death      m  31    years        31   30-49    30-34
28   2014-07-20   Death      f   6    years         6     5-9      5-9
29   2014-07-16    <NA>      m  67    years        67   50-69    65-69
30   2014-07-19   Death      f  14    years        14   10-14    10-14
31   2014-07-27 Recover      f  10    years        10   10-14    10-14
32   2014-07-19   Death      f  21    years        21   20-29    20-24
33         <NA> Recover      m  20    years        20   20-29    20-24
34   2014-07-26   Death      m  45    years        45   30-49    45-49
35   2014-08-14 Recover      f   1    years         1     0-4      0-4
36   2014-08-01    <NA>      m  12    years        12   10-14    10-14
37   2014-07-23   Death      f   3    years         3     0-4      0-4
38   2014-08-28 Recover      f  15    years        15   15-19    15-19
39   2014-07-28 Recover      f  20    years        20   20-29    20-24
40   2014-07-19   Death      m  36    years        36   30-49    35-39
41         <NA>    <NA>      f   7    years         7     5-9      5-9
42   2014-08-03    <NA>      m  13    years        13   10-14    10-14
43         <NA>   Death      f  14    years        14   10-14    10-14
44         <NA>   Death      m   3    years         3     0-4      0-4
45         <NA>   Death      m  10    years        10   10-14    10-14
46   2014-08-06   Death      f   1    years         1     0-4      0-4
47   2014-08-21 Recover      m   0    years         0     0-4      0-4
48   2014-09-13    <NA>      f  20    years        20   20-29    20-24
49   2014-08-04   Death      m  26    years        26   20-29    25-29
50         <NA>   Death      m  14    years        14   10-14    10-14
                                hospital       lon      lat infector  source
1                                  Other -13.21574 8.468973   f547d6   other
2                                Missing -13.21523 8.451719     <NA>    <NA>
3  St. Mark's Maternity Hospital (SMMH) -13.21291 8.464817     <NA>    <NA>
4                           Port Hospital -13.23637 8.475476   f90f5f   other
5                       Military Hospital -13.22286 8.460824   11f8ea   other
```

```
6                          Port Hospital -13.22263 8.461831   aec8ec   other
7                                Missing -13.23315 8.462729   893f25   other
8                                Missing -13.23210 8.461444   133ee7   other
9                                Missing -13.22255 8.461913     <NA>    <NA>
10                               Missing -13.25722 8.472923     <NA>    <NA>
11                         Port Hospital -13.22063 8.484016   996f3a   other
12                     Military Hospital -13.25399 8.458371   133ee7   other
13                               Missing -13.23851 8.477617   37a6f6   other
14                               Missing -13.20939 8.475702   9f6884   other
15                                 Other -13.21573 8.477799   4802b1   other
16                         Port Hospital -13.22434 8.471451     <NA>    <NA>
17                         Port Hospital -13.23361 8.478048     <NA>    <NA>
18                         Port Hospital -13.21422 8.485280     <NA>    <NA>
19                               Missing -13.23397 8.469575   a75c7f   other
20                                 Other -13.25356 8.459574   8e104d   other
21                         Port Hospital -13.22501 8.474049   ab634e   other
22                         Port Hospital -13.21607 8.488029     <NA>    <NA>
23 St. Mark's Maternity Hospital (SMMH) -13.26807 8.473437     <NA>    <NA>
24                               Missing -13.22667 8.484083   b799eb   other
25                                 Other -13.21602 8.462422     <NA>    <NA>
26                               Missing -13.24826 8.470268   5d9e4d   other
27 St. Mark's Maternity Hospital (SMMH) -13.21563 8.463984   a15e13   other
28                     Military Hospital -13.21424 8.464135     <NA>    <NA>
29                         Port Hospital -13.26149 8.456231   ea3740   other
30                      Central Hospital -13.24530 8.483346   beb26e funeral
31                     Military Hospital -13.26306 8.474940   567136   other
32                      Central Hospital -13.23433 8.478321   894024 funeral
33                               Missing -13.21991 8.469393   36e2e7   other
34                     Military Hospital -13.22273 8.484806   a2086d   other
35                                 Other -13.23431 8.471212   7baf73   other
36                               Missing -13.21878 8.484384   eb2277 funeral
37                               Missing -13.24837 8.484662     <NA>    <NA>
38                         Port Hospital -13.20975 8.477142     <NA>    <NA>
39                         Port Hospital -13.26809 8.462381   d6584f   other
40                         Port Hospital -13.25875 8.455686     <NA>    <NA>
41                               Missing -13.26264 8.463288   312ecf   other
42                      Central Hospital -13.26972 8.479407   52ea64   other
43                     Military Hospital -13.22090 8.463539   cfd79c   other
44                                 Other -13.23307 8.461790   d145b7   other
45                                 Other -13.26809 8.475087   174288   other
46                                 Other -13.25472 8.458258     <NA>    <NA>
```

```
47                                Missing -13.25737 8.453257   53608c funeral
48 St. Mark's Maternity Hospital (SMMH) -13.21374 8.473257   3b096b   other
49 St. Mark's Maternity Hospital (SMMH) -13.21760 8.479116   f5c142   other
50                                Missing -13.24864 8.484803     <NA>    <NA>
   wt_kg ht_cm ct_blood fever chills cough aches vomit temp time_admission
1     27    48       22    no     no   yes    no   yes 36.8           <NA>
2     25    59       22  <NA>   <NA>  <NA>  <NA>  <NA> 36.9          09:36
3     91   238       21  <NA>   <NA>  <NA>  <NA>  <NA> 36.9          16:48
4     41   135       23    no     no    no    no    no 36.8          11:22
5     36    71       23    no     no   yes    no   yes 36.9          12:60
6     56   116       21    no     no   yes    no   yes 37.6          14:13
7     47    87       21  <NA>   <NA>  <NA>  <NA>  <NA> 37.3          14:33
8      0    11       22    no     no   yes    no   yes 37.0          09:25
9     86   226       22    no     no   yes    no   yes 36.4          11:16
10    69   174       22    no     no   yes    no    no 35.9          10:55
11    67   112       22    no     no   yes    no   yes 36.5          16:03
12    84   186       22    no     no   yes    no    no 36.9          11:14
13    68   174       22    no     no   yes    no    no 36.5          12:42
14    44    90       21    no     no   yes    no    no 37.1          11:06
15    34    91       23    no     no   yes    no   yes 36.5          09:10
16    66   152       22    no     no   yes   yes    no 37.3          08:45
17    78   214       23    no    yes   yes    no    no 37.0           <NA>
18    47   137       21    no     no   yes    no    no 38.0          15:41
19    53   117       22  <NA>   <NA>  <NA>  <NA>  <NA> 38.0          13:34
20    47   131       23    no     no   yes    no    no 36.0          18:58
21    71   150       21    no     no   yes    no   yes 37.0          12:43
22    86   241       23    no     no   yes    no    no 36.7          16:33
23    53   131       21    no    yes   yes    no    no 36.9          14:29
24    69   161       24    no     no   yes    no    no 36.5          07:18
25    38    80       23  <NA>   <NA>  <NA>  <NA>  <NA> 37.0          08:11
26    46    69       22    no     no   yes    no    no 36.5          16:32
27    68   188       24    no     no   yes    no    no 37.6          16:17
28    37    66       23    no    yes   yes    no    no 36.6          07:32
29   100   233       20  <NA>   <NA>  <NA>  <NA>  <NA> 36.6          17:45
30    56   142       24  <NA>   <NA>  <NA>  <NA>  <NA> 36.2           <NA>
31    50   110       24    no     no   yes    no    no 36.4          13:24
32    57   182       20    no     no   yes    no   yes 37.1          14:43
33    65   164       24  <NA>   <NA>  <NA>  <NA>  <NA> 37.5          02:33
34    72   214       21    no     no   yes    no   yes 37.5          11:36
35    29    26       22    no     no   yes    no   yes 37.4          17:28
36    69   157       21  <NA>   <NA>  <NA>  <NA>  <NA> 36.9          16:27
```

```
37    37    39       23  <NA>   <NA>  <NA>  <NA>  <NA> 36.4          <NA>
38    48   154       22    no     no   yes   yes   yes 37.3         20:49
39    54   133       23    no     no   yes   yes   yes 37.0          <NA>
40    71   168       23  <NA>   <NA>  <NA>  <NA>  <NA> 37.8         11:38
41    47   100       23    no     no   yes    no   yes 36.5         14:25
42    61   125       22    no     no   yes    no   yes 37.5         13:42
43    47   123       23  <NA>   <NA>  <NA>  <NA>  <NA> 36.7         21:22
44    35    67       22    no     no   yes    no   yes 37.0         13:33
45    53   134       22    no    yes   yes    no   yes 37.3         19:06
46    16    31       22    no     no   yes    no    no 36.6         17:14
47    13    36       23    no     no   yes    no   yes 36.5         20:09
48    59   125       22    no     no   yes    no   yes 36.6          <NA>
49    69   183       22    no     no    no    no   yes 37.6         10:23
50    67   169       22  <NA>   <NA>  <NA>  <NA>  <NA> 36.8         09:09
         bmi days_onset_hosp
1  117.18750               2
2   71.81844               1
3   16.06525               2
4   22.49657               2
5   71.41440               1
6   41.61712               1
7   62.09539               2
8    0.00000               1
9   16.83765               1
10  22.79033               2
11  53.41199               2
12  24.28026               2
13  22.46003               1
14  54.32099               0
15  41.05784               2
16  28.56648               0
17  17.03206               1
18  25.04129               1
19  38.71722               2
20  27.38768               2
21  31.55556               1
22  14.80691               1
23  30.88398               1
24  26.61934               0
25  59.37500               2
26  96.61836               1
```

```
27  19.23947              1
28  84.94031              2
29  18.41994              1
30  27.77227              0
31  41.32231              0
32  17.20807              2
33  24.16716              1
34  15.72190              1
35 428.99408              2
36  27.99302              2
37 243.26101              1
38  20.23950              0
39  30.52745              2
40  25.15590              2
41  47.00000              0
42  39.04000              2
43  31.06616              0
44  77.96837              0
45  29.51660              2
46 166.49324              2
47 100.30864             NA
48  37.76000              1
49  20.60378              0
50  23.45856              1
```

General cleaning

Some simple ways we can prepare our data to make it better for plotting can include making the contents of the data better for display - which does not necessarily equate to better for data manipulation. For example:

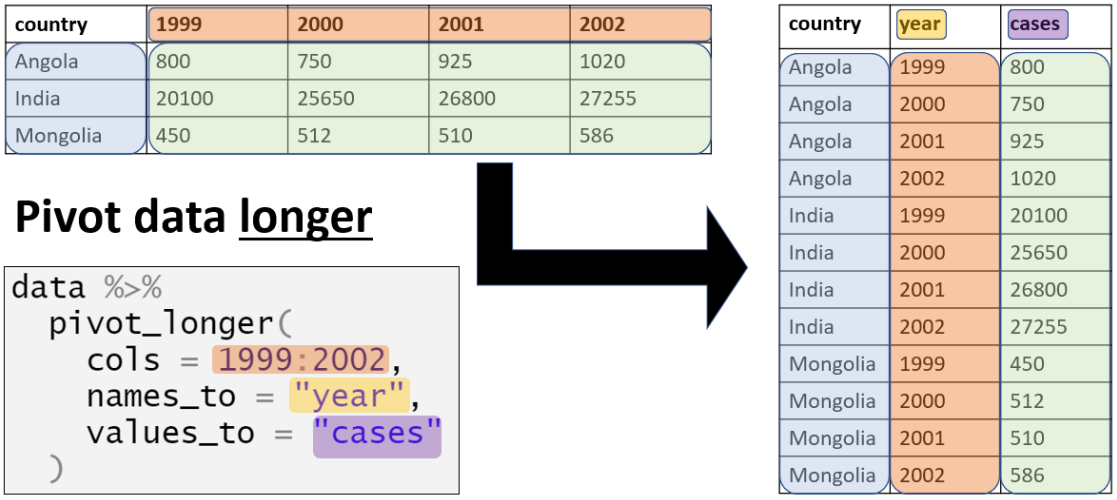
- Replace `NA` values in a character column with the character string "Unknown"
- Consider converting column to class *factor* so their values have prescribed ordinal levels
- Clean some columns so that their "data friendly" values with underscores etc are changed to normal text or title case.

Here are some examples of this in action:

```r
# make display version of columns with more friendly names
linelist <- linelist %>%
  mutate(
    gender_disp = case_when(gender == "m" ~ "Male",         # m to Male
                            gender == "f" ~ "Female",       # f to Female,
                            is.na(gender) ~ "Unknown"),     # NA to Unknown

    outcome_disp = replace_na(outcome, "Unknown")           # replace NA outcome with "un
  )
```

Pivoting longer

As a matter of data structure, for **ggplot2** we often also want to pivot our data into *longer* formats. Read more about this is the page on [Pivoting data].



For example, say that we want to plot data that are in a "wide" format, such as for each case in the `linelist` and their symptoms. Below we create a mini-linelist called `symptoms_data` that contains only the `case_id` and symptoms columns.

```r
symptoms_data <- linelist %>%
  select(c(case_id, fever, chills, cough, aches, vomit))
```

Here is how the first 50 rows of this mini-linelist look - see how they are formatted "wide" with each symptom as a column:

```
 case_id fever chills cough aches vomit
```

```
1   5fe599    no     no   yes    no   yes
2   8689b7  <NA>   <NA>  <NA>  <NA>  <NA>
3   11f8ea  <NA>   <NA>  <NA>  <NA>  <NA>
4   b8812a    no     no    no    no    no
5   893f25    no     no   yes    no   yes
6   be99c8    no     no   yes    no   yes
7   07e3e8  <NA>   <NA>  <NA>  <NA>  <NA>
8   369449    no     no   yes    no   yes
9   f393b4    no     no   yes    no   yes
10  1389ca    no     no   yes    no    no
11  2978ac    no     no   yes    no   yes
12  57a565    no     no   yes    no    no
13  fc15ef    no     no   yes    no    no
14  2eaa9a    no     no   yes    no    no
15  bbfa93    no     no   yes    no   yes
16  c97dd9    no     no   yes   yes    no
17  f50e8a    no    yes   yes    no    no
18  3a7673    no     no   yes    no    no
19  7f5a01  <NA>   <NA>  <NA>  <NA>  <NA>
20  ddddee    no     no   yes    no    no
21  99e8fa    no     no   yes    no   yes
22  567136    no     no   yes    no    no
23  9371a9    no    yes   yes    no    no
24  bc2adf    no     no   yes    no    no
25  403057  <NA>   <NA>  <NA>  <NA>  <NA>
26  8bd1e8    no     no   yes    no    no
27  f327be    no     no   yes    no    no
28  42e1a9    no    yes   yes    no    no
29  90e5fe  <NA>   <NA>  <NA>  <NA>  <NA>
30  959170  <NA>   <NA>  <NA>  <NA>  <NA>
31  8ebf6e    no     no   yes    no    no
32  e56412    no     no   yes    no   yes
33  6d788e  <NA>   <NA>  <NA>  <NA>  <NA>
34  a47529    no     no   yes    no   yes
35  67be4e    no     no   yes    no   yes
36  da8ecb  <NA>   <NA>  <NA>  <NA>  <NA>
37  148f18  <NA>   <NA>  <NA>  <NA>  <NA>
38  2cb9a5    no     no   yes   yes   yes
39  f5c142    no     no   yes   yes   yes
40  70a9fe  <NA>   <NA>  <NA>  <NA>  <NA>
41  3ad520    no     no   yes    no   yes
```

```
42  062638    no     no   yes    no   yes
43  c76676  <NA>   <NA>  <NA>  <NA>  <NA>
44  baacc1    no     no   yes    no   yes
45  497372    no    yes   yes    no   yes
46  23e499    no     no   yes    no    no
47  38cc4a    no     no   yes    no   yes
48  3789ee    no     no   yes    no   yes
49  c71dcd    no     no    no    no   yes
50  6b70f0  <NA>   <NA>  <NA>  <NA>  <NA>
```

If we wanted to plot the number of cases with specific symptoms, we are limited by the fact that each symptom is a specific column. However, we can *pivot* the symptoms columns to a longer format like this:

```
symptoms_data_long <- symptoms_data %>%    # begin with "mini" linelist called symptoms_

  pivot_longer(
    cols = -case_id,                       # pivot all columns except case_id (all the s
    names_to = "symptom_name",             # assign name for new column that holds the s
    values_to = "symptom_is_present") %>%  # assign name for new column that holds the v

  mutate(symptom_is_present = replace_na(symptom_is_present, "unknown")) # convert NA to
```

Here are the first 50 rows. Note that case has 5 rows - one for each possible symptom. The new columns `symptom_name` and `symptom_is_present` are the result of the pivot. Note that this format may not be very useful for other operations, but is useful for plotting.

```
# A tibble: 50 x 3
   case_id symptom_name symptom_is_present
   <chr>   <chr>        <chr>
 1 5fe599  fever        no
 2 5fe599  chills       no
 3 5fe599  cough        yes
 4 5fe599  aches        no
 5 5fe599  vomit        yes
 6 8689b7  fever        unknown
 7 8689b7  chills       unknown
 8 8689b7  cough        unknown
 9 8689b7  aches        unknown
10 8689b7  vomit        unknown
# i 40 more rows
```

5.1 Basics of ggplot

"Grammar of graphics" - ggplot2

Plotting with **ggplot2** is based on "adding" plot layers and design elements on top of one another, with each command added to the previous ones with a plus symbol (+). The result is a multi-layer plot object that can be saved, modified, printed, exported, etc.

> The idea behind the Grammar of Graphics it is that you can build every graph from the same 3 components: (1) a data set, (2) a coordinate system, and (3) geoms — i.e. visual marks that represent data points [source]

ggplot objects can be highly complex, but the basic order of layers will usually look like this:

1. Begin with the baseline `ggplot()` command - this "opens" the ggplot and allow subsequent functions to be added with `+`. Typically the dataset is also specified in this command

2. Add "geom" layers - these functions visualize the data as *geometries* (*shapes*), e.g. as a bar graph, line plot, scatter plot, histogram (or a combination!). These functions all start with `geom_` as a prefix.

3. Add design elements to the plot such as axis labels, title, fonts, sizes, color schemes, legends, or axes rotation

In code this amounts to the basic template:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

We can further expand this template to include aspects of the visualization such as theme and labels:

```
# plot data from my_data columns as red points
ggplot(data = my_data)+                    # use the dataset "my_data"
  geom_point(                              # add a layer of points (dots)
    mapping = aes(x = col1, y = col2),     # "map" data column to axes
    color = "red")+                        # other specification for the geom
  labs()+                                  # here you add titles, axes labels, etc.
  theme()                                  # here you adjust color, font, size etc of non-d
```

We will explain each component in the sections below.

5.2 `ggplot()`

The opening command of any ggplot2 plot is `ggplot()`. This command simply creates a blank canvas upon which to add layers. It "opens" the way for further layers to be added with a `+` symbol.

Typically, the command `ggplot()` includes the `data =` argument for the plot. This sets the default dataset to be used for subsequent layers of the plot.

This command will end with a `+` after its closing parentheses. This leaves the command "open". The ggplot will only execute/appear when the full command includes a final layer *without* a `+` at the end.

```
# This will create plot that is a blank canvas
ggplot(data = linelist)
```

5.3 Geoms

A blank canvas is certainly not sufficient - we need to create geometries (shapes) from our data (e.g. bar plots, histograms, scatter plots, box plots).

This is done by adding layers "geoms" to the initial `ggplot()` command. There are many **ggplot2** functions that create "geoms". Each of these functions begins with "geom_", so we will refer to them generically as `geom_XXXX()`. There are over 40 geoms in **ggplot2** and many others created by fans. View them at the ggplot2 gallery. Some common geoms are listed below:

- Histograms - `geom_histogram()`
- Bar charts - `geom_bar()` or `geom_col()` (see "Bar plot" section)
- Box plots - `geom_boxplot()`
- Points (e.g. scatter plots) - `geom_point()`
- Line graphs - `geom_line()` or `geom_path()`
- Trend lines - `geom_smooth()`

In one plot you can display one or multiple geoms. Each is added to previous **ggplot2** commands with a `+`, and they are plotted sequentially such that later geoms are plotted on top of previous ones.

5.4 Mapping data to the plot

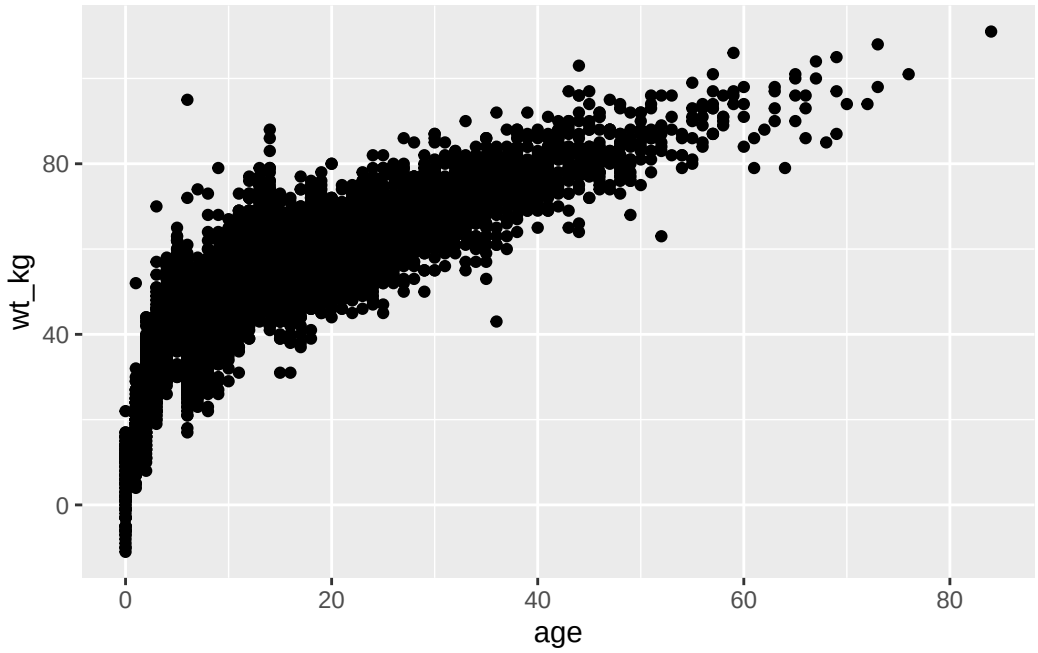
Most geom functions must be told *what to use* to create their shapes - so you must tell them how they should *map (assign) columns in your data* to components of the plot like the axes, shape colors, shape sizes, etc. For most geoms, the *essential* components that must be mapped to columns in the data are the x-axis, and (if necessary) the y-axis.

This "mapping" occurs with the `mapping =` argument. The mappings you provide to `mapping` must be wrapped in the `aes()` function, so you would write something like `mapping = aes(x = col1, y = col2)`, as shown below.

Below, in the `ggplot()` command the data are set as the case `linelist`. In the `mapping = aes()` argument the column `age` is mapped to the x-axis, and the column `wt_kg` is mapped to the y-axis.

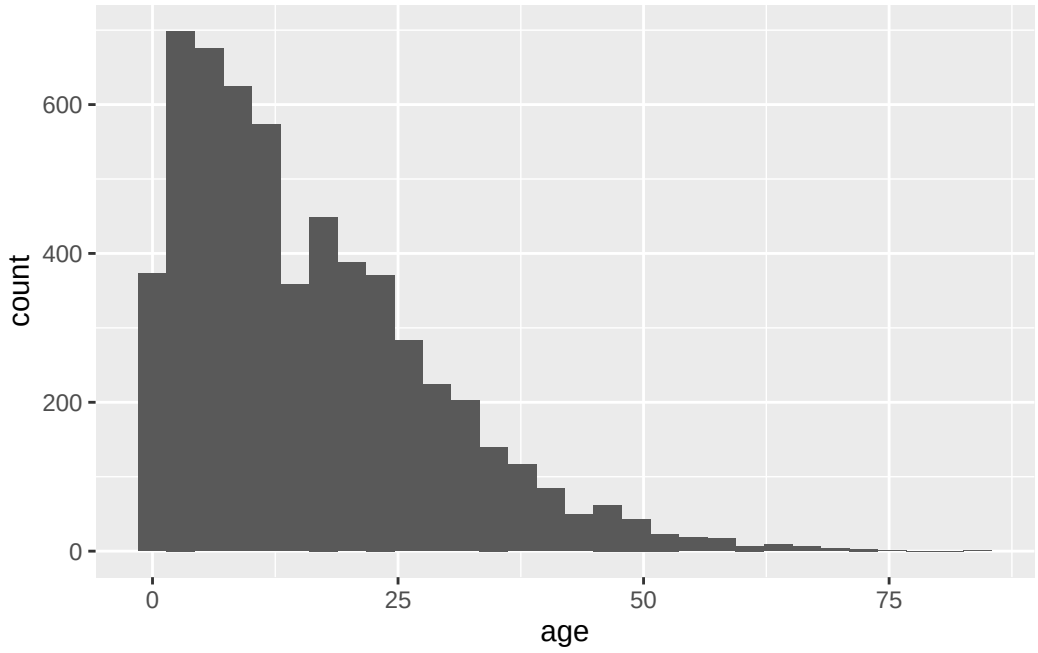
After a `+`, the plotting commands continue. A shape is created with the "geom" function `geom_point()`. This geom *inherits* the mappings from the `ggplot()` command above - it knows the axis-column assignments and proceeds to visualize those relationships as *points* on the canvas.

```
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point()
```



As another example, the following commands utilize the same data, a slightly different mapping, and a different geom. The `geom_histogram()` function only requires a column mapped to the x-axis, as the counts y-axis is generated automatically.

```
ggplot(data = linelist, mapping = aes(x = age))+
  geom_histogram()
```



Plot aesthetics

In ggplot terminology a plot "aesthetic" has a specific meaning. It refers to a visual property of *plotted data*. Note that "aesthetic" here refers to the *data being plotted in geoms/shapes* - not the surrounding display such as titles, axis labels, background color, that you might associate with the word "aesthetics" in common English. In ggplot those details are called "themes" and are adjusted within a `theme()` command (see this section).

Therefore, plot object *aesthetics* can be colors, sizes, transparencies, placement, etc. *of the plotted data*. Not all geoms will have the same aesthetic options, but many can be used by most geoms. Here are some examples:

- `shape =` Display a point with `geom_point()` as a dot, star, triangle, or square…

- `fill =` The interior color (e.g. of a bar or boxplot)
- `color =` The exterior line of a bar, boxplot, etc., or the point color if using `geom_point()`
- `size =` Size (e.g. line thickness, point size)
- `alpha =` Transparency (1 = opaque, 0 = invisible)
- `binwidth =` Width of histogram bins
- `width =` Width of "bar plot" columns
- `linetype =` Line type (e.g. solid, dashed, dotted)

These plot object aesthetics can be assigned values in two ways:

1) Assigned a static value (e.g. `color = "blue"`) to apply across all plotted observations
2) Assigned to a column of the data (e.g. `color = hospital`) such that display of each observation depends on its value in that column

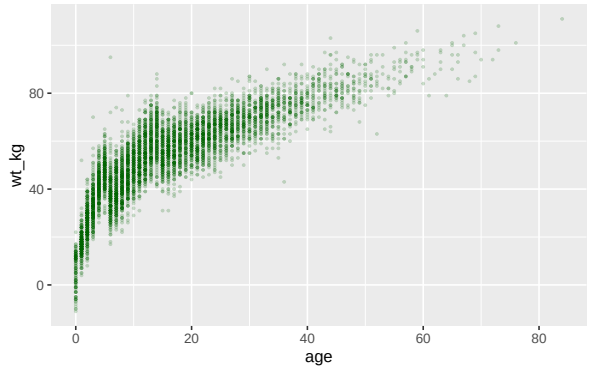
Set to a static value

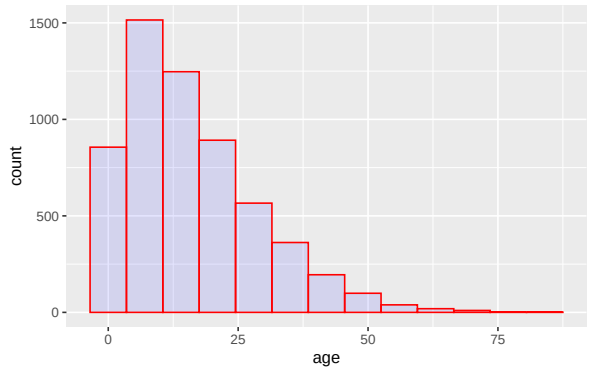
If you want the plot object aesthetic to be static, that is - to be the same for every observation in the data, you write its assignment within the geom but *outside* of any `mapping = aes()` statement. These assignments could look like `size = 1` or `color = "blue"`. Here are two examples:

- In the first example, the `mapping = aes()` is in the `ggplot()` command and the axes are mapped to age and weight columns in the data. The plot aesthetics `color =`, `size =`, and `alpha =` (transparency) are assigned to static values. For clarity, this is done in the `geom_point()` function, as you may add other geoms afterward that would take different values for their plot aesthetics.
- In the second example, the histogram requires only the x-axis mapped to a column. The histogram `binwidth =`, `color =`, `fill =` (internal color), and `alpha =` are again set within the geom to static values.

```
# scatterplot
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+  # set data and axes mapping
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)         # set static point aest
```

```r
# histogram
ggplot(data = linelist, mapping = aes(x = age))+       # set data and axes
  geom_histogram(                # display histogram
    binwidth = 7,                  # width of bins
    color = "red",                 # bin line color
    fill = "blue",                 # bin interior color
    alpha = 0.1)                   # bin transparency
```





Scaled to column values

The alternative is to scale the plot object aesthetic by the values in a column. In this approach, the display of this aesthetic will depend on that observation's value in that column of the data. If the column values are continuous, the display scale (legend) for that aesthetic will be continuous. If the column values are discrete, the legend will display each value and the plotted data will appear as distinctly "grouped" (read more in the grouping section of this page).

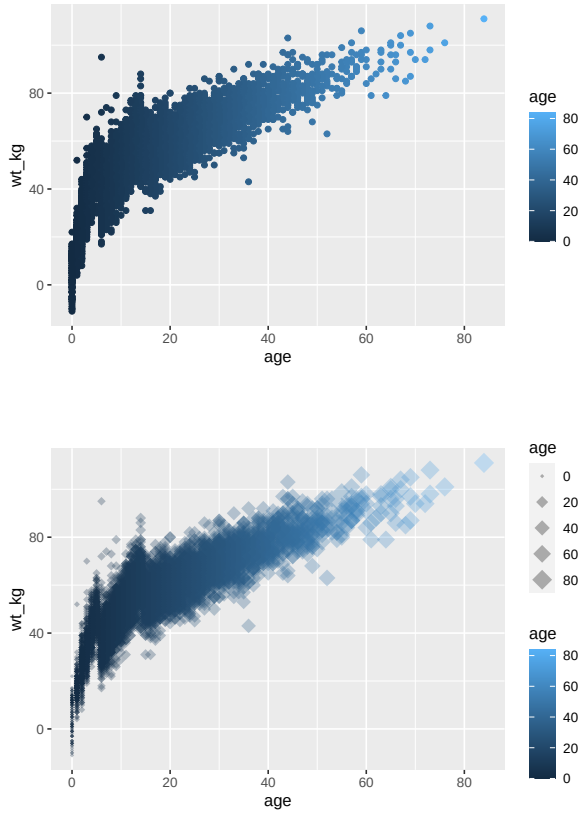
To achieve this, you map that plot aesthetic to a *column name* (not in quotes). This must be done *within a* ***`mapping = aes()`*** *function* (note: there are several places in the code you can make these mapping assignments, as discussed below).

Two examples are below.

- In the first example, the `color =` aesthetic (of each point) is mapped to the column `age` - and a scale has appeared in a legend! For now just note that the scale exists - we will show how to modify it in later sections.

- In the second example two new plot aesthetics are also mapped to columns (`color =` and `size =`), while the plot aesthetics `shape =` and `alpha =` are mapped to static values outside of any `mapping = aes()` function.

```
# scatterplot
ggplot(data = linelist,   # set data
       mapping = aes(     # map aesthetics to column values
         x = age,           # map x-axis to age
         y = wt_kg,         # map y-axis to weight
         color = age)
       )+     # map color to age
  geom_point()         # display data as points

# scatterplot
ggplot(data = linelist,   # set data
       mapping = aes(     # map aesthetics to column values
         x = age,           # map x-axis to age
         y = wt_kg,         # map y-axis to weight
         color = age,       # map color to age
         size = age))+      # map size to age
  geom_point(             # display data as points
    shape = "diamond",      # points display as diamonds
    alpha = 0.3)            # point transparency at 30%
```

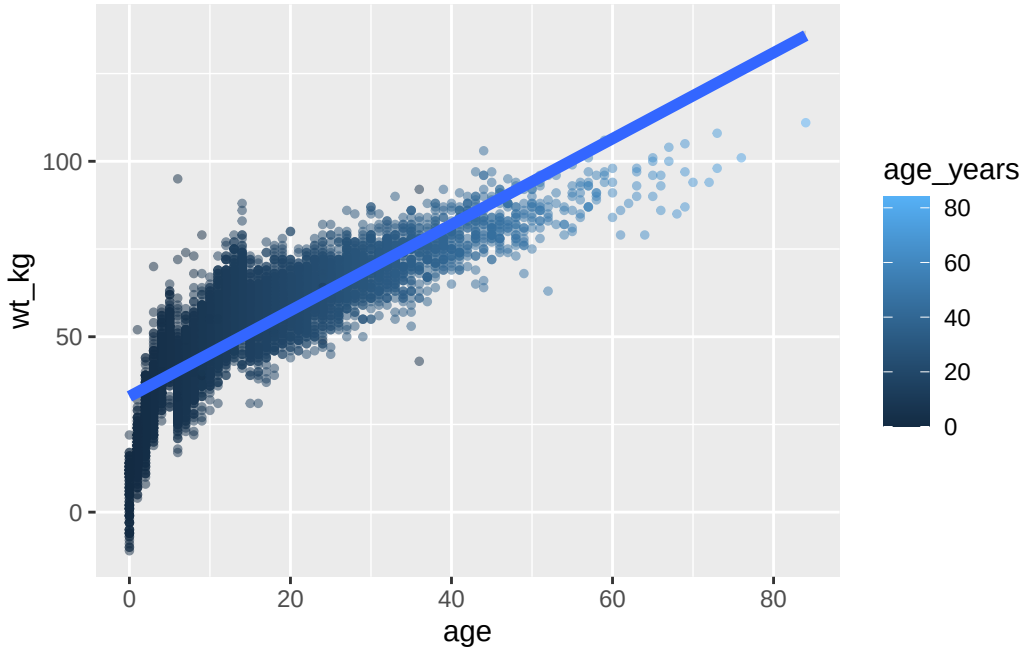


> **Note**
>
> Axes assignments are always assigned to columns in the data (not to static values), and this is always done within `mapping = aes()`.

It becomes important to keep track of your plot layers and aesthetics when making more complex plots - for example plots with multiple geoms. In the example below, the `size =` aesthetic is assigned twice - once for `geom_point()` and once for `geom_smooth()` - both times as a static value.

```
ggplot(data = linelist,
       mapping = aes(           # map aesthetics to columns
         x = age,
         y = wt_kg,
         color = age_years)
       ) +
  geom_point(                   # add points for each row of data
    size = 1,
```

```
  alpha = 0.5) +
geom_smooth(                    # add a trend line
  method = "lm",                # with linear method
  size = 2)                     # size (width of line) of 2
```



Where to make mapping assignments

Aesthetic mapping within `mapping = aes()` can be written in several places in your plotting commands and can even be written more than once. This can be written in the top `ggplot()` command, and/or for each individual geom beneath. The nuances include:

- Mapping assignments made in the top `ggplot()` command will be inherited as defaults across any geom below, like how `x =` and `y =` are inherited
- Mapping assignments made within one geom apply only to that geom

Likewise, `data =` specified in the top `ggplot()` will apply by default to any geom below, but you could also specify data for each geom (but this is more difficult).

Thus, each of the following commands will create the same plot:

```
# These commands will produce the exact same plot
ggplot(data = linelist, mapping = aes(x = age))+
```

```
  geom_histogram()

ggplot(data = linelist)+
  geom_histogram(mapping = aes(x = age))

ggplot()+
  geom_histogram(data = linelist, mapping = aes(x = age))
```

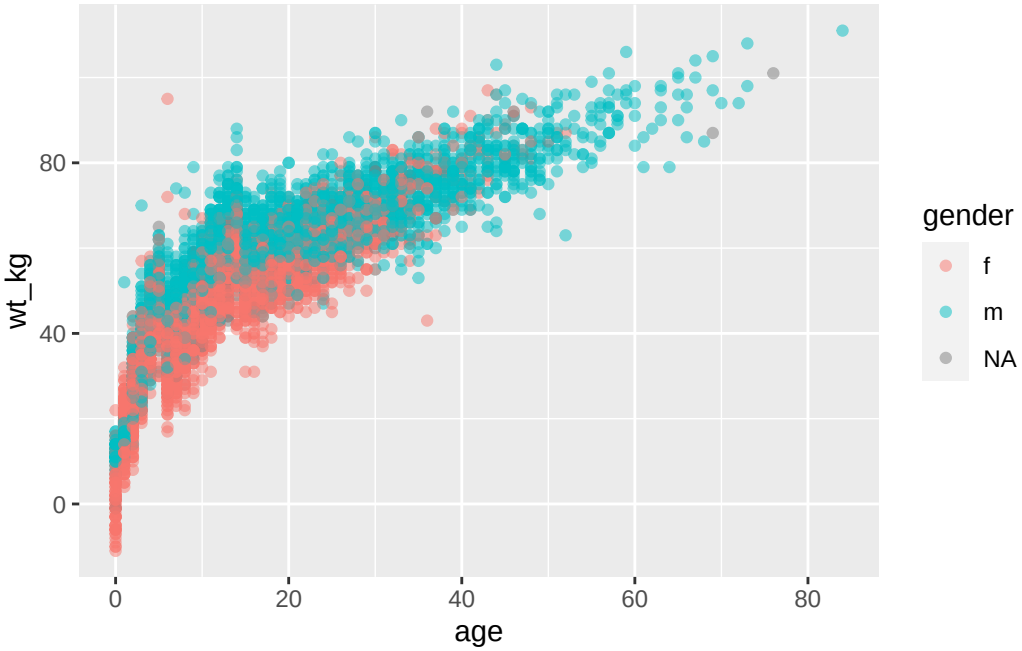
Groups

You can easily group the data and "plot by group". In fact, you have already done this!

Assign the "grouping" column to the appropriate plot aesthetic, within a `mapping = aes()`. Above, we demonstrated this using continuous values when we assigned point `size =` to the column `age`. However this works the same way for discrete/categorical columns.

For example, if you want points to be displayed by gender, you would set `mapping = aes(color = gender)`. A legend automatically appears. This assignment can be made within the `mapping = aes()` in the top `ggplot()` command (and be inherited by the geom), or it could be set in a separate `mapping = aes()` within the geom. Both approaches are shown below:

```
ggplot(data = linelist,
       mapping = aes(x = age, y = wt_kg, color = gender))+
  geom_point(alpha = 0.5)
```




```
# This alternative code produces the same plot
ggplot(data = linelist,
       mapping = aes(x = age, y = wt_kg))+
  geom_point(
    mapping = aes(color = gender),
    alpha = 0.5)
```

Note that depending on the geom, you will need to use different arguments to group the data. For `geom_point()` you will most likely use `color =`, `shape =` or `size =`. Whereas for `geom_bar()` you are more likely to use `fill =`. This just depends on the geom and what plot aesthetic you want to reflect the groupings.

For your information - the most basic way of grouping the data is by using only the `group =` argument within `mapping = aes()`. However, this by itself will not change the colors, fill, or shapes. Nor will it create a legend. Yet the data are grouped, so statistical displays may be affected.

To adjust the order of groups in a plot, see the [ggplot tips] page or the page on [Factors]. There are many examples of grouped plots in the sections below on plotting continuous and categorical data.

5.5 Facets / Small-multiples

Facets, or "small-multiples", are used to split one plot into a multi-panel figure, with one panel ("facet") per group of data. The same type of plot is created multiple times, each one using a sub-group of the same dataset.

Faceting is a functionality that comes with **ggplot2**, so the legends and axes of the facet "panels" are automatically aligned. There are other packages discussed in the [ggplot tips] page that are used to combine completely different plots (**cowplot** and **patchwork**) into one figure.

Faceting is done with one of the following **ggplot2** functions:

1. `facet_wrap()` To show a different panel for each level of a *single* variable. One example of this could be showing a different epidemic curve for each hospital in a region. Facets are ordered alphabetically, unless the variable is a factor with other ordering defined.

- You can invoke certain options to determine the layout of the facets, e.g. `nrow = 1` or `ncol = 1` to control the number of rows or columns that the faceted plots are arranged within.

2. `facet_grid()` This is used when you want to bring a second variable into the faceting arrangement. Here each panel of a grid shows the intersection between values in *two columns*. For example, epidemic curves for each hospital-age group combination with hospitals along the top (columns) and age groups along the sides (rows).

- `nrow` and `ncol` are not relevant, as the subgroups are presented in a grid

Each of these functions accept a formula syntax to specify the column(s) for faceting. Both accept up to two columns, one on each side of a tilde `~`.

- For `facet_wrap()` most often you will write only one column preceded by a tilde `~` like `facet_wrap(~hospital)`. However you can write two columns `facet_wrap(outcome ~ hospital)` - each unique combination will display in a separate panel, but they will not be arranged in a grid. The headings will show combined terms and these won't be specific logic to the columns vs. rows. If you are providing only one faceting variable, a period `.` is used as a placeholder on the other side of the formula - see the code examples.
- For `facet_grid()` you can also specify one or two columns to the formula (grid `rows ~ columns`). If you only want to specify one, you can place a period `.` on the other side of the tilde like `facet_grid(. ~ hospital)` or `facet_grid(hospital ~ .)`.

Facets can quickly contain an overwhelming amount of information - its good to ensure you don't have too many levels of each variable that you choose to facet by. Here are some quick examples with the malaria dataset which consists of daily case counts of malaria for facilities, by age group.

Below we import and do some quick modifications for simplicity:

```
# These data are daily counts of malaria cases, by facility-day
malaria_data <- import(here("data", "malaria_facility_count_data.rds")) %>%  # import
  select(-submitted_date, -Province, -newid)                                 # remove unne
```

The first 50 rows of the malaria data are below. Note there is a column `malaria_tot`, but also columns for counts by age group (these will be used in the second, `facet_grid()` example).

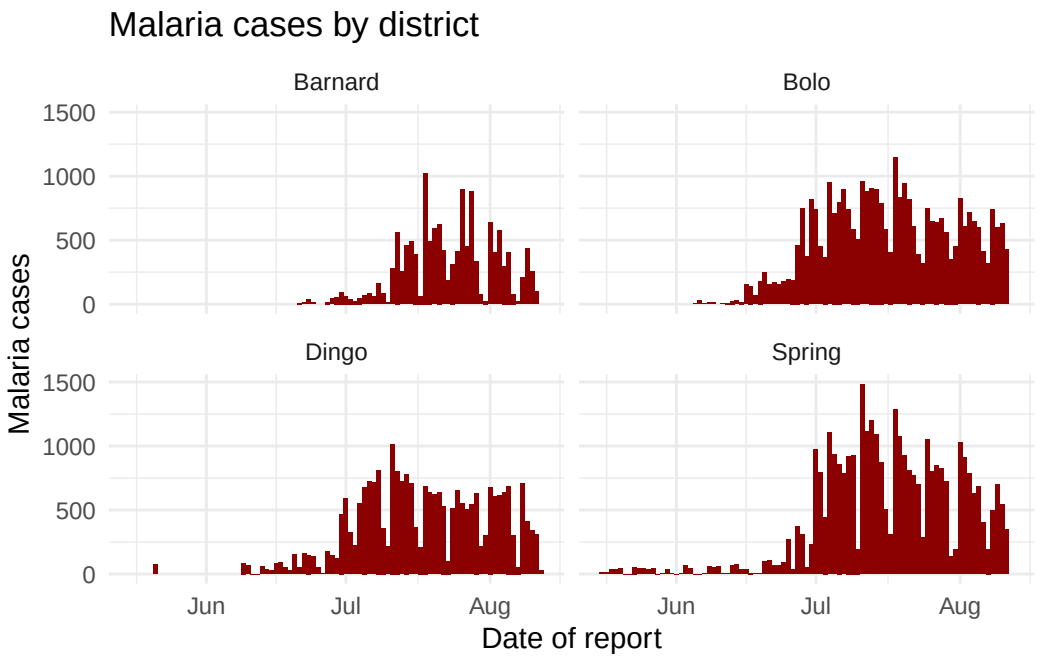
```
# A tibble: 50 x 7
   location_name data_date  District `malaria_rdt_0-4` `malaria_rdt_5-14`
   <chr>         <date>     <chr>                <int>               <int>
 1 Facility 1    2020-08-11 Spring                  11                  12
 2 Facility 2    2020-08-11 Bolo                    11                  10
 3 Facility 3    2020-08-11 Dingo                    8                   5
 4 Facility 4    2020-08-11 Bolo                    16                  16
 5 Facility 5    2020-08-11 Bolo                     9                   2
 6 Facility 6    2020-08-11 Dingo                    3                   1
 7 Facility 6    2020-08-10 Dingo                    4                   0
 8 Facility 5    2020-08-10 Bolo                    15                  14
 9 Facility 5    2020-08-09 Bolo                    11                  11
10 Facility 5    2020-08-08 Bolo                    19                  15
# i 40 more rows
# i 2 more variables: malaria_rdt_15 <int>, malaria_tot <int>
```

`facet_wrap()`

For the moment, let's focus on the columns `malaria_tot` and `District`. Ignore the age-specific count columns for now. We will plot epidemic curves with `geom_col()`, which produces a column for each day at the specified y-axis height given in column `malaria_tot` (the data are already daily counts, so we use `geom_col()` - see the "Bar plot" section below).

When we add the command `facet_wrap()`, we specify a tilde and then the column to facet on (`District` in this case). You can place another column on the left side of the tilde, - this will create one facet for each combination - but we recommend you do this with `facet_grid()` instead. In this use case, one facet is created for each unique value of `District`.

```
# A plot with facets by district
ggplot(malaria_data, aes(x = data_date, y = malaria_tot)) +
  geom_col(width = 1, fill = "darkred") +       # plot the count data as columns
  theme_minimal()+                              # simplify the background panels
  labs(                                         # add plot labels, title, etc.
    x = "Date of report",
    y = "Malaria cases",
    title = "Malaria cases by district") +
  facet_wrap(~District)                        # the facets are created
```



`facet_grid()`

We can use a `facet_grid()` approach to cross two variables. Let's say we want to cross `District` and age. Well, we need to do some data transformations on the age columns to get these data into ggplot-preferred "long" format. The age groups all have their own columns - we want them in a single column called `age_group` and another called `num_cases`. See the page on [Pivoting data] for more information on this process.

```
malaria_age <- malaria_data %>%
  select(-malaria_tot) %>%
  pivot_longer(
```

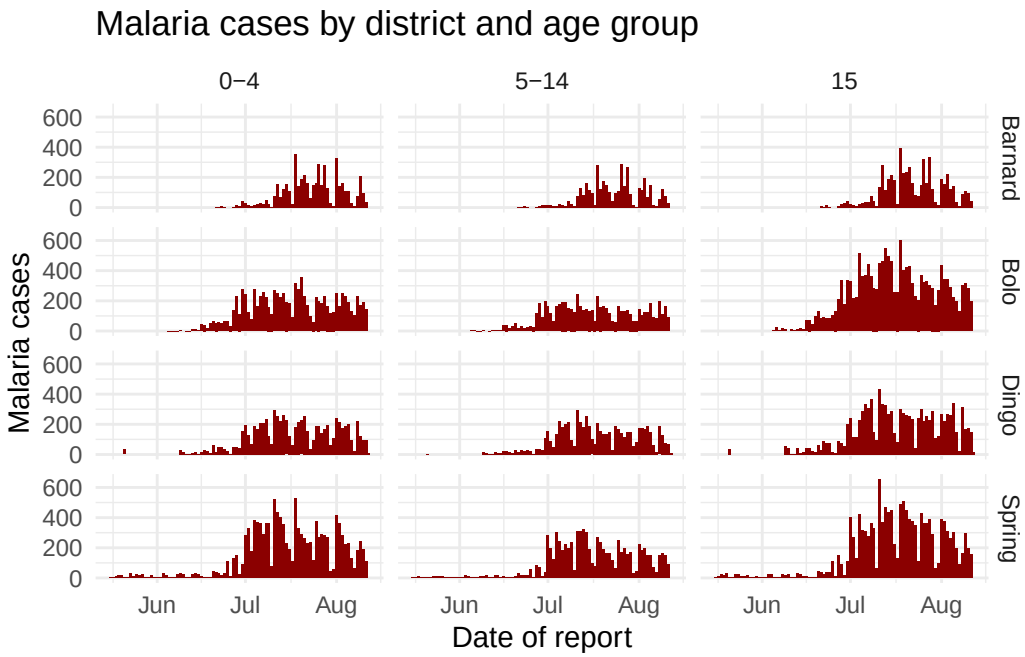
```
    cols = c(starts_with("malaria_rdt_")),  # choose columns to pivot longer
    names_to = "age_group",      # column names become age group
    values_to = "num_cases"      # values to a single column (num_cases)
  ) %>%
  mutate(
    age_group = str_replace(age_group, "malaria_rdt_", ""),
    age_group = forcats::fct_relevel(age_group, "5-14", after = 1))
```

Now the first 50 rows of data look like this:

```
# A tibble: 50 x 5
   location_name data_date  District age_group num_cases
   <chr>         <date>     <chr>    <fct>         <int>
 1 Facility 1    2020-08-11 Spring   0-4              11
 2 Facility 1    2020-08-11 Spring   5-14             12
 3 Facility 1    2020-08-11 Spring   15               23
 4 Facility 2    2020-08-11 Bolo     0-4              11
 5 Facility 2    2020-08-11 Bolo     5-14             10
 6 Facility 2    2020-08-11 Bolo     15                5
 7 Facility 3    2020-08-11 Dingo    0-4               8
 8 Facility 3    2020-08-11 Dingo    5-14              5
 9 Facility 3    2020-08-11 Dingo    15                5
10 Facility 4    2020-08-11 Bolo     0-4              16
# i 40 more rows
```

When you pass the two variables to `facet_grid()`, easiest is to use formula notation (e.g. `x ~ y`) where x is rows and y is columns. Here is the plot, using `facet_grid()` to show the plots for each combination of the columns `age_group` and `District`.

```
ggplot(malaria_age, aes(x = data_date, y = num_cases)) +
  geom_col(fill = "darkred", width = 1) +
  theme_minimal()+
  labs(
    x = "Date of report",
    y = "Malaria cases",
    title = "Malaria cases by district and age group"
  ) +
  facet_grid(District ~ age_group)
```



Free or fixed axes

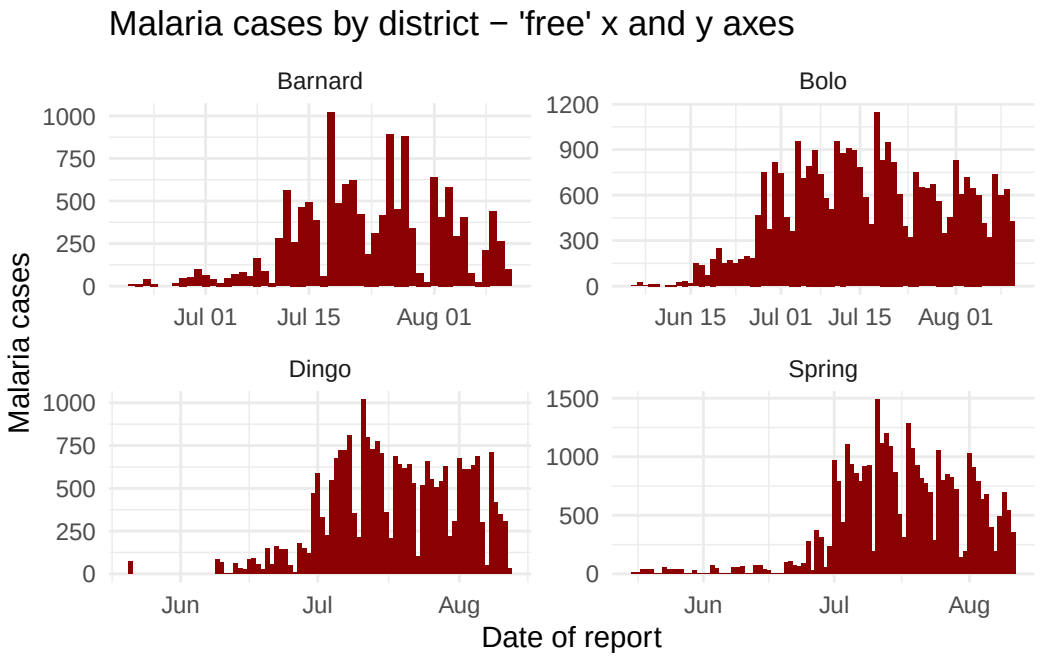
The axes scales displayed when faceting are by default the same (fixed) across all the facets. This is helpful for cross-comparison, but not always appropriate.

When using `facet_wrap()` or `facet_grid()`, we can add `scales = "free_y"` to "free" or release the y-axes of the panels to scale appropriately to their data subset. This is particularly useful if the actual counts are small for one of the subcategories and trends are otherwise hard to see. Instead of "free_y" we can also write "free_x" to do the same for the x-axis (e.g. for dates) or "free" for both axes. Note that in `facet_grid`, the y scales will be the same for facets in the same row, and the x scales will be the same for facets in the same column.

When using `facet_grid` only, we can add `space = "free_y"` or `space = "free_x"` so that the actual height or width of the facet is weighted to the values of the figure within. This only works if `scales = "free"` (y or x) is already applied.

```
# Free y-axis
ggplot(malaria_data, aes(x = data_date, y = malaria_tot)) +
  geom_col(width = 1, fill = "darkred") +       # plot the count data as columns
  theme_minimal()+                              # simplify the background panels
  labs(                                         # add plot labels, title, etc.
    x = "Date of report",
```

```
    y = "Malaria cases",
    title = "Malaria cases by district - 'free' x and y axes") +
  facet_wrap(~District, scales = "free")        # the facets are created
```



5.6 Exporting plots

Exporting ggplots is made easy with the `ggsave()` function from **ggplot2**. It can work in two ways, either:

- Specify the name of the plot object, then the file path and name with extension
  - For example: `ggsave(my_plot, here("plots", "my_plot.png"))`

- Run the command with only a file path, to save the last plot that was printed
  - For example: `ggsave(here("plots", "my_plot.png"))`

You can export as png, pdf, jpeg, tiff, bmp, svg, or several other file types, by specifying the file extension in the file path.

You can also specify the arguments `width =`, `height =`, and `units =` (either "in", "cm", or "mm"). You can also specify `dpi =` with a number for plot resolution (e.g. 300). See the function details by entering `?ggsave` or reading the documentation online.

Remember that you can use `here()` syntax to provide the desired file path. See the [Import and export] page for more information.

5.7 Labels

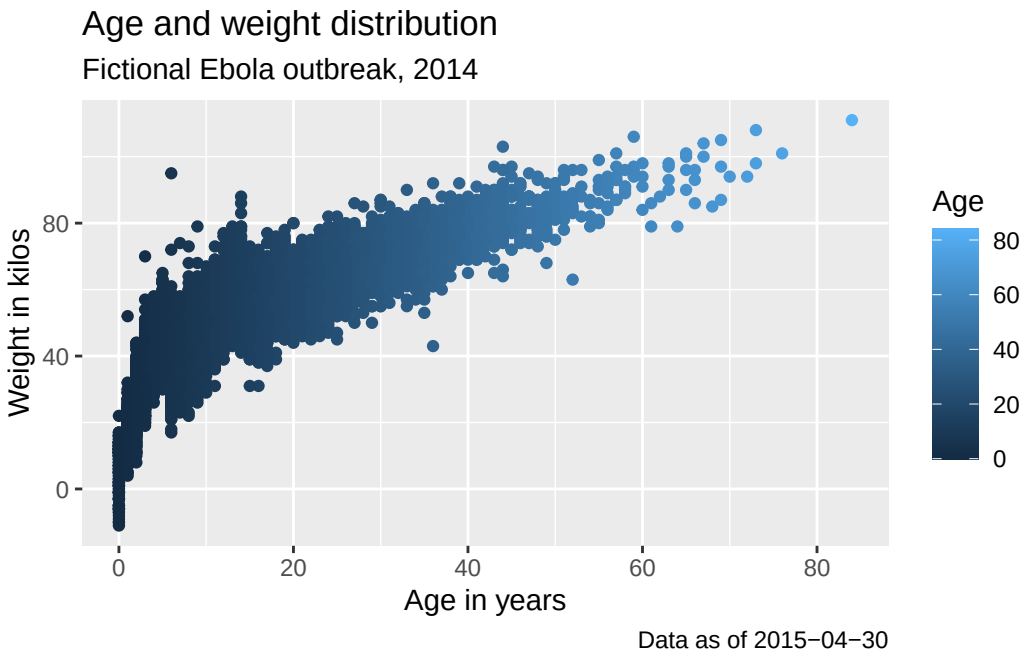
Surely you will want to add or adjust the plot's labels. These are most easily done within the `labs()` function which is added to the plot with `+` just as the geoms were.

Within `labs()` you can provide character strings to these arguements:

- `x =` and `y =` The x-axis and y-axis title (labels)
- `title =` The main plot title
- `subtitle =` The subtitle of the plot, in smaller text below the title
- `caption =` The caption of the plot, in bottom-right by default

Here is a plot we made earlier, but with nicer labels:

```
age_by_wt <- ggplot(
  data = linelist,   # set data
  mapping = aes(     # map aesthetics to column values
         x = age,           # map x-axis to age
         y = wt_kg,         # map y-axis to weight
         color = age))+     # map color to age
  geom_point()+           # display data as points
  labs(
    title = "Age and weight distribution",
    subtitle = "Fictional Ebola outbreak, 2014",
    x = "Age in years",
    y = "Weight in kilos",
    color = "Age",
    caption = stringr::str_glue("Data as of {max(linelist$date_hospitalisation, na.rm=T)}"

age_by_wt
```



Note how in the caption assignment we used `str_glue()` from the **stringr** package to implant dynamic R code within the string text. The caption will show the "Data as of:" date that reflects the maximum hospitalization date in the linelist.

5.8 Plot continuous data

Throughout this page, you have already seen many examples of plotting continuous data. Here we briefly consolidate these and present a few variations.
Visualisations covered here include:

- Plots for one continuous variable:
  - **Histogram**, a classic graph to present the distribution of a continuous variable.
  - **Box plot** (also called box and whisker), to show the 25th, 50th, and 75th percentiles, tail ends of the distribution, and outliers (important limitations).
  - **Jitter plot**, to show all values as points that are 'jittered' so they can (mostly) all be seen, even where two have the same value.
  - **Violin plot**, show the distribution of a continuous variable based on the symmetrical width of the 'violin'.

  - **Sina plot**, are a combination of jitter and violin plots, where individual points are shown but in the symmetrical shape of the distribution (via **ggforce** package).

- **Scatter plot** for two continuous variables.
- **Heat plots** for three continuous variables (linked to [Heat plots] page)

Histograms

Histograms may look like bar charts, but are distinct because they measure the distribution of a *continuous* variable. There are no spaces between the "bars", and only one column is provided to `geom_histogram()`.

Below is code for generating **histograms**, which group continuous data into ranges and display in adjacent bars of varying height. This is done using `geom_histogram()`. See the "Bar plot" section of the ggplot basics page to understand difference between `geom_histogram()`, `geom_bar()`, and `geom_col()`.

We will show the distribution of ages of cases. Within `mapping = aes()` specify which column you want to see the distribution of. You can assign this column to either the x or the y axis.

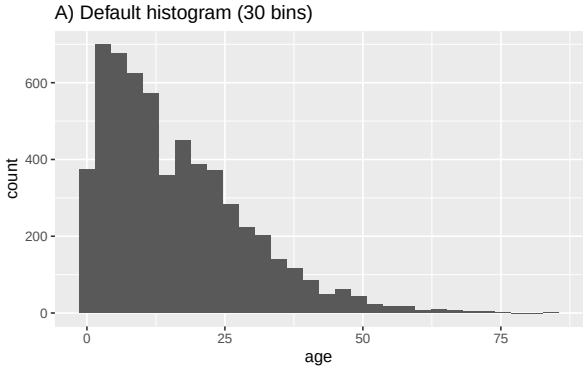
The rows will be assigned to "bins" based on their numeric age, and these bins will be graphically represented by bars. If you specify a number of bins with the `bins =` plot aesthetic, the break points are evenly spaced between the minimum and maximum values of the histogram. If `bins =` is unspecified, an appropriate number of bins will be guessed and this message displayed after the plot:

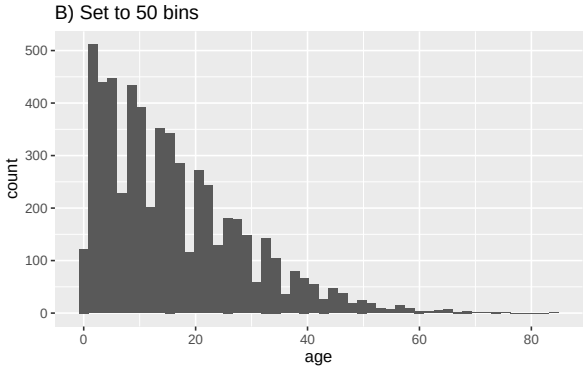
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

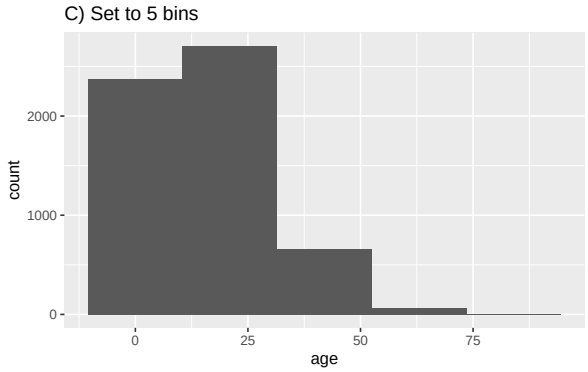
If you do not want to specify a number of bins to `bins =`, you could alternatively specify `binwidth =` in the units of the axis. We give a few examples showing different bins and bin widths:

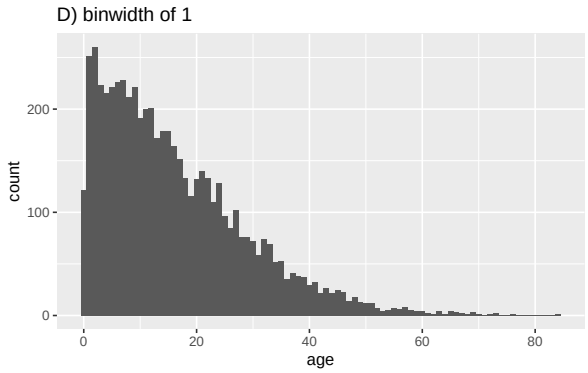
```
# A) Regular histogram
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram()+
  labs(title = "A) Default histogram (30 bins)")

# B) More bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(bins = 50)+
  labs(title = "B) Set to 50 bins")
```

```r
# C) Fewer bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(bins = 5)+
  labs(title = "C) Set to 5 bins")


# D) More bins
ggplot(data = linelist, aes(x = age))+  # provide x variable
  geom_histogram(binwidth = 1)+
  labs(title = "D) binwidth of 1")
```



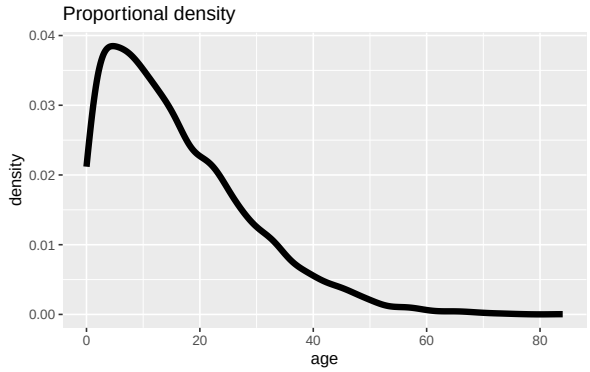


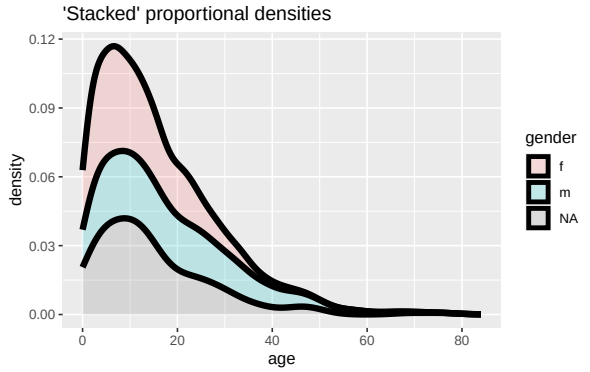




To get smoothed proportions, you can use `geom_density()`:

```
# Frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age)) +
  geom_density(size = 2, alpha = 0.2)+
  labs(title = "Proportional density")

# Stacked frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age, fill = gender)) +
  geom_density(size = 2, alpha = 0.2, position = "stack")+
  labs(title = "'Stacked' proportional densities")
```





To get a "stacked" histogram (of a continuous column of data), you can do one of the following:

1) Use `geom_histogram()` with the `fill =` argument within `aes()` and assigned to the grouping column, or

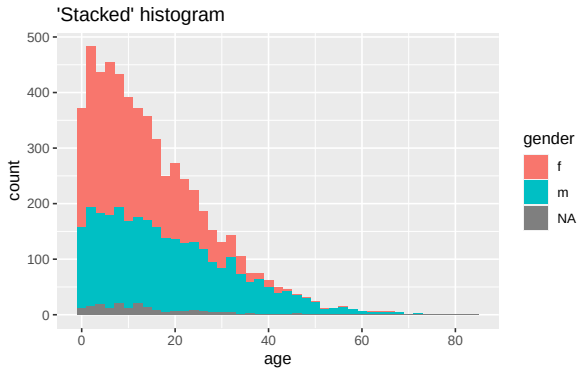
2) Use `geom_freqpoly()`, which is likely easier to read (you can still set `binwidth =`)

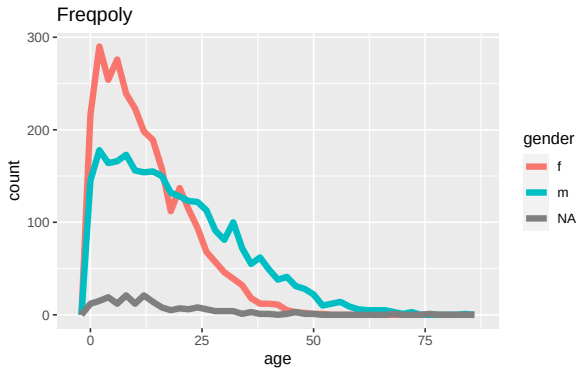
3) To see proportions of all values, set the `y = after_stat(density)` (use this syntax exactly - not changed for your data). Note: these proportions will show *per group*.

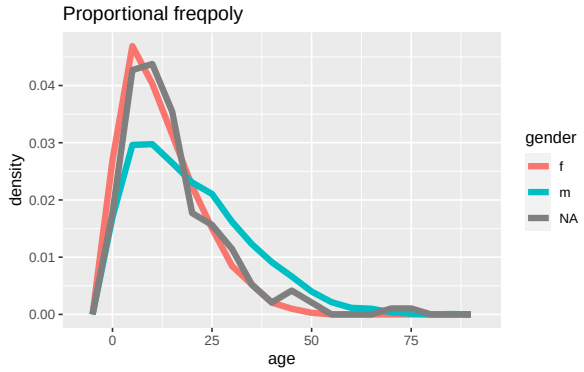
Each is shown below (*note use of `color =` vs. `fill =` in each):

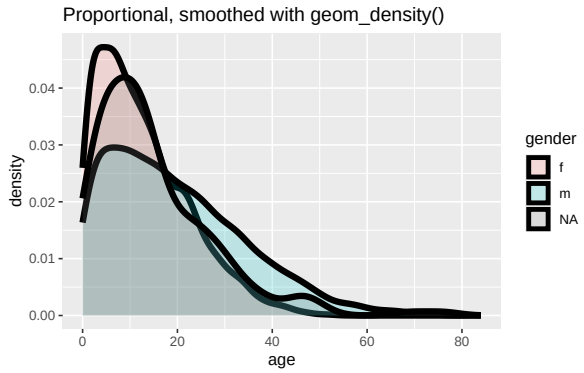
```
# "Stacked" histogram
ggplot(data = linelist, mapping = aes(x = age, fill = gender)) +
  geom_histogram(binwidth = 2)+
  labs(title = "'Stacked' histogram")

# Frequency
```

```r
ggplot(data = linelist, mapping = aes(x = age, color = gender)) +
  geom_freqpoly(binwidth = 2, size = 2)+
  labs(title = "Freqpoly")

# Frequency with proportion axis
ggplot(data = linelist, mapping = aes(x = age, y = after_stat(density), color = gender)) +
  geom_freqpoly(binwidth = 5, size = 2)+
  labs(title = "Proportional freqpoly")

# Frequency with proportion axis, smoothed
ggplot(data = linelist, mapping = aes(x = age, y = after_stat(density), fill = gender)) +
  geom_density(size = 2, alpha = 0.2)+
  labs(title = "Proportional, smoothed with geom_density()")
```









If you want to have some fun, try `geom_density_ridges` from the **ggridges** package (vignette here.

Read more in detail about histograms at the **tidyverse** page on geom_histogram().

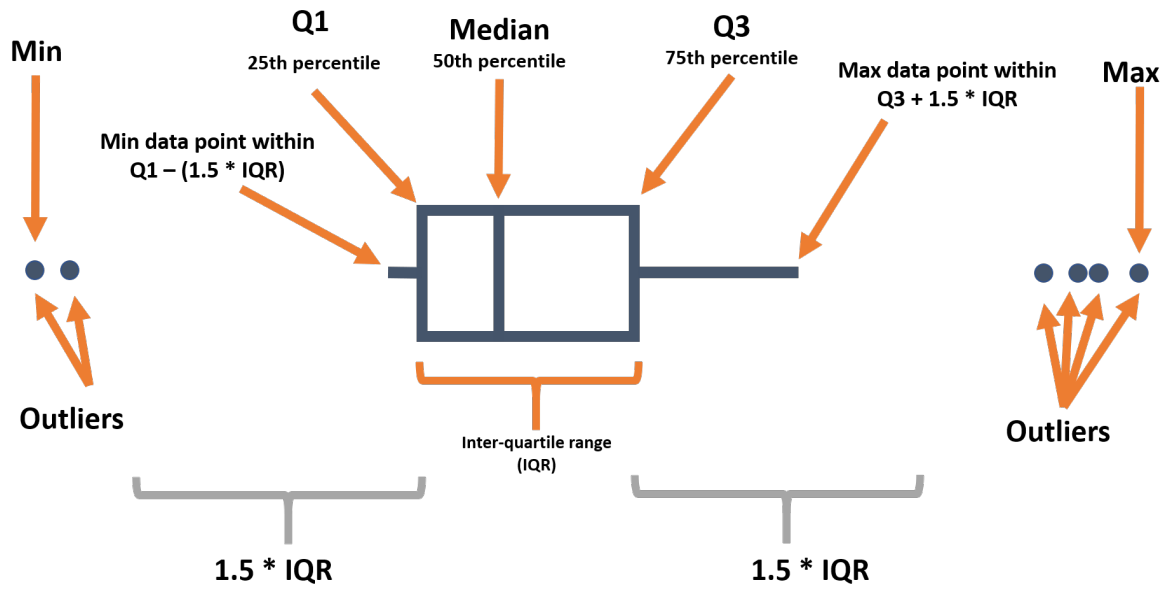
Box plots

Box plots are common, but have important limitations. They can obscure the actual distribution - e.g. a bi-modal distribution. See this R graph gallery and this data-to-viz article for more details. However, they do nicely display the inter-quartile range and outliers - so they can be overlaid on top of other types of plots that show the distribution in more detail.

Below we remind you of the various components of a boxplot:

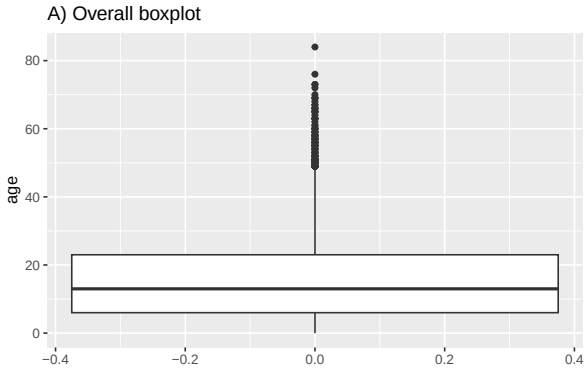
When using `geom_boxplot()` to create a box plot, you generally map only one axis (x or y) within `aes()`. The axis specified determines if the plots are horizontal or vertical.

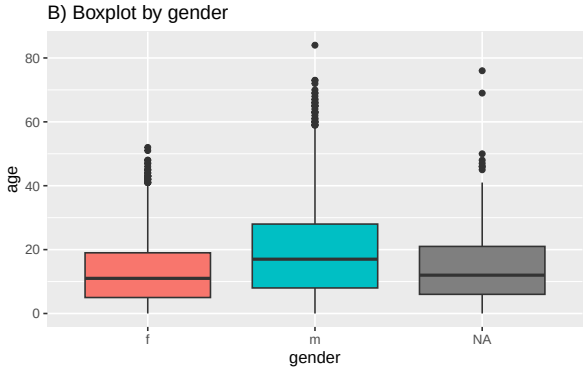
In most geoms, you create a plot per group by mapping an aesthetic like `color =` or `fill =` to a column within `aes()`. However, for box plots achieve this by assigning the grouping column to the un-assigned axis (x or y). Below is code for a boxplot of *all* age values in the dataset,



and second is code to display one box plot for each (non-missing) gender in the dataset. Note that `NA` (missing) values will appear as a separate box plot unless removed. In this example we also set the `fill` to the column `outcome` so each plot is a different color - but this is not necessary.

```
# A) Overall boxplot
ggplot(data = linelist)+
  geom_boxplot(mapping = aes(y = age))+   # only y axis mapped (not x)
  labs(title = "A) Overall boxplot")

# B) Box plot by group
ggplot(data = linelist, mapping = aes(y = age, x = gender, fill = gender)) +
  geom_boxplot()+
  theme(legend.position = "none")+   # remove legend (redundant)
  labs(title = "B) Boxplot by gender")
```





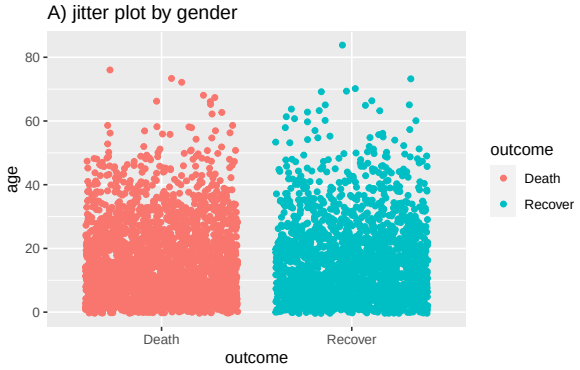
For code to add a box plot to the edges of a scatter plot ("marginal" plots) see the page [ggplot tips].

Violin, jitter, and sina plots

Below is code for creating **violin plots** (`geom_violin`) and **jitter plots** (`geom_jitter`) to show distributions. You can specify that the fill or color is also determined by the data, by inserting these options within `aes()`.

```
# A) Jitter plot by group
ggplot(data = linelist %>% drop_na(outcome),      # remove missing values
       mapping = aes(y = age,                     # Continuous variable
           x = outcome,                           # Grouping variable
           color = outcome))+                     # Color variable
  geom_jitter()+                                  # Create the violin plot
  labs(title = "A) jitter plot by gender")
```

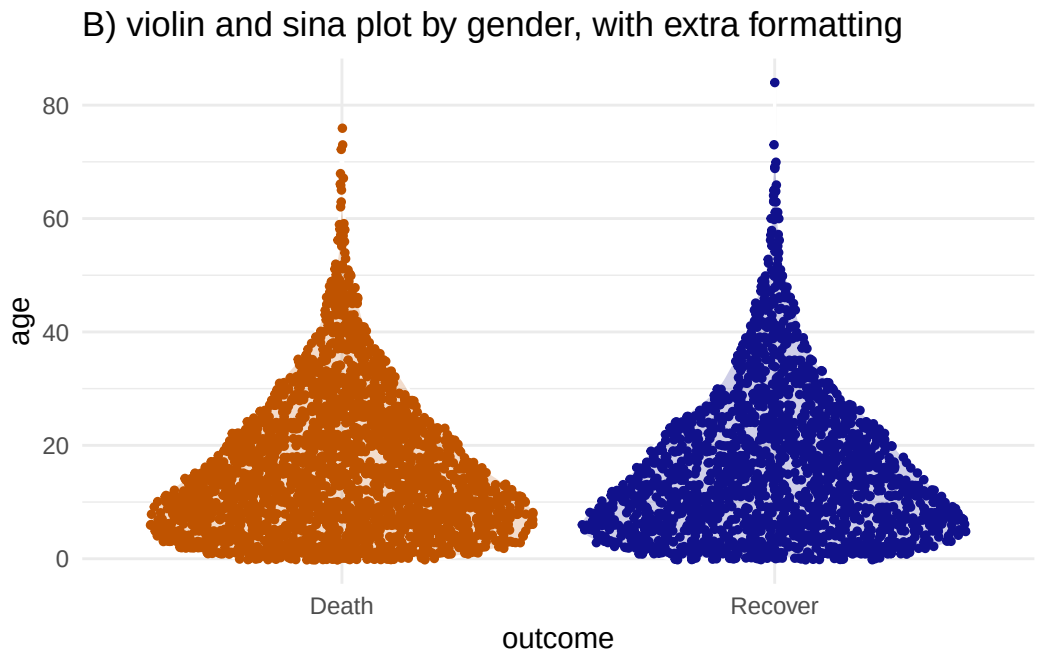
```r
# B) Violin plot by group
ggplot(data = linelist %>% drop_na(outcome),       # remove missing values
       mapping = aes(y = age,                      # Continuous variable
           x = outcome,                            # Grouping variable
           fill = outcome))+                       # fill variable (color)
  geom_violin()+                                   # create the violin plot
  labs(title = "B) violin plot by gender")
```



You can combine the two using the `geom_sina()` function from the **ggforce** package. The sina plots the jitter points in the shape of the violin plot. When overlaid on the violin plot (adjusting the transparencies) this can be easier to visually interpret.

```r
# A) Sina plot by group
ggplot(
  data = linelist %>% drop_na(outcome),
  aes(y = age,            # numeric variable
```

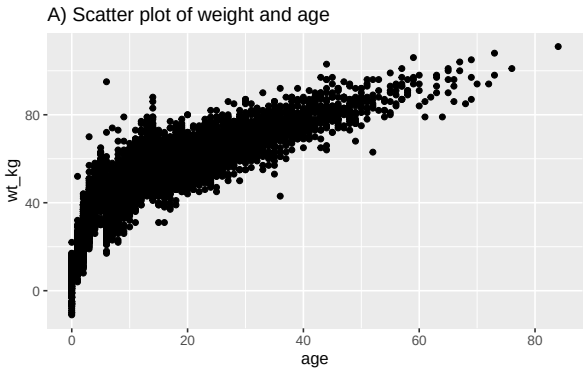
```
    x = outcome)) +    # group variable
geom_violin(
  aes(fill = outcome), # fill (color of violin background)
  color = "white",     # white outline
  alpha = 0.2)+        # transparency
geom_sina(
  size=1,                # Change the size of the jitter
  aes(color = outcome))+ # color (color of dots)
scale_fill_manual(       # Define fill for violin background by death/recover
  values = c("Death" = "#bf5300",
             "Recover" = "#11118c")) +
scale_color_manual(      # Define colours for points by death/recover
  values = c("Death" = "#bf5300",
             "Recover" = "#11118c")) +
theme_minimal() +                                  # Remove the gray background
theme(legend.position = "none") +                  # Remove unnecessary legend
labs(title = "B) violin and sina plot by gender, with extra formatting")
```



Two continuous variables

Following similar syntax, `geom_point()` will allow you to plot two continuous variables against each other in a **scatter plot**. This is useful for showing actual values rather than their distributions. A basic scatter plot of age vs weight is shown in (A). In (B) we again use `facet_grid()` to show the relationship between two continuous variables in the linelist.

```
# Basic scatter plot of weight and age
ggplot(data = linelist,
       mapping = aes(y = wt_kg, x = age))+
  geom_point() +
  labs(title = "A) Scatter plot of weight and age")

# Scatter plot of weight and age by gender and Ebola outcome
ggplot(data = linelist %>% drop_na(gender, outcome), # filter retains non-missing gender/o
       mapping = aes(y = wt_kg, x = age))+
  geom_point() +
  labs(title = "B) Scatter plot of weight and age faceted by gender and outcome")+
  facet_grid(gender ~ outcome)
```



Three continuous variables

You can display three continuous variables by utilizing the `fill =` argument to create a *heat plot*. The color of each "cell" will reflect the value of the third continuous column of data. See the [ggplot tips] page and the page on on [Heat plots] for more details and several examples.

There are ways to make 3D plots in R, but for applied epidemiology these are often difficult to interpret and therefore less useful for decision-making.

5.9 Plot categorical data

Categorical data can be character values, could be logical (TRUE/FALSE), or factors (see the [Factors] page).

Preparation

Data structure

The first thing to understand about your categorical data is whether it exists as raw observations like a linelist of cases, or as a summary or aggregate data frame that holds counts or proportions. The state of your data will impact which plotting function you use:

- If your data are raw observations with one row per observation, you will likely use `geom_bar()`
- If your data are already aggregated into counts or proportions, you will likely use `geom_col()`

Column class and value ordering

Next, examine the class of the columns you want to plot. We look at `hospital`, first with `class()` from **base** R, and with `tabyl()` from **janitor**.

```
# View class of hospital column - we can see it is a character
class(linelist$hospital)
```

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

```
# Look at values and proportions within hospital column
linelist %>%
  tabyl(hospital)
```

```
                          hospital    n    percent
                   Central Hospital  454 0.07710598
                  Military Hospital  896 0.15217391
                           Missing 1469 0.24949049
                             Other  885 0.15030571
                     Port Hospital 1762 0.29925272
 St. Mark's Maternity Hospital (SMMH)  422 0.07167120
```

We can see the values within are characters, as they are hospital names, and by default they are ordered alphabetically. There are 'other' and 'missing' values, which we would prefer to be the last subcategories when presenting breakdowns. So we change this column into a factor and re-order it. This is covered in more detail in the [Factors] page.

```
# Convert to factor and define level order so "Other" and "Missing" are last
linelist <- linelist %>%
  mutate(
    hospital = fct_relevel(hospital,
      "St. Mark's Maternity Hospital (SMMH)",
      "Port Hospital",
      "Central Hospital",
      "Military Hospital",
      "Other",
      "Missing"))
```

```
levels(linelist$hospital)
```

```
[1] "St. Mark's Maternity Hospital (SMMH)"
[2] "Port Hospital"
[3] "Central Hospital"
[4] "Military Hospital"
[5] "Other"
[6] "Missing"
```

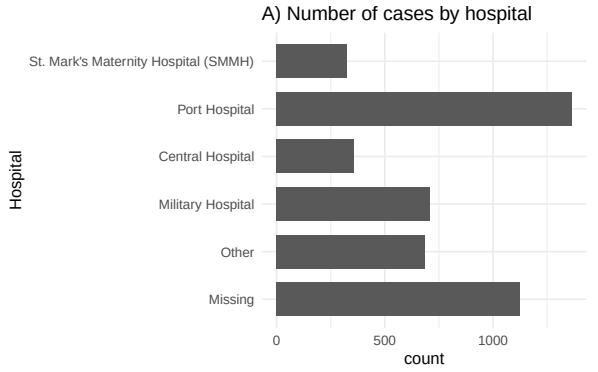
geom_bar()

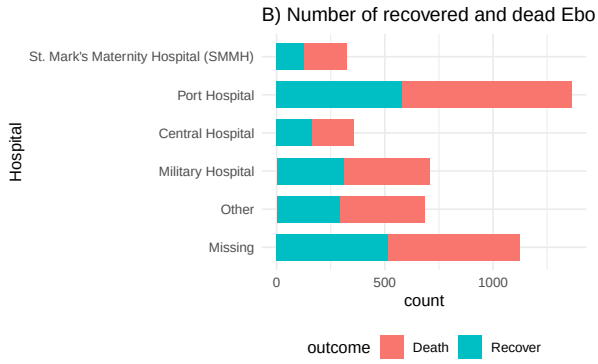
Use `geom_bar()` if you want bar height (or the height of stacked bar components) to reflect *the number of relevant rows in the data*. These bars will have gaps between them, unless the `width =` plot aesthetic is adjusted.

- Provide only one axis column assignment (typically x-axis). If you provide x and y, you will get `Error: stat_count() can only have an x or y aesthetic.`
- You can create stacked bars by adding a `fill =` column assignment within `mapping = aes()`
- The opposite axis will be titled "count" by default, because it represents the number of rows

Below, we have assigned outcome to the y-axis, but it could just as easily be on the x-axis. If you have longer character values, it can sometimes look better to flip the bars sideways and put the legend on the bottom. This may impact how your factor levels are ordered - in this case we reverse them with `fct_rev()` to put missing and other at the bottom.

```
# A) Outcomes in all cases
ggplot(linelist %>% drop_na(outcome)) +
  geom_bar(aes(y = fct_rev(hospital)), width = 0.7) +
  theme_minimal()+
  labs(title = "A) Number of cases by hospital",
       y = "Hospital")


# B) Outcomes in all cases by hosptial
ggplot(linelist %>% drop_na(outcome)) +
  geom_bar(aes(y = fct_rev(hospital), fill = outcome), width = 0.7) +
  theme_minimal()+
  theme(legend.position = "bottom") +
  labs(title = "B) Number of recovered and dead Ebola cases, by hospital",
       y = "Hospital")
```





`geom_col()`

Use `geom_col()` if you want bar height (or height of stacked bar components) to reflect pre-calculated *values* that exists in the data. Often, these are summary or "aggregated" counts, or proportions.

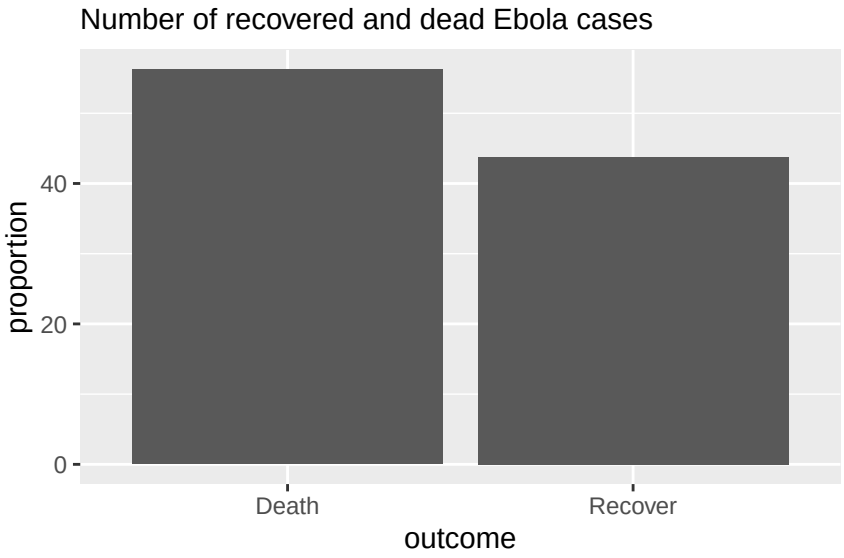
Provide column assignments for *both* axes to `geom_col()`. Typically your x-axis column is discrete and your y-axis column is numeric.

Let's say we have this dataset `outcomes`:

```
# A tibble: 2 x 3
  outcome     n proportion
  <chr>   <int>      <dbl>
1 Death    1022       56.2
2 Recover   796       43.8
```

Below is code using `geom_col` for creating simple bar charts to show the distribution of Ebola patient outcomes. With geom_col, both x and y need to be specified. Here x is the categorical variable along the x axis, and y is the generated proportions column `proportion`.

```
# Outcomes in all cases
ggplot(outcomes) +
  geom_col(aes(x=outcome, y = proportion)) +
  labs(subtitle = "Number of recovered and dead Ebola cases")
```



To show breakdowns by hospital, we would need our table to contain more information, and to be in "long" format. We create this table with the frequencies of the combined categories `outcome` and `hospital` (see [Grouping data] page for grouping tips).

```
outcomes2 <- linelist %>%
  drop_na(outcome) %>%
  count(hospital, outcome) %>%  # get counts by hospital and outcome
  group_by(hospital) %>%        # Group so proportions are out of hospital total
  mutate(proportion = n/sum(n)*100) # calculate proportions of hospital total

head(outcomes2) # Preview data
```

```
# A tibble: 6 x 4
# Groups:   hospital [3]
  hospital                                outcome     n proportion
```

```
  <fct>                                <chr>   <int>      <dbl>
1 St. Mark's Maternity Hospital (SMMH) Death     199       61.2
2 St. Mark's Maternity Hospital (SMMH) Recover   126       38.8
3 Port Hospital                        Death     785       57.6
4 Port Hospital                        Recover   579       42.4
5 Central Hospital                     Death     193       53.9
6 Central Hospital                     Recover   165       46.1
```

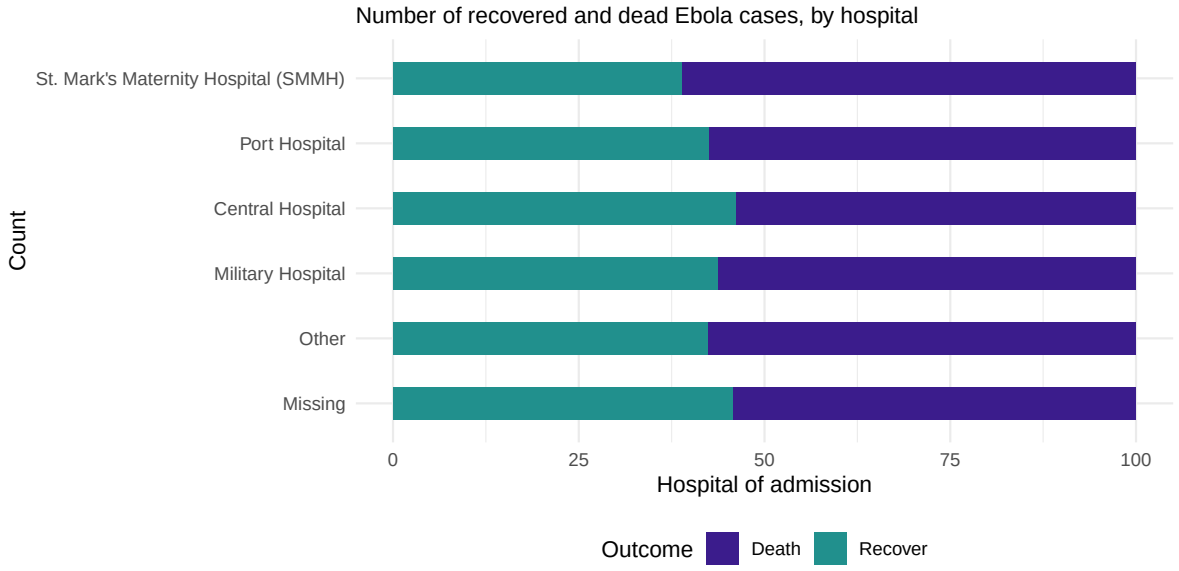
We then create the ggplot with some added formatting:

- **Axis flip**: Swapped the axis around with `coord_flip()` so that we can read the hospital names.
- **Columns side-by-side**: Added a `position = "dodge"` argument so that the bars for death and recover are presented side by side rather than stacked. Note stacked bars are the default.
- **Column width**: Specified 'width', so the columns are half as thin as the full possible width.
- **Column order**: Reversed the order of the categories on the y axis so that 'Other' and 'Missing' are at the bottom, with `scale_x_discrete(limits=rev)`. Note that we used that rather than `scale_y_discrete` because hospital is stated in the `x` argument of `aes()`, even if visually it is on the y axis. We do this because Ggplot seems to present categories backwards unless we tell it not to.

- **Other details**: Labels/titles and colours added within `labs` and `scale_fill_color` respectively.

```r
# Outcomes in all cases by hospital
ggplot(outcomes2) +
  geom_col(
    mapping = aes(
      x = proportion,                 # show pre-calculated proportion values
      y = fct_rev(hospital),          # reverse level order so missing/other at bottom
      fill = outcome),                # stacked by outcome
    width = 0.5)+                    # thinner bars (out of 1)
  theme_minimal() +                  # Minimal theme
  theme(legend.position = "bottom")+
  labs(subtitle = "Number of recovered and dead Ebola cases, by hospital",
       fill = "Outcome",             # legend title
       y = "Count",                  # y axis title
       x = "Hospital of admission")+ # x axis title
  scale_fill_manual(                 # adding colors manually
    values = c("Death"= "#3B1c8C",
```

```
                "Recover" = "#21908D" ))
```



Note that the proportions are binary, so we may prefer to drop 'recover' and just show the proportion who died. This is just for illustration purposes.

If using `geom_col()` with dates data (e.g. an epicurve from aggregated data) - you will want to adjust the `width =` argument to remove the "gap" lines between the bars. If using daily data set `width = 1`. If weekly, `width = 7`. Months are not possible because each month has a different number of days.

5.10 Themes

One of the best parts of **ggplot2** is the amount of control you have over the plot - you can define anything! As mentioned above, the design of the plot that is *not* related to the data shapes/geometries are adjusted within the `theme()` function. For example, the plot background color, presence/absence of gridlines, and the font/size/color/alignment of text (titles, subtitles, captions, axis text…). These adjustments can be done in one of two ways:

- Add a *complete theme* `theme_()` function to make sweeping adjustments - these include `theme_classic()`, `theme_minimal()`, `theme_dark()`, `theme_light()` `theme_grey()`, `theme_bw()` among others

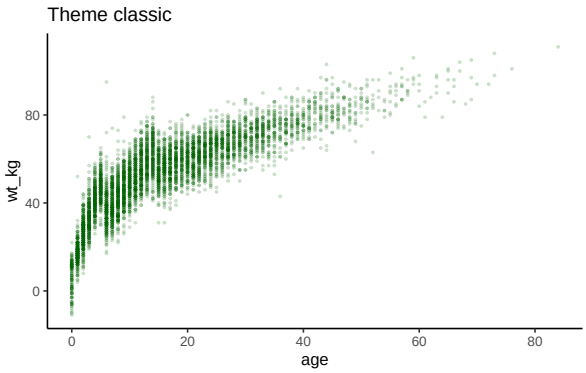
- Adjust each tiny aspect of the plot individually within `theme()`

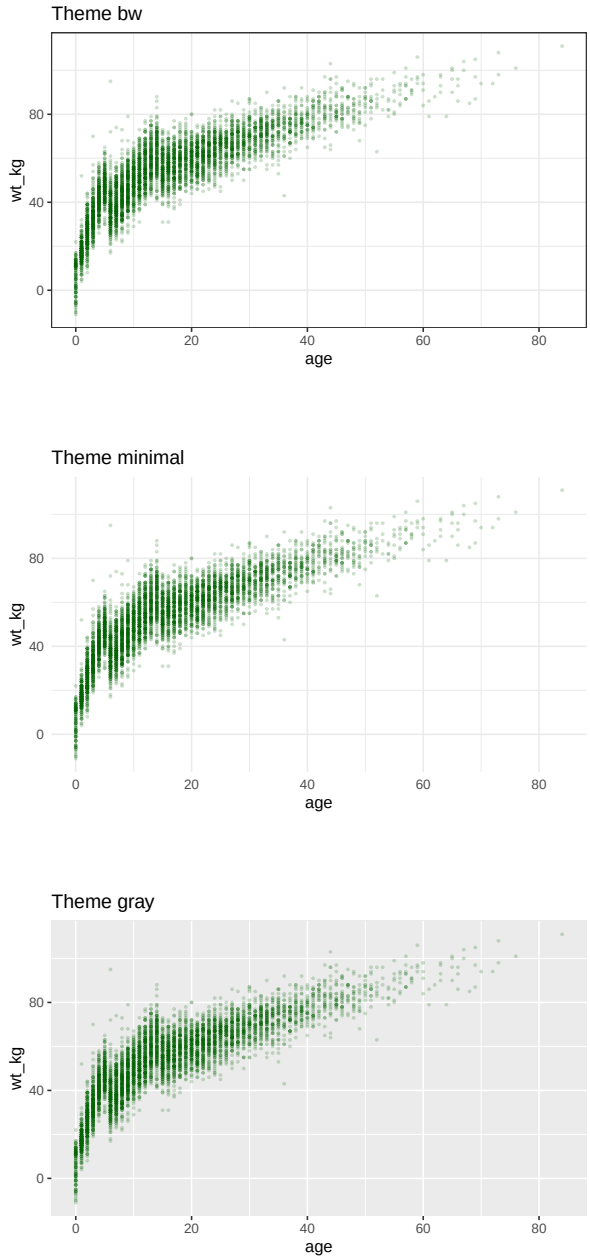
Complete themes

As they are quite straight-forward, we will demonstrate the complete theme functions below and will not describe them further here. Note that any micro-adjustments with `theme()` should be made *after* use of a complete theme.

Write them with empty parentheses.

```
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)+
  labs(title = "Theme classic")+
  theme_classic()

ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)+
  labs(title = "Theme bw")+
  theme_bw()

ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)+
  labs(title = "Theme minimal")+
  theme_minimal()

ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(color = "darkgreen", size = 0.5, alpha = 0.2)+
  labs(title = "Theme gray")+
  theme_gray()
```





Modify theme

The `theme()` function can take a large number of arguments, each of which edits a very specific aspect of the plot. There is no way we could cover all of the arguments, but we will describe the general pattern for them and show you how to find the argument name that you need. The basic syntax is this:

1. Within `theme()` write the argument name for the plot element you want to edit, like `plot.title =`

2. Provide an `element_()` function to the argument

- Most often, use `element_text()`, but others include `element_rect()` for canvas background colors, or `element_blank()` to remove plot elements

4. Within the `element_()` function, write argument assignments to make the fine adjustments you desire

So, that description was quite abstract, so here are some examples.

The below plot looks quite silly, but it serves to show you a variety of the ways you can adjust your plot.

- We begin with the plot `age_by_wt` defined just above and add `theme_classic()`

- For finer adjustments we add `theme()` and include one argument for each plot element to adjust

It can be nice to organize the arguments in logical sections. To describe just some of those used below:

- `legend.position =` is unique in that it accepts simple values like "bottom", "top", "left", and "right". But generally, text-related arguments require that you place the details *within* `element_text()`.

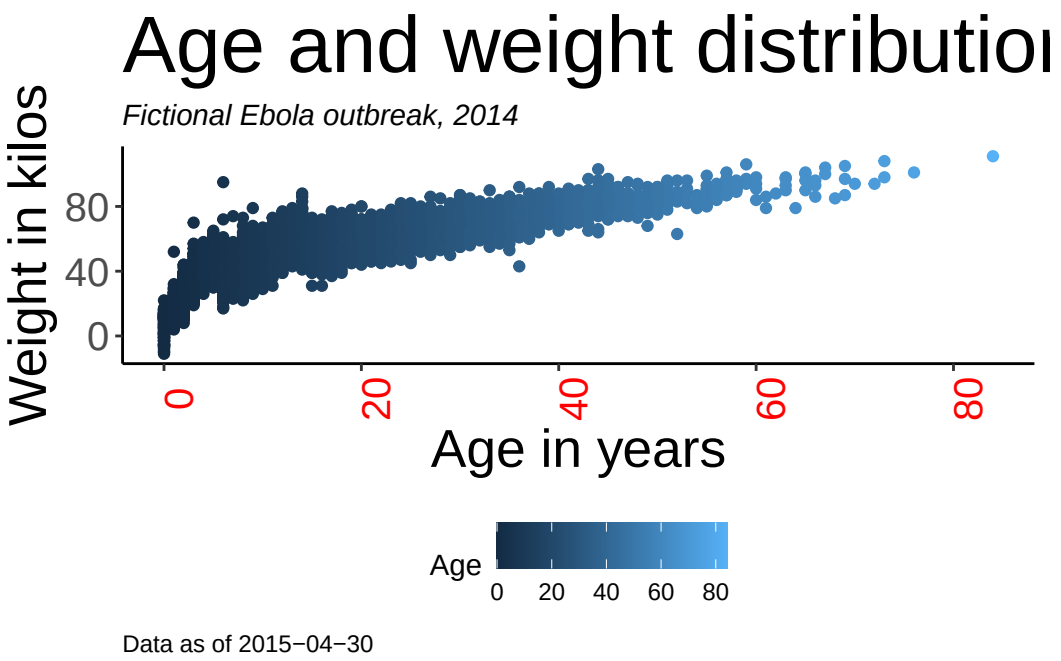
- Title size with `element_text(size = 30)`

- The caption horizontal alignment with `element_text(hjust = 0)` (from right to left)

- The subtitle is italicized with `element_text(face = "italic")`

```
age_by_wt +
  theme_classic()+                                  # pre-defined theme adjustments
  theme(
    legend.position = "bottom",                     # move legend to bottom

    plot.title = element_text(size = 30),           # size of title to 30
    plot.caption = element_text(hjust = 0),         # left-align caption
    plot.subtitle = element_text(face = "italic"), # italicize subtitle
```

```
  axis.text.x = element_text(color = "red", size = 15, angle = 90), # adjusts only x-axi
  axis.text.y = element_text(size = 15),          # adjusts only y-axis text

  axis.title = element_text(size = 20)            # adjusts both axes titles
  )
```



Here are some especially common `theme()` arguments. You will recognize some patterns, such as appending `.x` or `.y` to apply the change only to one axis.

`theme()` argument	What it adjusts
`plot.title = element_text()`	The title
`plot.subtitle = element_text()`	The subtitle
`plot.caption = element_text()`	The caption (family, face, color, size, angle, vjust, hjust…)
`axis.title = element_text()`	Axis titles (both x and y) (size, face, angle, color…)
`axis.title.x = element_text()`	Axis title x-axis only (use `.y` for y-axis only)
`axis.text = element_text()`	Axis text (both x and y)
`axis.text.x = element_text()`	Axis text x-axis only (use `.y` for y-axis only)
`axis.ticks = element_blank()`	Remove axis ticks

`theme()` argument	What it adjusts
`axis.line = element_line()`	Axis lines (colour, size, linetype: solid dashed dotted etc)
`strip.text = element_text()`	Facet strip text (colour, face, size, angle…)
`strip.background = element_rect()`	facet strip (fill, colour, size…)

But there are so many theme arguments! How could I remember them all? Do not worry - it is impossible to remember them all. Luckily there are a few tools to help you:

The **tidyverse** documentation on modifying theme, which has a complete list.

> Tip
>
> Run `theme_get()` from **ggplot2** to print a list of all 90+ `theme()` arguments to the console.

> Tip
>
> If you ever want to remove an element of a plot, you can also do it through `theme()`. Just pass `element_blank()` to an argument to have it disappear completely. For legends, set `legend.position = "none"`.

5.11 Scales for color, fill, axes, etc.

In **ggplot2**, when aesthetics of plotted data (e.g. size, color, shape, fill, plot axis) are mapped to columns in the data, the exact display can be adjusted with the corresponding "scale" command. In this section we explain some common scale adjustments.

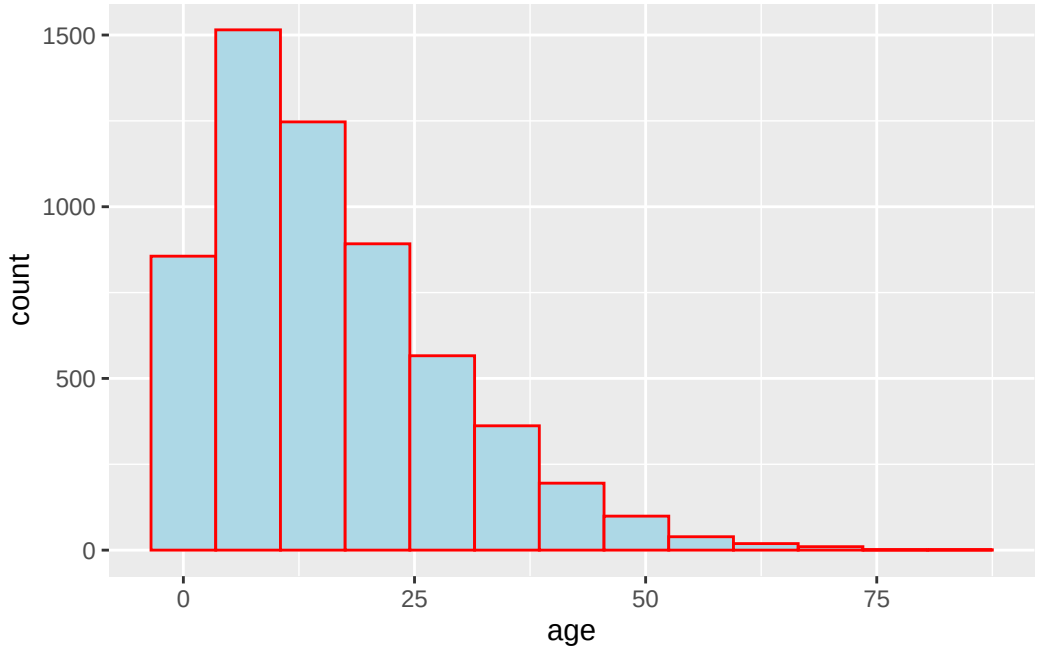
5.11.1 Color schemes

One thing that can initially be difficult to understand with **ggplot2** is control of color schemes. Note that this section discusses the color of *plot objects* (geoms/shapes) such as points, bars, lines, tiles, etc. To adjust color of accessory text, titles, or background color see the Themes section of the [ggplot basics] page.

To control "color" of *plot objects* you will be adjusting either the `color =` aesthetic (the *exterior* color) or the `fill =` aesthetic (the *interior* color). One exception to this pattern is `geom_point()`, where you really only get to control `color =`, which controls the color of the point (interior and exterior).

When setting colour or fill you can use colour names recognized by R like `"red"` (see complete list or enter `?colors`), or a specific hex colour such as `"#ff0505"`.

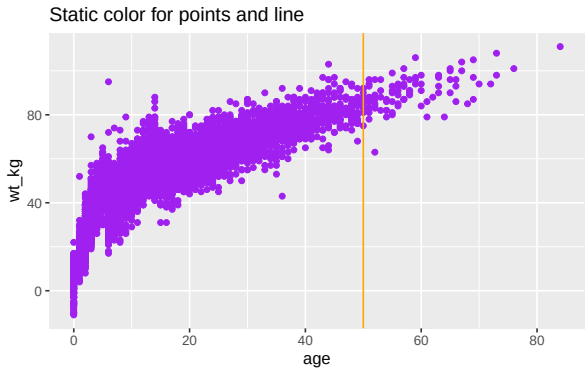
```
# histogram -
ggplot(data = linelist, mapping = aes(x = age))+       # set data and axes
  geom_histogram(              # display histogram
    binwidth = 7,                # width of bins
    color = "red",               # bin line color
    fill = "lightblue")          # bin interior color (fill)
```

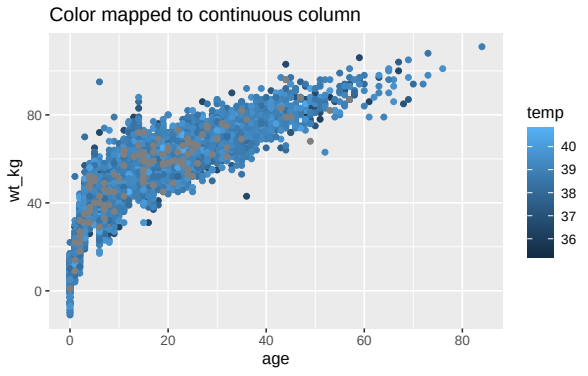


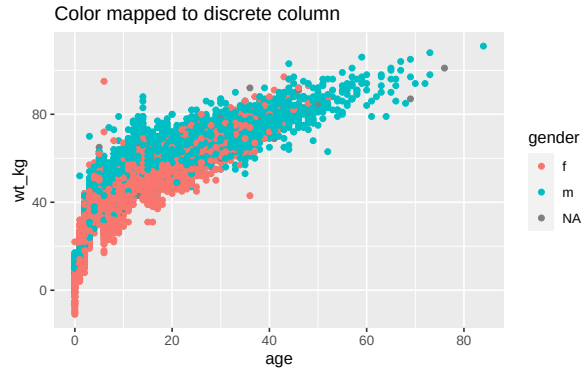
As explained the [ggplot basics] section on mapping data to the plot, aesthetics such as `fill =` and `color =` can be defined either *outside* of a `mapping = aes()` statement or *inside* of one. If *outside* the `aes()`, the assigned value should be static (e.g. `color = "blue"`) and will apply for *all* data plotted by the geom. If *inside*, the aesthetic should be mapped to a column, like `color = hospital`, and the expression will vary by the value for that row in the data. A few examples:

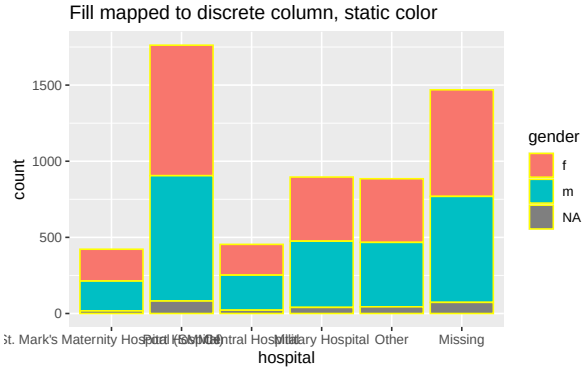
```
# Static color for points and for line
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(color = "purple")+
  geom_vline(xintercept = 50, color = "orange")+
```

```
  labs(title = "Static color for points and line")

# Color mapped to continuous column
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(mapping = aes(color = temp))+
  labs(title = "Color mapped to continuous column")

# Color mapped to discrete column
ggplot(data = linelist, mapping = aes(x = age, y = wt_kg))+
  geom_point(mapping = aes(color = gender))+
  labs(title = "Color mapped to discrete column")

# bar plot, fill to discrete column, color to static value
ggplot(data = linelist, mapping = aes(x = hospital))+
  geom_bar(mapping = aes(fill = gender), color = "yellow")+
  labs(title = "Fill mapped to discrete column, static color")
```









Scales

Once you map a column to a plot aesthetic (e.g. `x =`, `y =`, `fill =`, `color =`…), your plot will gain a scale/legend. See above how the scale can be continuous, discrete, date, etc. values depending on the class of the assigned column. If you have multiple aesthetics mapped to columns, your plot will have multiple scales.

You can control the scales with the appropriate `scales_()` function. The scale functions of **ggplot()** have 3 parts that are written like this: `scale_AESTHETIC_METHOD()`.

1) The first part, `scale_()`, is fixed.

2) The second part, the AESTHETIC, should be the aesthetic that you want to adjust the scale for (`_fill_`, `_shape_`, `_color_`, `_size_`, `_alpha_`…) - the options here also include `_x_` and `_y_`.

3) The third part, the METHOD, will be either `_discrete()`, `continuous()`, `_date()`, `_gradient()`, or `_manual()` depending on the class of the column and *how* you want to control it. There are others, but these are the most-often used.

Be sure that you use the correct function for the scale! Otherwise your scale command will not appear to change anything. If you have multiple scales, you may use multiple scale functions to adjust them! For example:

Scale arguments

Each kind of scale has its own arguments, though there is some overlap. Query the function like `?scale_color_discrete` in the R console to see the function argument documentation.

For continuous scales, use `breaks =` to provide a sequence of values with `seq()` (take `to =`, `from =`, and `by =` as shown in the example below. Set `expand = c(0,0)` to eliminate padding space around the axes (this can be used on any `_x_` or `_y_` scale.

For discrete scales, you can adjust the order of level appearance with `breaks =`, and how the values display with the `labels =` argument. Provide a character vector to each of those (see example below). You can also drop `NA` easily by setting `na.translate = FALSE`.

The nuances of date scales are covered more extensively in the [Epidemic curves] page.

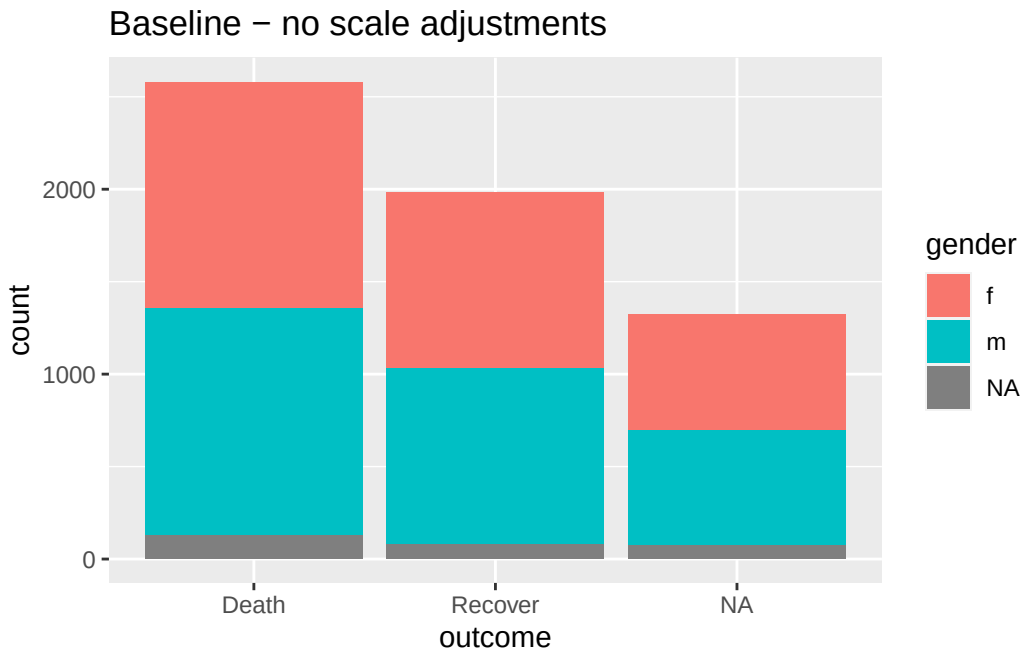
Manual adjustments

One of the most useful tricks is using "manual" scaling functions to explicitly assign colors as you desire. These are functions with the syntax `scale_xxx_manual()` (e.g. `scale_colour_manual()` or `scale_fill_manual()`). Each of the below arguments are demonstrated in the code example below.

- Assign colors to data values with the `values =` argument
- Specify a color for `NA` with `na.value =`
- Change how the values are *written* in the legend with the `labels =` argument
- Change the legend title with `name =`

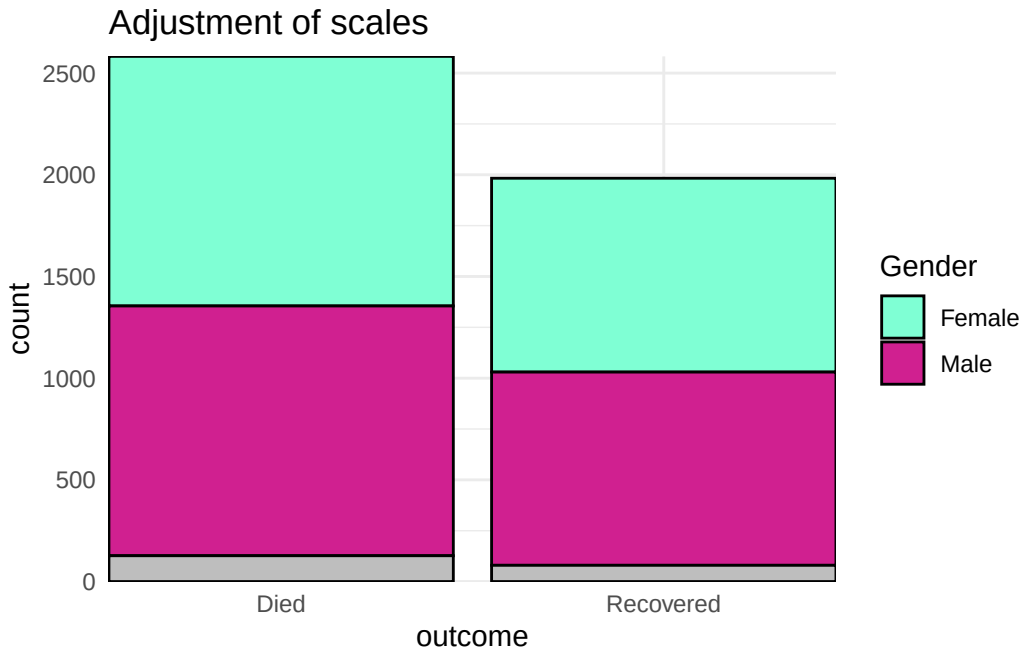
Below, we create a bar plot and show how it appears by default, and then with three scales adjusted - the continuous y-axis scale, the discrete x-axis scale, and manual adjustment of the fill (interior bar color).

```
# BASELINE - no scale adjustment
ggplot(data = linelist)+
  geom_bar(mapping = aes(x = outcome, fill = gender))+
  labs(title = "Baseline - no scale adjustments")
```




```
# SCALES ADJUSTED
ggplot(data = linelist)+

  geom_bar(mapping = aes(x = outcome, fill = gender), color = "black")+

  theme_minimal()+                    # simplify background

  scale_y_continuous(                 # continuous scale for y-axis (counts)
    expand = c(0,0),                  # no padding
    breaks = seq(from = 0,
                 to = 3000,
                 by = 500))+

  scale_x_discrete(                    # discrete scale for x-axis (gender)
    expand = c(0,0),                   # no padding
    drop = FALSE,                      # show all factor levels (even if not in data)
    na.translate = FALSE,              # remove NA outcomes from plot
    labels = c("Died", "Recovered"))+ # Change display of values


  scale_fill_manual(                   # Manually specify fill (bar interior color)
    values = c("m" = "violetred",      # reference values in data to assign colors
```

```
              "f" = "aquamarine"),
  labels = c("m" = "Male",          # re-label the legend (use "=" assignment to avoid m
             "f" = "Female",
             "Missing"),
  name = "Gender",                  # title of legend
  na.value = "grey"                 # assign a color for missing values
)+
labs(title = "Adjustment of scales") # Adjust the title of the fill legend
```

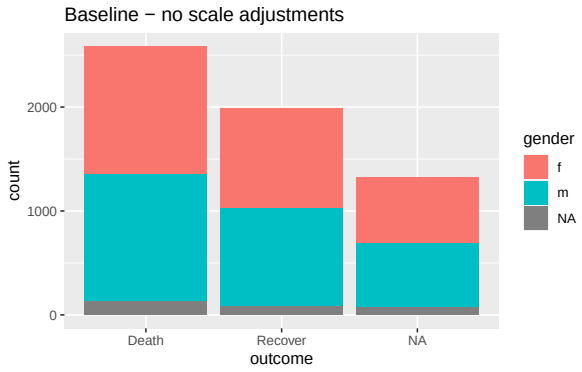


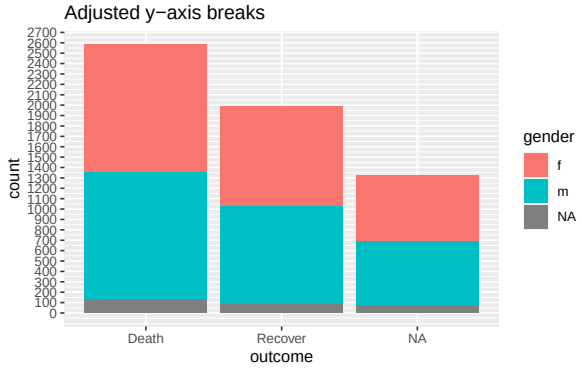
Continuous axes scales

When data are mapping to the plot axes, these too can be adjusted with scales commands. A common example is adjusting the display of an axis (e.g. y-axis) that is mapped to a column with continuous data.

We may want to adjust the breaks or display of the values in the ggplot using `scale_y_continuous()`. As noted above, use the argument `breaks =` to provide a sequence of values that will serve as "breaks" along the scale. These are the values at which numbers will display. To this argument, you can provide a `c()` vector containing the desired break values, or you can provide a regular sequence of numbers using the **base** R function `seq()`. This `seq()` function accepts `to =`, `from =`, and `by =`.

```r
# BASELINE - no scale adjustment
ggplot(data = linelist)+
  geom_bar(mapping = aes(x = outcome, fill = gender))+
  labs(title = "Baseline - no scale adjustments")

#
ggplot(data = linelist)+
  geom_bar(mapping = aes(x = outcome, fill = gender))+
  scale_y_continuous(
    breaks = seq(
      from = 0,
      to = 3000,
      by = 100)
  )+
  labs(title = "Adjusted y-axis breaks")
```





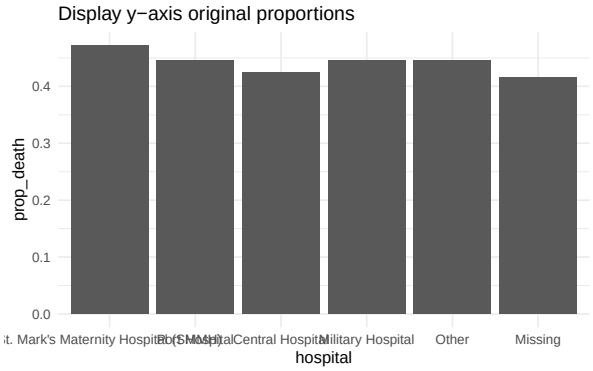
Display percents

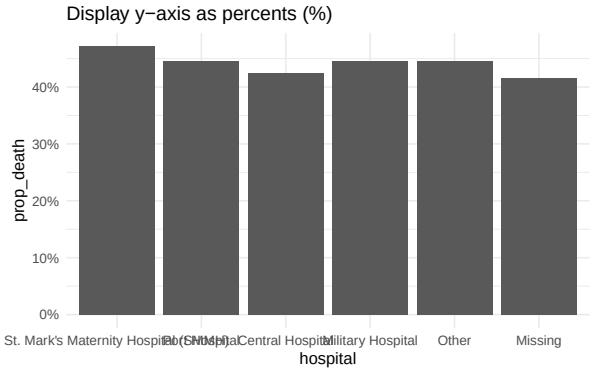
If your original data values are proportions, you can easily display them as percents with "%" by providing `labels = scales::percent` in your scales command, as shown below.

While an alternative would be to convert the values to character and add a "%" character to the end, this approach will cause complications because your data will no longer be continuous numeric values.

```
# Original y-axis proportions
############################
linelist %>%                                    # start with linelist
  group_by(hospital) %>%                        # group data by hospital
  summarise(                                    # create summary columns
    n = n(),                                      # total number of rows in group
    deaths = sum(outcome == "Death", na.rm=T),    # number of deaths in group
    prop_death = deaths/n) %>%                    # proportion deaths in group
  ggplot(                                       # begin plotting
    mapping = aes(
      x = hospital,
      y = prop_death))+
  geom_col()+
  theme_minimal()+
  labs(title = "Display y-axis original proportions")



# Display y-axis proportions as percents
########################################
linelist %>%
  group_by(hospital) %>%
  summarise(
    n = n(),
    deaths = sum(outcome == "Death", na.rm=T),
    prop_death = deaths/n) %>%
  ggplot(
    mapping = aes(
      x = hospital,
      y = prop_death))+
  geom_col()+
  theme_minimal()+
  labs(title = "Display y-axis as percents (%)")+
```

```
  scale_y_continuous(
    labels = scales::percent                    # display proportions as percents
  )
```





Log scale

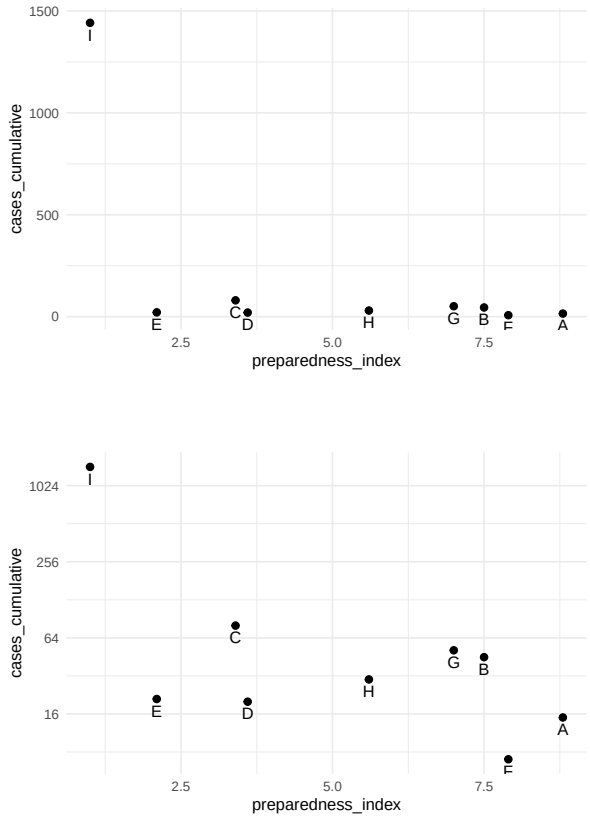
To transform a continuous axis to log scale, add `trans = "log2"` to the scale command. For purposes of example, we create a data frame of regions with their respective `preparedness_index` and cumulative cases values.

```
plot_data <- data.frame(
  region = c("A", "B", "C", "D", "E", "F", "G", "H", "I"),
  preparedness_index = c(8.8, 7.5, 3.4, 3.6, 2.1, 7.9, 7.0, 5.6, 1.0),
  cases_cumulative = c(15, 45, 80, 20, 21, 7, 51, 30, 1442)
)

plot_data
```

```
  region preparedness_index cases_cumulative
1      A                8.8               15
2      B                7.5               45
3      C                3.4               80
4      D                3.6               20
5      E                2.1               21
6      F                7.9                7
7      G                7.0               51
8      H                5.6               30
9      I                1.0             1442
```

The cumulative cases for region "I" are dramatically greater than all the other regions. In circumstances like this, you may elect to display the y-axis using a log scale so the reader can see differences between the regions with fewer cumulative cases.

```
# Original y-axis
preparedness_plot <- ggplot(data = plot_data,
       mapping = aes(
         x = preparedness_index,
         y = cases_cumulative))+
  geom_point(size = 2)+            # points for each region
  geom_text(
    mapping = aes(label = region),
    vjust = 1.5)+                  # add text labels
  theme_minimal()

preparedness_plot                  # print original plot


# print with y-axis transformed
preparedness_plot+                   # begin with plot saved above
  scale_y_continuous(trans = "log2") # add transformation for y-axis
```



Gradient scales

Fill gradient scales can involve additional nuance. The defaults are usually quite pleasing, but you may want to adjust the values, cutoffs, etc.

To demonstrate how to adjust a continuous color scale, we'll use a data set from the [Contact tracing] page that contains the ages of cases and of their source cases.

```
case_source_relationships <- rio::import(here::here("data", "godata", "relationships_clean
  select(source_age, target_age)
```

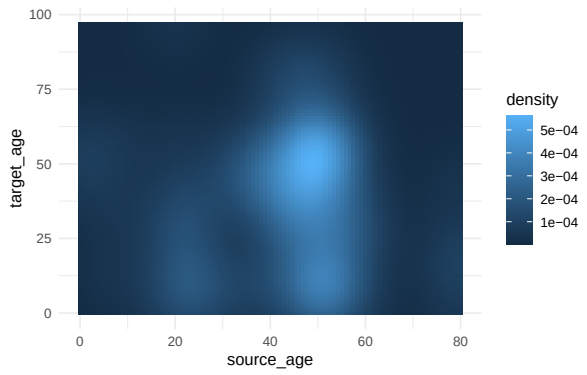
Below, we produce a "raster" heat tile density plot. We won't elaborate how (see the link in paragraph above) but we will focus on how we can adjust the color scale. Read more about the `stat_density2d()` **ggplot2** function here. Note how the `fill` scale is *continuous*.

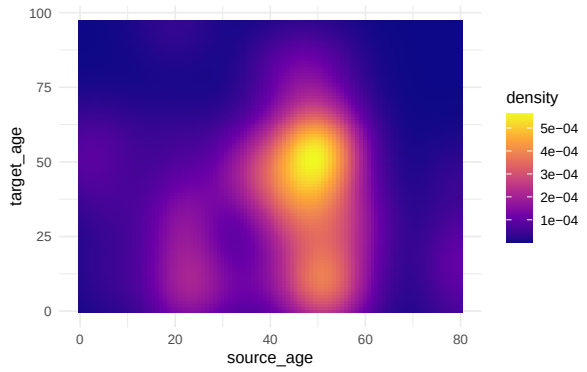
```
trans_matrix <- ggplot(
    data = case_source_relationships,
    mapping = aes(x = source_age, y = target_age))+
```

```
  stat_density2d(
    geom = "raster",
    mapping = aes(fill = after_stat(density)),
    contour = FALSE)+
  theme_minimal()
```

Now we show some variations on the fill scale:

```
trans_matrix
trans_matrix + scale_fill_viridis_c(option = "plasma")
```



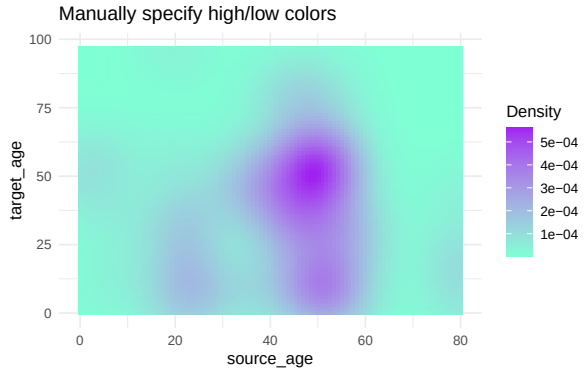


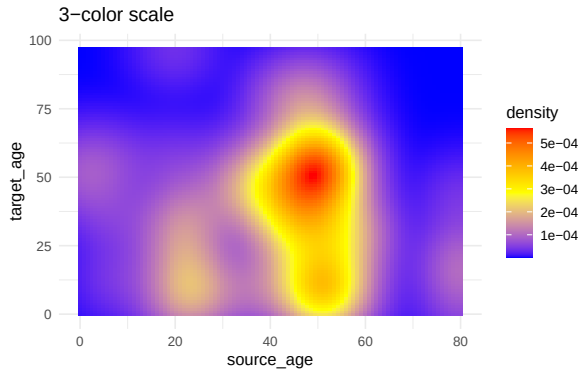
Now we show some examples of actually adjusting the break points of the scale:

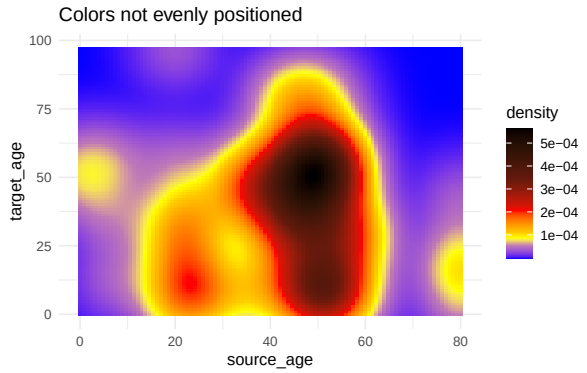
- `scale_fill_gradient()` accepts two colors (high/low)
- `scale_fill_gradientn()` accepts a vector of any length of colors to `values =` (intermediate values will be interpolated)

- Use `scales::rescale()` to adjust how colors are positioned along the gradient; it rescales your vector of positions to be between 0 and 1.

```
trans_matrix +
  scale_fill_gradient(     # 2-sided gradient scale
    low = "aquamarine",    # low value
    high = "purple",       # high value
    na.value = "grey",     # value for NA
    name = "Density")+     # Legend title
  labs(title = "Manually specify high/low colors")

# 3+ colors to scale
trans_matrix +
  scale_fill_gradientn(    # 3-color scale (low/mid/high)
    colors = c("blue", "yellow","red") # provide colors in vector
  )+
  labs(title = "3-color scale")

# Use of rescale() to adjust placement of colors along scale
trans_matrix +
  scale_fill_gradientn(    # provide any number of colors
    colors = c("blue", "yellow","red", "black"),
    values = scales::rescale(c(0, 0.05, 0.07, 0.10, 0.15, 0.20, 0.3, 0.5)) # positions for
    )+
  labs(title = "Colors not evenly positioned")

# use of limits to cut-off values that get fill color
trans_matrix +
  scale_fill_gradientn(
    colors = c("blue", "yellow","red"),
    limits = c(0, 0.0002))+
  labs(title = "Restrict value limits, resulting in grey space")
```


Manually specify high/low colors
target_age
source_age
Density
5e−04
4e−04
3e−04
2e−04
1e−04
0
20
40
60
80
0
25
50
75
100


3−color scale
target_age
source_age
density
5e−04
4e−04
3e−04
2e−04
1e−04
0
20
40
60
80
0
25
50
75
100


Colors not evenly positioned
target_age
source_age
density
5e−04
4e−04
3e−04
2e−04
1e−04
0
20
40
60
80
0
25
50
75
100

Palettes

Colorbrewer and Viridis

More generally, if you want predefined palettes, you can use the `scale_xxx_brewer` or `scale_xxx_viridis_y` functions.

The 'brewer' functions can draw from colorbrewer.org palettes.

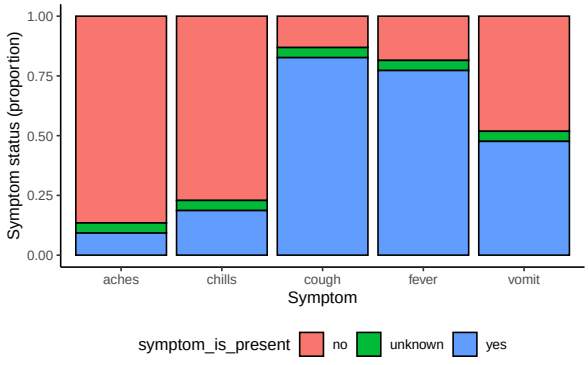
The 'viridis' functions draw from viridis (colourblind friendly!) palettes, which "provide colour maps that are perceptually uniform in both colour and black-and-white. They are also designed to be perceived by viewers with common forms of colour blindness." (read more here and here). Define if the palette is discrete, continuous, or binned by specifying this at the end of the function (e.g. discrete is `scale_xxx_viridis_d`).

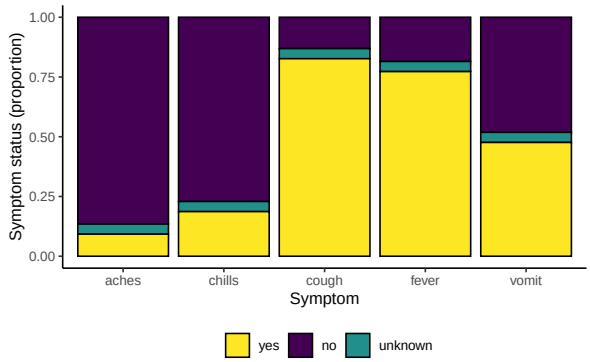
It is advised that you test your plot in this color blindness simulator. If you have a red/green color scheme, try a "hot-cold" (red-blue) scheme instead as described here

Here is an example from the [ggplot basics] page, using various color schemes.

```
symp_plot <- linelist %>%                                          # begin with linelist
  select(c(case_id, fever, chills, cough, aches, vomit)) %>%     # select columns
  pivot_longer(                                                    # pivot longer
    cols = -case_id,
    names_to = "symptom_name",
    values_to = "symptom_is_present") %>%
  mutate(                                                          # replace missing values
    symptom_is_present = replace_na(symptom_is_present, "unknown")) %>%
  ggplot(                                                          # begin ggplot!
    mapping = aes(x = symptom_name, fill = symptom_is_present))+
  geom_bar(position = "fill", col = "black") +
```

```
  theme_classic() +
  theme(legend.position = "bottom")+
  labs(
    x = "Symptom",
    y = "Symptom status (proportion)"
  )

symp_plot  # print with default colors

##################################
# print with manually-specified colors
symp_plot +
  scale_fill_manual(
    values = c("yes" = "black",          # explicitly define colours
               "no" = "white",
               "unknown" = "grey"),
    breaks = c("yes", "no", "unknown"), # order the factors correctly
    name = ""                           # set legend to no title

  )

##################################
# print with viridis discrete colors
symp_plot +
  scale_fill_viridis_d(
    breaks = c("yes", "no", "unknown"),
    name = ""
  )
```





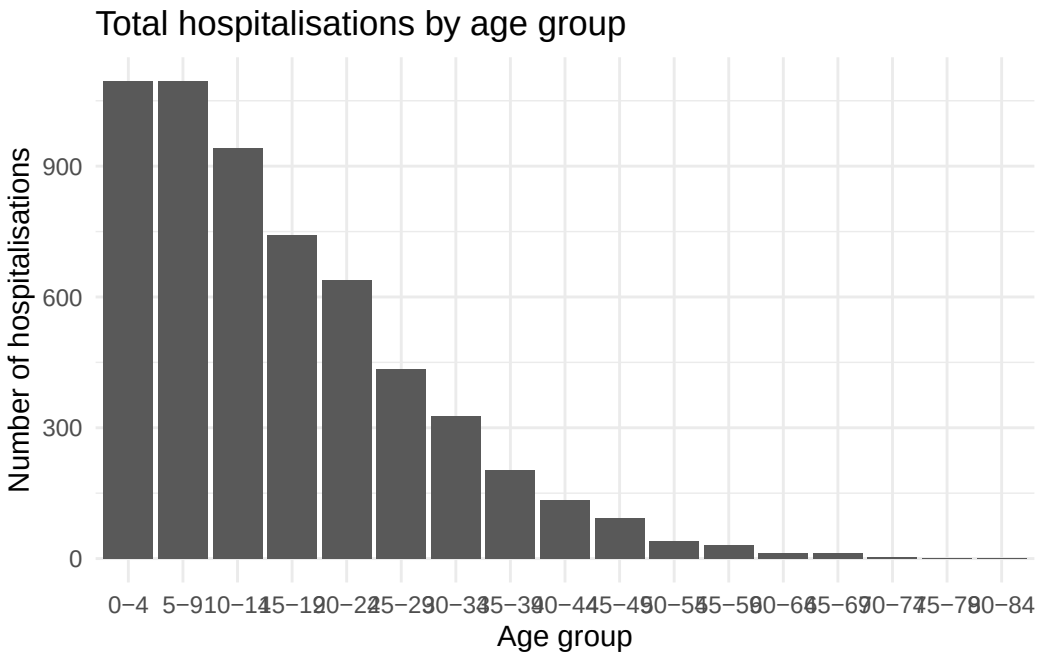
5.12 Change order of discrete variables

Changing the order that discrete variables appear in is often difficult to understand for people who are new to `ggplot2` graphs. It's easy to understand how to do this however once you understand how `ggplot2` handles discrete variables under the hood. Generally speaking, if a discrete varaible is used, it is automatically converted to a `factor` type - which orders factors by alphabetical order by default. To handle this, you simply have to reorder the factor levels to reflect the order you would like them to appear in the chart. For more detailed information on how to reorder `factor` objects, see the factor section of the guide.

We can look at a common example using age groups - by default the 5-9 age group will be placed in the middle of the age groups (given alphanumeric order), but we can move it behind the 0-4 age group of the chart by releveling the factors.

```
ggplot(
  data = linelist %>% drop_na(age_cat5),                          # remove rows where age_c
  mapping = aes(x = fct_relevel(age_cat5, "5-9", after = 1))) +  # relevel factor
```

```
geom_bar() +

labs(x = "Age group", y = "Number of hospitalisations",
     title = "Total hospitalisations by age group") +

theme_minimal()
```



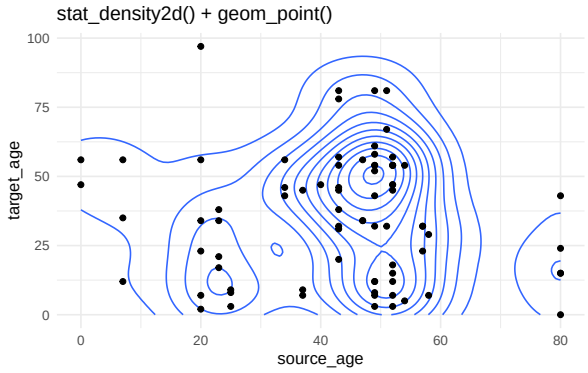
5.13 Advanced ggplot (optional)

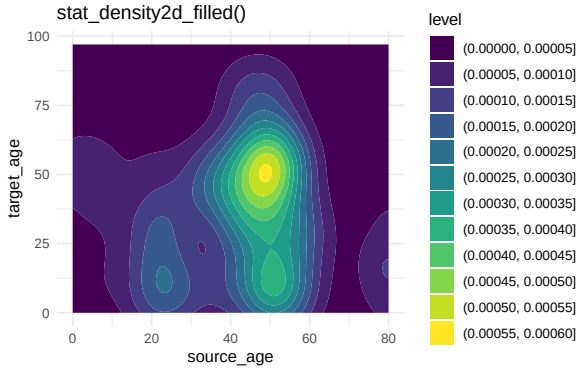
These are a collection of less common plot types, ggplot2 extensions, and advanced examples of some of the things you can do visualizing data in R.

5.13.1 Contour lines

Contour plots are helpful when you have many points that might cover each other ("over-plotting"). The case-source data used above are again plotted, but more simply using `stat_density2d()` and `stat_density2d_filled()` to produce discrete contour levels - like a topographical map. Read more about the statistics here.

```r
case_source_relationships %>%
  ggplot(aes(x = source_age, y = target_age))+
  stat_density2d()+
  geom_point()+
  theme_minimal()+
  labs(title = "stat_density2d() + geom_point()")


case_source_relationships %>%
  ggplot(aes(x = source_age, y = target_age))+
  stat_density2d_filled()+
  theme_minimal()+
  labs(title = "stat_density2d_filled()")
```





5.13.2 Marginal distributions

To show the distributions on the edges of a `geom_point()` scatterplot, you can use the **ggExtra** package and its function `ggMarginal()`. Save your original ggplot as an object, then pass it to `ggMarginal()` as shown below. Here are the key arguments:

- You must specify the `type =` as either "histogram", "density" "boxplot", "violin", or "densigram".
- By default, marginal plots will appear for both axes. You can set `margins =` to "x" or "y" if you only want one.
- Other optional arguments include `fill =` (bar color), `color =` (line color), `size =` (plot size relative to margin size, so larger number makes the marginal plot smaller).
- You can provide other axis-specific arguments to `xparams =` and `yparams =`. For example, to have different histogram bin sizes, as shown below.

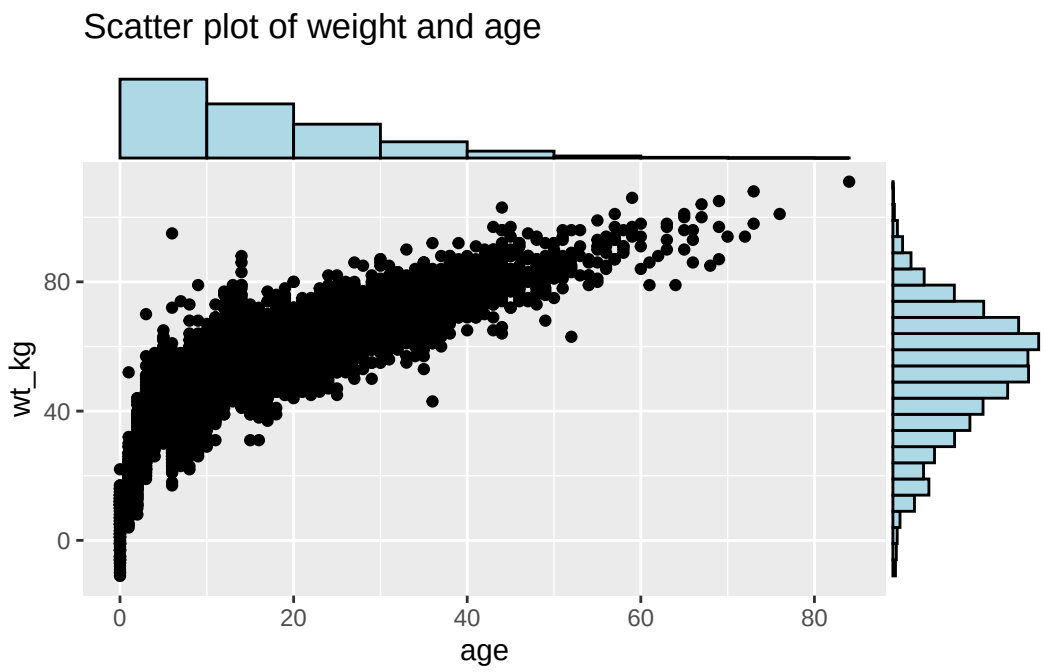
You can have the marginal plots reflect groups (columns that have been assigned to `color =` in your `ggplot()` mapped aesthetics). If this is the case, set the `ggMarginal()` argument `groupColour =` or `groupFill =` to `TRUE`, as shown below.

Read more at this vignette, in the R Graph Gallery or the function R documentation `?ggMarginal`.

```
# Install/load ggExtra
pacman::p_load(ggExtra)

# Basic scatter plot of weight and age
scatter_plot <- ggplot(data = linelist)+
  geom_point(mapping = aes(y = wt_kg, x = age)) +
  labs(title = "Scatter plot of weight and age")
```

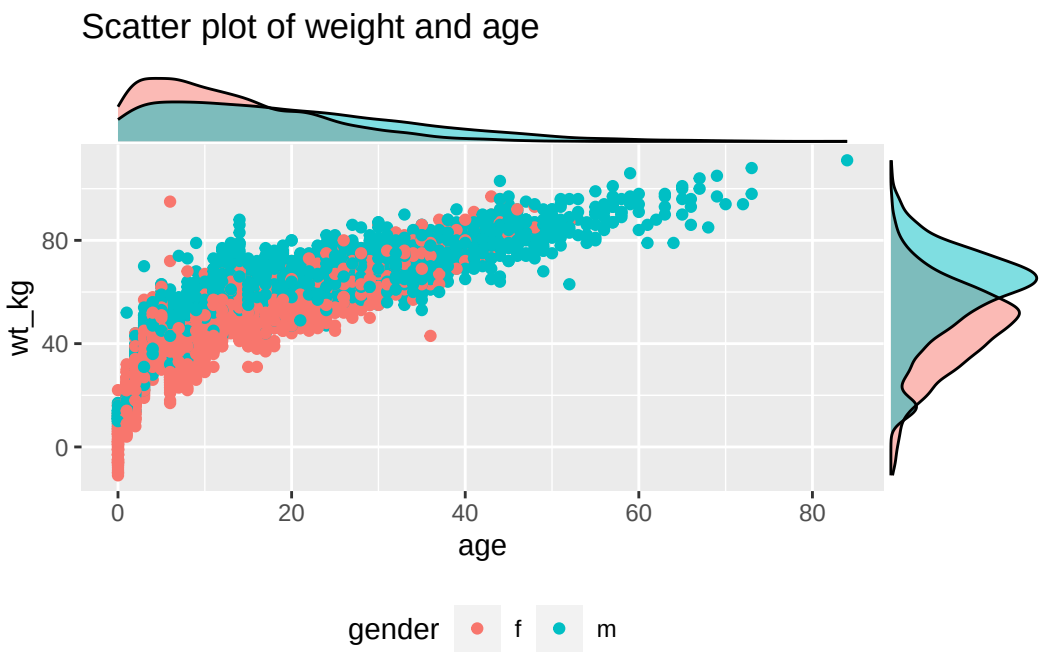
To add marginal histograms use `type = "histogram"`. You can optionally set `groupFill = TRUE` to get stacked histograms.

```
# with histograms
ggMarginal(
  scatter_plot,                    # add marginal histograms
  type = "histogram",              # specify histograms
  fill = "lightblue",              # bar fill
  xparams = list(binwidth = 10),   # other parameters for x-axis marginal
  yparams = list(binwidth = 5))    # other parameters for y-axis marginal
```



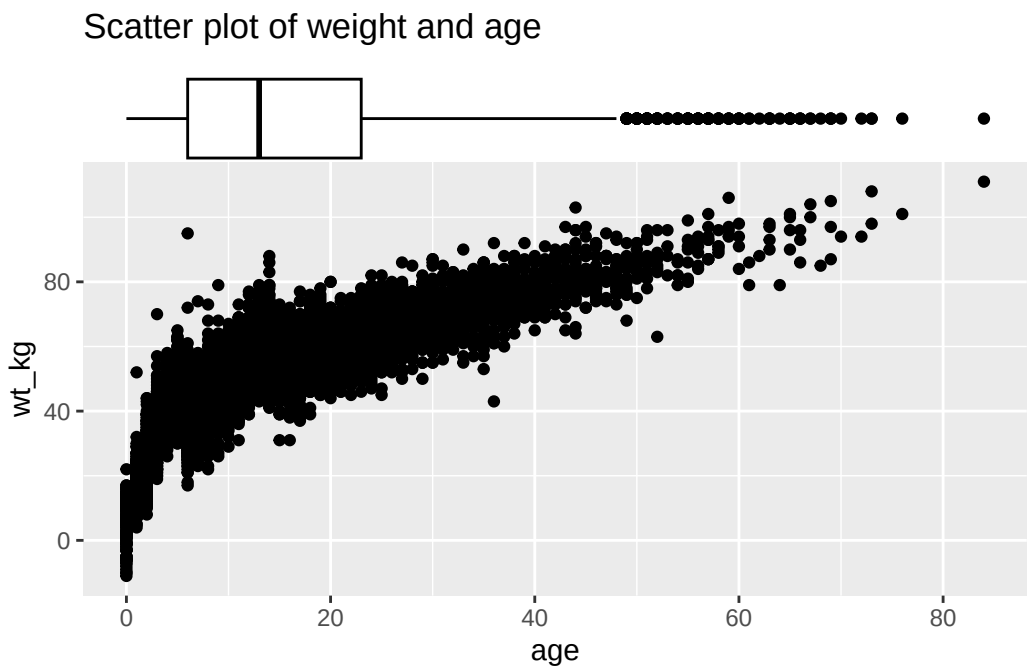
Marginal density plot with grouped/colored values:

```
# Scatter plot, colored by outcome
# Outcome column is assigned as color in ggplot. groupFill in ggMarginal set to TRUE
scatter_plot_color <- ggplot(data = linelist %>% drop_na(gender))+
  geom_point(mapping = aes(y = wt_kg, x = age, color = gender)) +
  labs(title = "Scatter plot of weight and age")+
  theme(legend.position = "bottom")

ggMarginal(scatter_plot_color, type = "density", groupFill = TRUE)
```



Set the `size =` arguemnt to adjust the relative size of the marginal plot. Smaller number makes a larger marginal plot. You also set `color =`. Below are is a marginal boxplot, with demonstration of the `margins =` argument so it appears on only one axis:

```
# with boxplot
ggMarginal(
  scatter_plot,
  margins = "x",       # only show x-axis marginal plot
  type = "boxplot")
```



5.13.3 Smart Labeling

In **ggplot2**, it is also possible to add text to plots. However, this comes with the notable limitation where text labels often clash with data points in a plot, making them look messy or hard to read. There is no ideal way to deal with this in the base package, but there is a **ggplot2** add-on, known as **ggrepel** that makes dealing with this very simple!

The **ggrepel** package provides two new functions, `geom_label_repel()` and `geom_text_repel()`, which replace `geom_label()` and `geom_text()`. Simply use these functions instead of the base functions to produce neat labels. Within the function, map the aesthetics `aes()` as always, but include the argument `label =` to which you provide a column name containing the values you want to display (e.g. patient id, or name, etc.). You can make more complex labels by combining columns and newlines (`\n`) within `str_glue()` as shown below.

A few tips:

- Use `min.segment.length = 0` to always draw line segments, or `min.segment.length = Inf` to never draw them

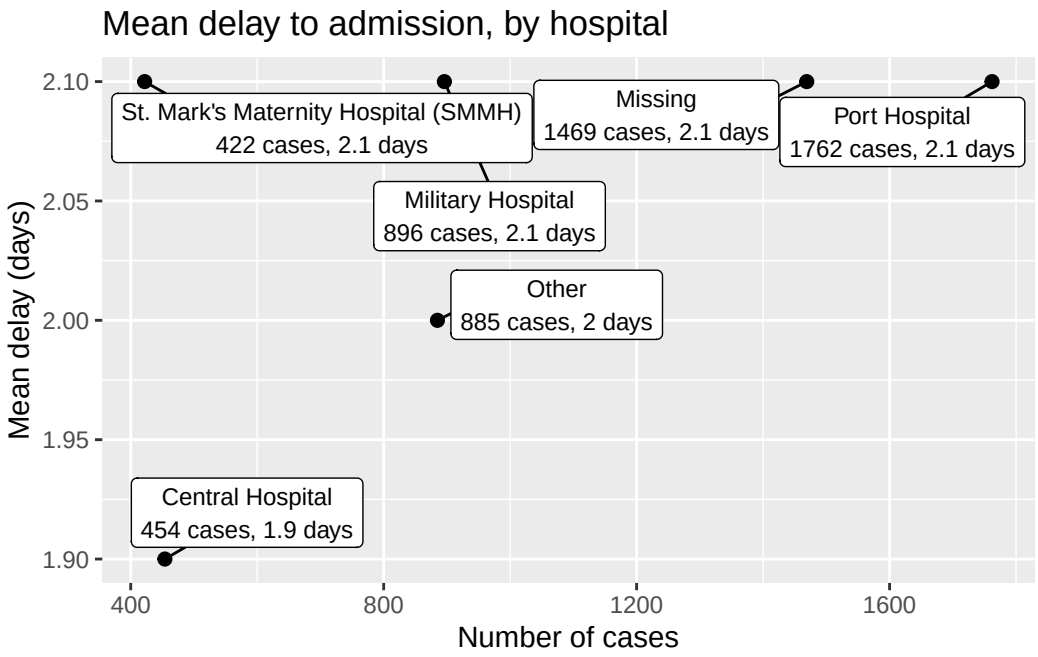
- Use `size =` outside of `aes()` to set text size

- Use `force =` to change the degree of repulsion between labels and their respective points (default is 1)

- Include `fill =` within `aes()` to have label colored by value
  - A letter "a" may appear in the legend - add `guides(fill = guide_legend(override.aes = aes(color = NA)))+` to remove it

See this is very in-depth tutorial for more.

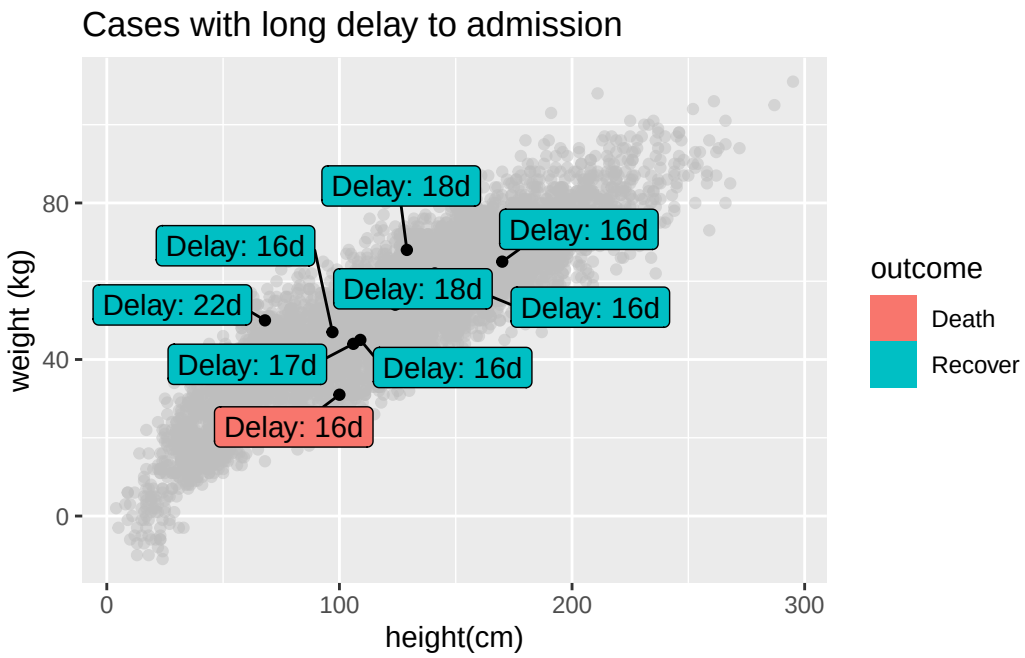
```
pacman::p_load(ggrepel)

linelist %>%                                                  # start with linelist
  group_by(hospital) %>%                                      # group by hospital
  summarise(                                                  # create new dataset with summa
    n_cases = n(),                                              # number of cases per hospita
    delay_mean = round(mean(days_onset_hosp, na.rm=T),1),       # mean delay per hospital
  ) %>%
  ggplot(mapping = aes(x = n_cases, y = delay_mean))+         # send data frame to ggplot
  geom_point(size = 2)+                                       # add points
  geom_label_repel(                                           # add point labels
    mapping = aes(
      label = stringr::str_glue(
        "{hospital}\n{n_cases} cases, {delay_mean} days")  # how label displays
      ),
    size = 3,                                                 # text size in labels
    min.segment.length = 0)+                                  # show all line segments
  labs(                                                       # add axes labels
    title = "Mean delay to admission, by hospital",
    x = "Number of cases",
    y = "Mean delay (days)")
```



You can label only a subset of the data points - by using standard `ggplot()` syntax to provide different `data =` for each `geom` layer of the plot. Below, All cases are plotted, but only a few are labeled.

```
ggplot()+
  # All points in grey
  geom_point(
    data = linelist,                                    # all data provided to this layer
    mapping = aes(x = ht_cm, y = wt_kg),
    color = "grey",
    alpha = 0.5)+                                                 # grey and semi-transparent

  # Few points in black
  geom_point(
    data = linelist %>% filter(days_onset_hosp > 15),  # filtered data provided to this la
    mapping = aes(x = ht_cm, y = wt_kg),
    alpha = 1)+                                                   # default black and not tra

  # point labels for few points
  geom_label_repel(
    data = linelist %>% filter(days_onset_hosp > 15),  # filter the data for the labels
    mapping = aes(
```

```
    x = ht_cm,
    y = wt_kg,
    fill = outcome,                                          # label color by outcome
    label = stringr::str_glue("Delay: {days_onset_hosp}d")), # label created with str_gl
  min.segment.length = 0) +                                  # show line segments for al

# remove letter "a" from inside legend boxes
guides(fill = guide_legend(override.aes = aes(color = NA)))+

# axis labels
labs(
  title = "Cases with long delay to admission",
  y = "weight (kg)",
  x = "height(cm)")
```



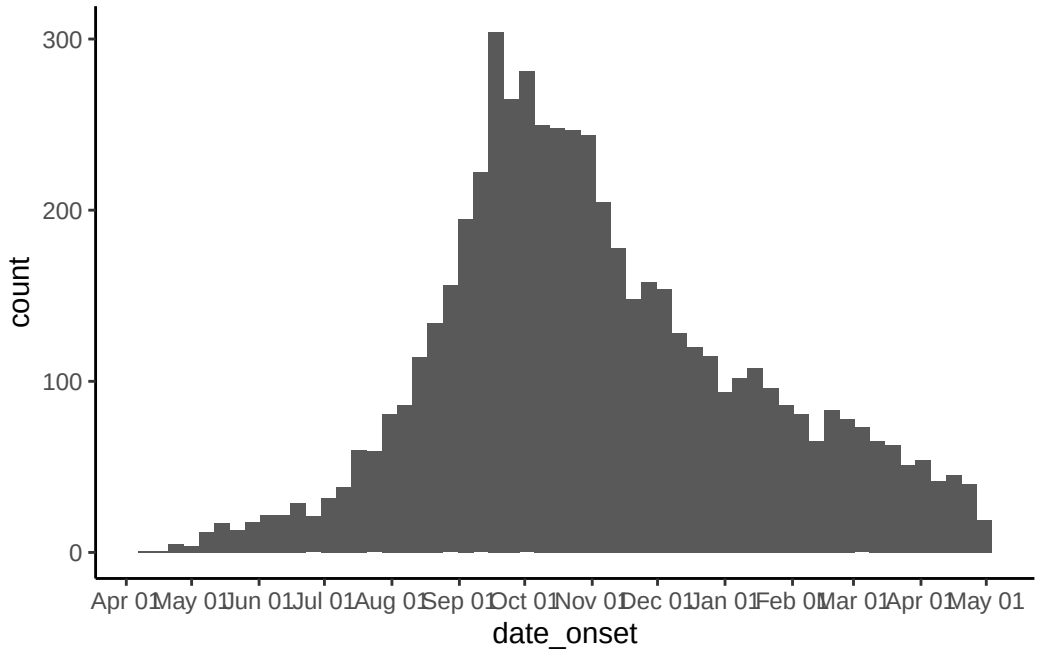
5.13.4 Time axes

Working with time axes in ggplot can seem daunting, but is made very easy with a few key functions. Remember that when working with time or date that you should ensure that the correct variables are formatted as date or datetime class - see the [Working with dates] page for more information on this, or [Epidemic curves] page (ggplot section) for examples.

The single most useful set of functions for working with dates in `ggplot2` are the scale functions (`scale_x_date()`, `scale_x_datetime()`, and their cognate y-axis functions). These functions let you define how often you have axis labels, and how to format axis labels. To find out how to format dates, see the *working with dates* section again! You can use the `date_breaks` and `date_labels` arguments to specify how dates should look:

1. `date_breaks` allows you to specify how often axis breaks occur - you can pass a string here (e.g. `"3 months"`, or "`2 days`")
2. `date_labels` allows you to define the format dates are shown in. You can pass a date format string to these arguments (e.g. `"%b-%d-%Y"`):

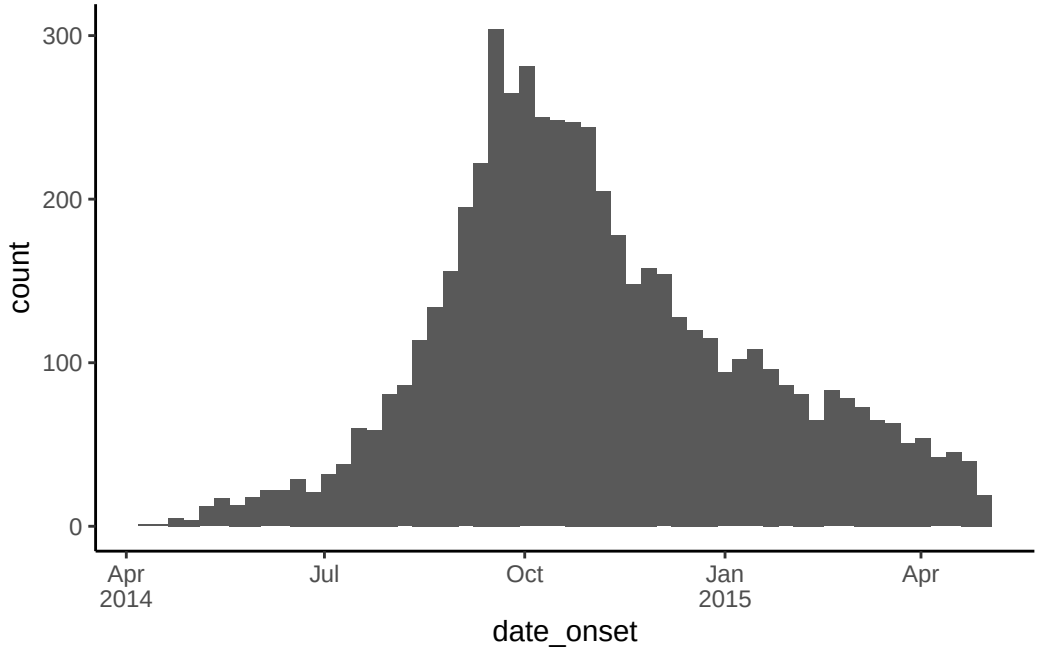
```
# make epi curve by date of onset when available
ggplot(linelist, aes(x = date_onset)) +
  geom_histogram(binwidth = 7) +
  scale_x_date(
    # 1 break every 1 month
    date_breaks = "1 months",
    # labels should show month then date
    date_labels = "%b %d"
  ) +
  theme_classic()
```



One easy solution to efficient date labels on the x-axis is to to assign the `labels =` argument in `scale_x_date()` to the function `label_date_short()` from the package **scales**. This function will automatically construct efficient date labels (read more here). An additional benefit of this function is that the labels will automatically adjust as your data expands over time, from days, to weeks, to months and years.

See a complete example in the Epicurves page section on multi-level date labels, but a quick example is shown below for reference:

```
ggplot(linelist, aes(x = date_onset)) +
  geom_histogram(binwidth = 7) +
  scale_x_date(
    labels = scales::label_date_short()  # automatically efficient date labels
  )+
  theme_classic()
```

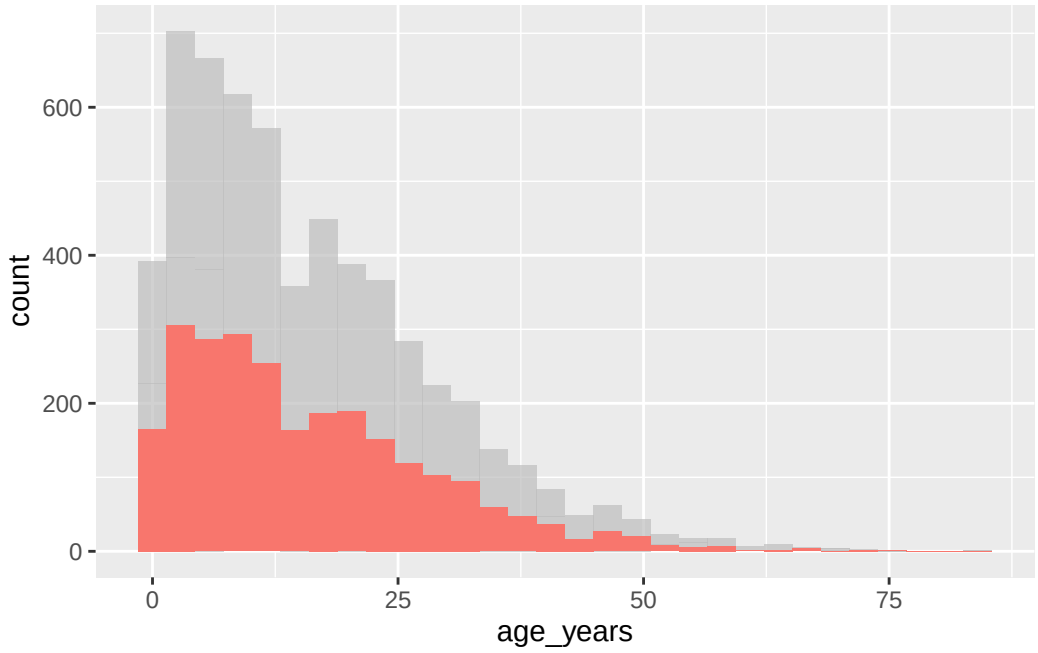


5.13.5 Highlighting

Highlighting specific elements in a chart is a useful way to draw attention to a specific instance of a variable while also providing information on the dispersion of the full dataset. While this is not easily done in base **ggplot2**, there is an external package that can help to do this known as **gghighlight**. This is easy to use within the ggplot syntax.

The **gghighlight** package uses the `gghighlight()` function to achieve this effect. To use this function, supply a logical statement to the function - this can have quite flexible outcomes, but here we'll show an example of the age distribution of cases in our linelist, highlighting them by outcome.

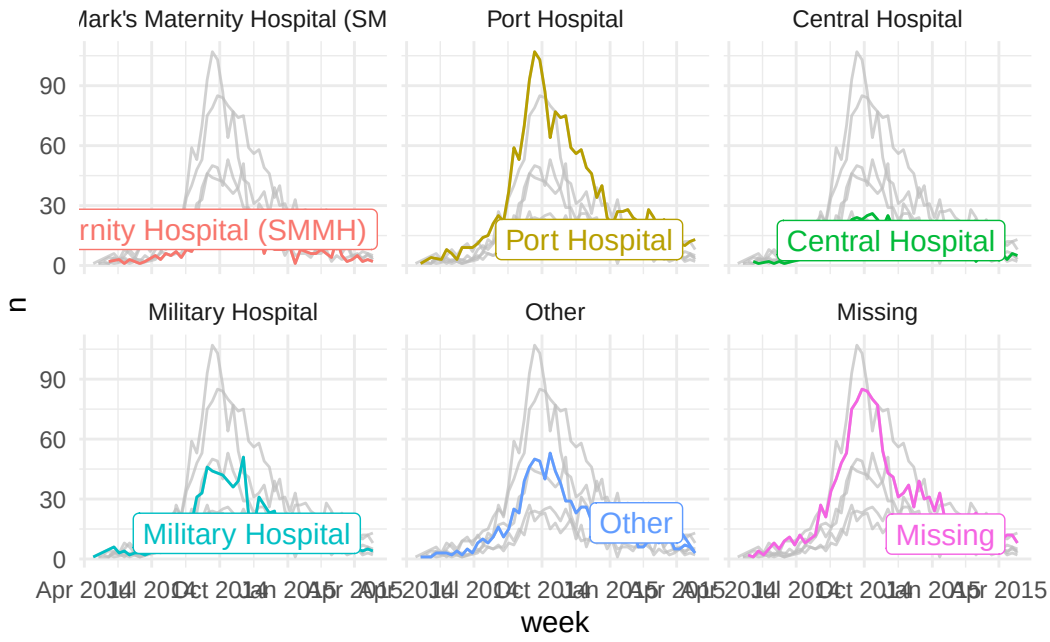
```
# load gghighlight
library(gghighlight)

# replace NA values with unknown in the outcome variable
linelist <- linelist %>%
  mutate(outcome = replace_na(outcome, "Unknown"))

# produce a histogram of all cases by age
ggplot(
  data = linelist,
  mapping = aes(x = age_years, fill = outcome)) +
  geom_histogram() +
  gghighlight::gghighlight(outcome == "Death")     # highlight instances where the patient
```



This also works well with faceting functions - it allows the user to produce facet plots with the background data highlighted that doesn't apply to the facet! Below we count cases by week and plot the epidemic curves by hospital (`color =` and `facet_wrap()` set to `hospital`

column).

```
# produce a histogram of all cases by age
linelist %>%
  count(week = lubridate::floor_date(date_hospitalisation, "week"),
        hospital) %>%
  ggplot()+
  geom_line(aes(x = week, y = n, color = hospital))+
  theme_minimal()+
  gghighlight::gghighlight() +                    # highlight instances where the patien
  facet_wrap(~hospital)                            # make facets by outcome
```



5.13.6 Plotting multiple datasets

Note that properly aligning axes to plot from multiple datasets in the same plot can be difficult. Consider one of the following strategies:

- Merge the data prior to plotting, and convert to "long" format with a column reflecting the dataset

- Use **cowplot** or a similar package to combine two plots (see below)

5.13.7 Combine plots

Two packages that are very useful for combining plots are **cowplot** and **patchwork**. In this page we will mostly focus on **cowplot**, with occassional use of **patchwork**.

Here is the online introduction to cowplot. You can read the more extensive documentation for each function online here. We will cover a few of the most common use cases and functions below.

The **cowplot** package works in tandem with **ggplot2** - essentially, you use it to arrange and combine ggplots and their legends into compound figures. It can also accept **base** R graphics.

```
pacman::p_load(
  tidyverse,      # data manipulation and visualisation
  cowplot,        # combine plots
  patchwork       # combine plots
)
```

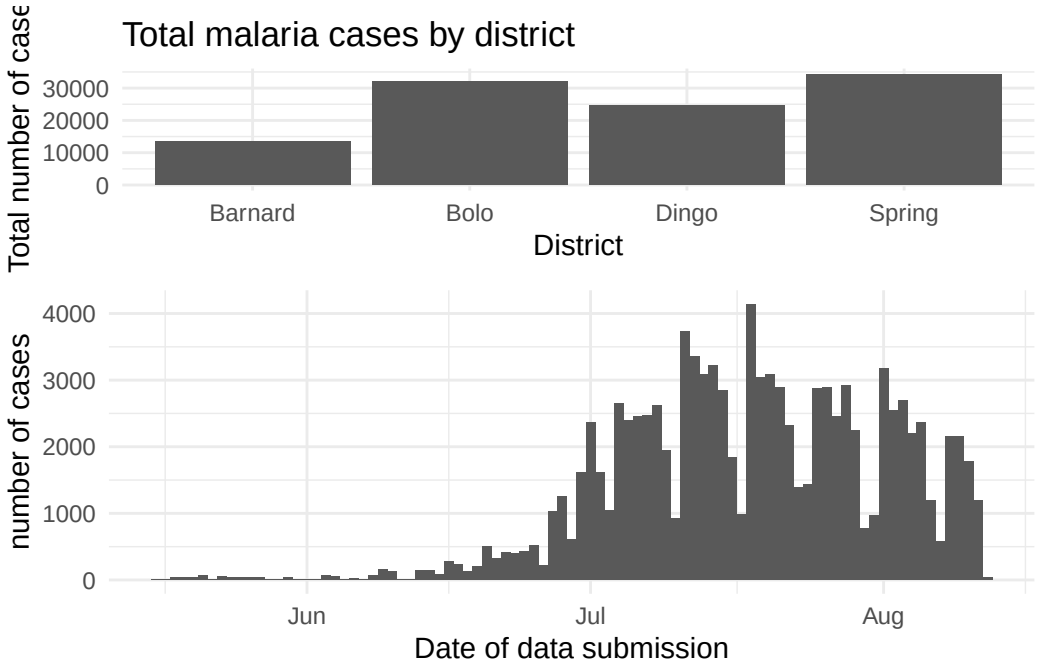
While faceting (described in the [ggplot basics] page) is a convenient approach to plotting, sometimes its not possible to get the results you want from its relatively restrictive approach. Here, you may choose to combine plots by sticking them together into a larger plot. There are three well known packages that are great for this - **cowplot**, **gridExtra**, and **patchwork**. However, these packages largely do the same things, so we'll focus on **cowplot** for this section.

`plot_grid()`

The **cowplot** package has a fairly wide range of functions, but the easiest use of it can be achieved through the use of `plot_grid()`. This is effectively a way to arrange predefined plots in a grid formation. We can work through another example with the malaria dataset - here we can plot the total cases by district, and also show the epidemic curve over time.

```
malaria_data <- rio::import(here::here("data", "malaria_facility_count_data.rds"))

# bar chart of total cases by district
p1 <- ggplot(malaria_data, aes(x = District, y = malaria_tot)) +
  geom_bar(stat = "identity") +
  labs(
    x = "District",
    y = "Total number of cases",
    title = "Total malaria cases by district"
  ) +
```

```
  theme_minimal()

# epidemic curve over time
p2 <- ggplot(malaria_data, aes(x = data_date, y = malaria_tot)) +
  geom_col(width = 1) +
  labs(
    x = "Date of data submission",
    y =  "number of cases"
  ) +
  theme_minimal()

cowplot::plot_grid(p1, p2,
                   # 1 column and two rows - stacked on top of each other
                    ncol = 1,
                    nrow = 2,
                    # top plot is 2/3 as tall as second
                    rel_heights = c(2, 3))
```



Combine legends

If your plots have the same legend, combining them is relatively straight-forward. Simple use the **cowplot** approach above to combine the plots, but remove the legend from one of them (de-duplicate).

If your plots have different legends, you must use an alternative approach:

1) Create and save your plots *without legends* using `theme(legend.position = "none")`

2) Extract the legends from each plot using `get_legend()` as shown below - *but extract legends from the plots modified to actually show the legend*

3) Combine the legends into a legends panel

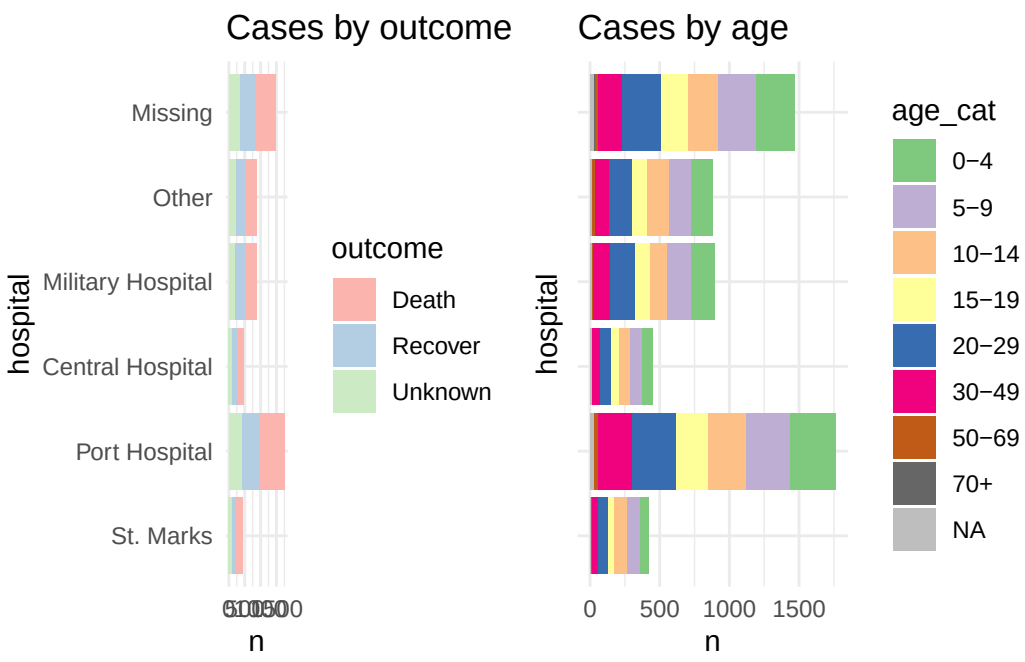
4) Combine the plots and legends panel

For demonstration we show the two plots separately, and then arranged in a grid with their own legends showing (ugly and inefficient use of space):

```
p1 <- linelist %>%
  mutate(hospital = recode(hospital, "St. Mark's Maternity Hospital (SMMH)" = "St. Marks")
  count(hospital, outcome) %>%
  ggplot()+
  geom_col(mapping = aes(x = hospital, y = n, fill = outcome))+
  scale_fill_brewer(type = "qual", palette = 4, na.value = "grey")+
  coord_flip()+
  theme_minimal()+
  labs(title = "Cases by outcome")


p2 <- linelist %>%
  mutate(hospital = recode(hospital, "St. Mark's Maternity Hospital (SMMH)" = "St. Marks")
  count(hospital, age_cat) %>%
  ggplot()+
  geom_col(mapping = aes(x = hospital, y = n, fill = age_cat))+
  scale_fill_brewer(type = "qual", palette = 1, na.value = "grey")+
  coord_flip()+
  theme_minimal()+
  theme(axis.text.y = element_blank())+
  labs(title = "Cases by age")
```

Here is how the two plots look when combined using `plot_grid()` without combining their legends:

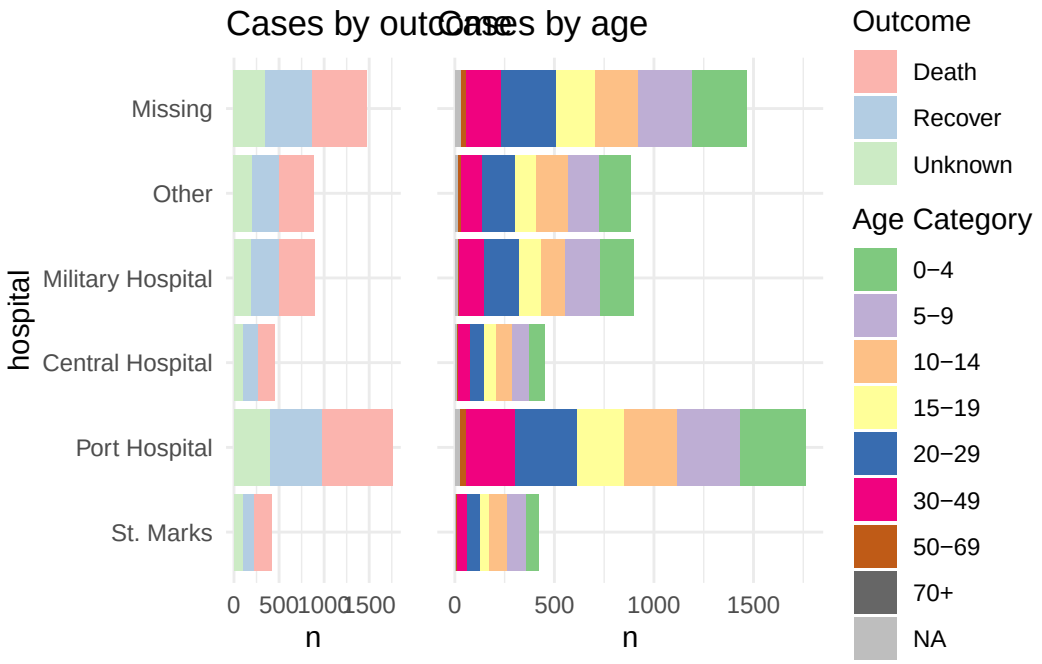
```
cowplot::plot_grid(p1, p2, rel_widths = c(0.3))
```



And now we show how to combine the legends. Essentially what we do is to define each plot *without* its legend (`theme(legend.position = "none")`), and then we define each plot's legend *separately*, using the `get_legend()` function from **cowplot**. When we extract the legend from the saved plot, we need to add `+` the legend back in, including specifying the placement ("right") and smaller adjustments for alignment of the legends and their titles. Then, we combine the legends together vertically, and then combine the two plots with the newly-combined legends. Voila!

```
# Define plot 1 without legend
p1 <- linelist %>%
  mutate(hospital = recode(hospital, "St. Mark's Maternity Hospital (SMMH)" = "St. Marks")
  count(hospital, outcome) %>%
  ggplot()+
  geom_col(mapping = aes(x = hospital, y = n, fill = outcome))+
  scale_fill_brewer(type = "qual", palette = 4, na.value = "grey")+
  coord_flip()+
  theme_minimal()+
  theme(legend.position = "none")+
  labs(title = "Cases by outcome")
```

```r
# Define plot 2 without legend
p2 <- linelist %>%
  mutate(hospital = recode(hospital, "St. Mark's Maternity Hospital (SMMH)" = "St. Marks")
  count(hospital, age_cat) %>%
  ggplot()+
  geom_col(mapping = aes(x = hospital, y = n, fill = age_cat))+
  scale_fill_brewer(type = "qual", palette = 1, na.value = "grey")+
  coord_flip()+
  theme_minimal()+
  theme(
    legend.position = "none",
    axis.text.y = element_blank(),
    axis.title.y = element_blank()
  )+
  labs(title = "Cases by age")


# extract legend from p1 (from p1 + legend)
leg_p1 <- cowplot::get_legend(p1 +
                                theme(legend.position = "right",        # extract vertical
                                      legend.justification = c(0,0.5))+ # so legends  alig
                                labs(fill = "Outcome"))                 # title of legend
# extract legend from p2 (from p2 + legend)
leg_p2 <- cowplot::get_legend(p2 +
                                theme(legend.position = "right",         # extract vertica
                                      legend.justification = c(0,0.5))+  # so legends alig
                                labs(fill = "Age Category"))             # title of legend

# create a blank plot for legend alignment
#blank_p <- patchwork::plot_spacer() + theme_void()

# create legends panel, can be one on top of the other (or use spacer commented above)
legends <- cowplot::plot_grid(leg_p1, leg_p2, nrow = 2, rel_heights = c(.3, .7))

# combine two plots and the combined legends panel
combined <- cowplot::plot_grid(p1, p2, legends, ncol = 3, rel_widths = c(.4, .4, .2))

combined  # print
```



This solution was learned from this post with a minor fix to align legends from this post.

TIP: Fun note - the "cow" in **cowplot** comes from the creator's name - Claus O. Wilke.

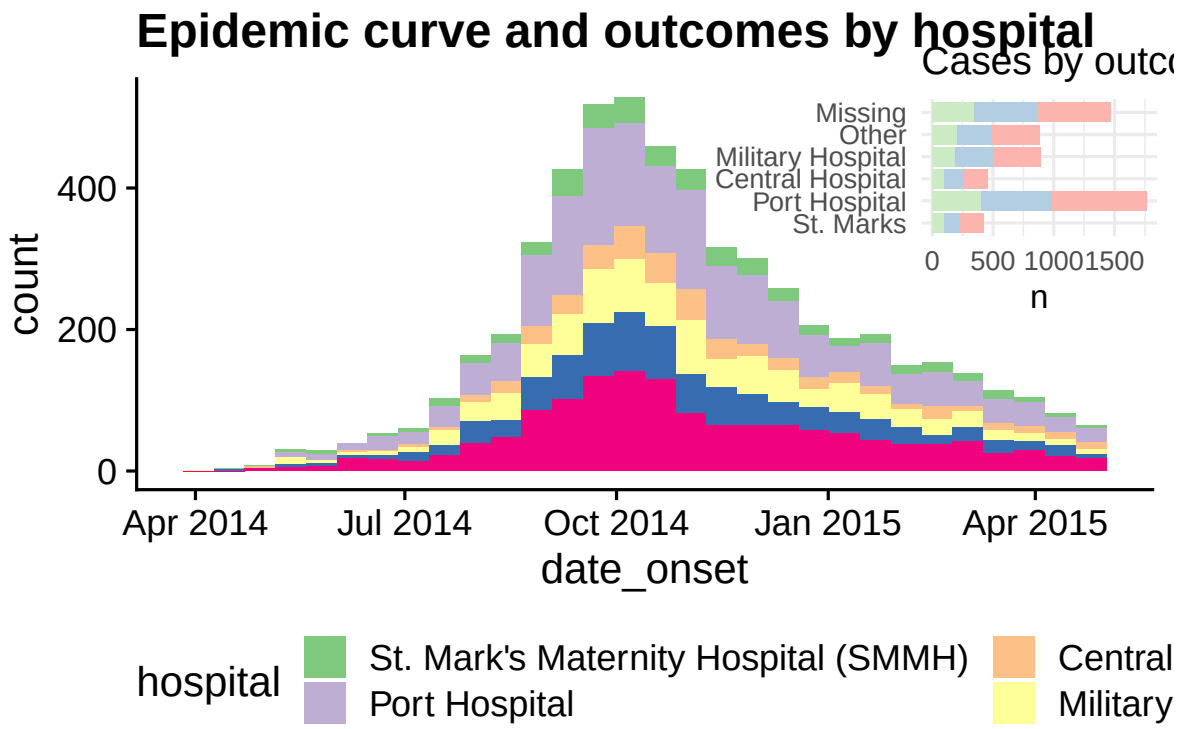
Inset plots

You can inset one plot in another using **cowplot**. Here are things to be aware of:

- Define the main plot with `theme_half_open()` from **cowplot**; it may be best to have the legend either on top or bottom
- Define the inset plot. Best is to have a plot where you do not need a legend. You can remove plot theme elements with `element_blank()` as shown below.
- Combine them by applying `ggdraw()` to the main plot, then adding `draw_plot()` on the inset plot and specifying the coordinates (x and y of lower left corner), height and width as proportion of the whole main plot.

```
# Define main plot
main_plot <- ggplot(data = linelist)+
  geom_histogram(aes(x = date_onset, fill = hospital))+
  scale_fill_brewer(type = "qual", palette = 1, na.value = "grey")+
```

```
  theme_half_open()+
  theme(legend.position = "bottom")+
  labs(title = "Epidemic curve and outcomes by hospital")


# Define inset plot
inset_plot <- linelist %>%
  mutate(hospital = recode(hospital, "St. Mark's Maternity Hospital (SMMH)" = "St. Marks")
  count(hospital, outcome) %>%
  ggplot()+
    geom_col(mapping = aes(x = hospital, y = n, fill = outcome))+
    scale_fill_brewer(type = "qual", palette = 4, na.value = "grey")+
    coord_flip()+
    theme_minimal()+
    theme(legend.position = "none",
          axis.title.y = element_blank())+
    labs(title = "Cases by outcome")


# Combine main with inset
cowplot::ggdraw(main_plot)+
     draw_plot(inset_plot,
               x = .6, y = .55,    #x = .07, y = .65,
               width = .4, height = .4)
```



This technique is explained more in these two vignettes:

Wilke lab
draw_plot() documentation

Part IV

Session 3: Managing Data

6 Managing Data

> Extended Materials
>
> You can find the original, extended version of these materials from chapters 8, 13, and 14.

This week we will be diving deeper into operations needed for data analysis. We will look at how to manipulate column names, group data by its variables, and append datasets together.

We will be continueing with the same simulated Ebola outbreak dataset.

```
linelist <- import("linelist_cleaned.rds")
```

6.1 Column names

In R, column *names* are the "header" or "top" value of a column. They are used to refer to columns in the code, and serve as a default label in figures.

Other statistical software such as SAS and STATA use *"labels"* that co-exist as longer printed versions of the shorter column names. While R does offer the possibility of adding column labels to the data, this is not emphasized in most practice. To make column names "printer-friendly" for figures, one typically adjusts their display within the plotting commands that create the outputs.

As R column names are used very often, so they must have "clean" syntax. We suggest the following:

- Short names
- No spaces (replace with underscores _)
- No unusual characters (&, #, <, >, …)

- Similar style nomenclature (e.g. all date columns named like **date_**onset, **date_**report, **date_**death…)

Re-naming columns manually is often necessary, even after the standardization step above. Below, re-naming is performed using the `rename()` function from the **dplyr** package, as part of a pipe chain. `rename()` uses the style `NEW = OLD` - the new column name is given before the old column name.

Below, a re-naming command is added to the cleaning pipeline. Spaces have been added strategically to align code for easier reading.

Now you can see that the columns names have been changed:

```
 [1] "case_id"             "generation"          "date_infection"
 [4] "date_onset"          "date_hospitalisation" "date_outcome"
 [7] "outcome"             "gender"              "age"
[10] "age_unit"            "age_years"           "age_cat"
[13] "age_cat5"            "hospital"            "lon"
[16] "lat"                 "infector"            "source"
[19] "wt_kg"               "ht_cm"               "ct_blood"
[22] "fever"               "chills"              "cough"
[25] "aches"               "vomit"               "temp"
[28] "time_admission"      "bmi"                 "days_onset_hosp"
```

Rename by column position

You can also rename by column position, instead of column name, for example:

```
rename(newNameForFirstColumn  = 1,
       newNameForSecondColumn = 2)
```

Rename via `select()` and `summarise()`

As a shortcut, you can also rename columns within the **dplyr** `select()` and `summarise()` functions. `select()` is used to keep only certain columns (and is covered later in this page). `summarise()` is covered in the [Grouping data] and [Descriptive tables] pages. These functions also uses the format `new_name = old_name`. Here is an example:

```
linelist_raw %>%
  select(# NEW name             # OLD name
         date_infection       = `infection date`,    # rename and KEEP ONLY these columns
         date_hospitalisation = `hosp date`)
```

Other conisderations with column names

Empty Excel column names

R cannot have dataset columns that do not have column names (headers). So, if you import an Excel dataset with data but no column headers, R will fill-in the headers with names like "...1" or "...2". The number represents the column number (e.g. if the 4th column in the dataset has no header, then R will name it "...4").

You can clean these names manually by referencing their position number (see example above), or their assigned name (`linelist_raw$...1`).

Merged Excel column names and cells

Merged cells in an Excel file are a common occurrence when receiving data. Merged cells can be nice for human reading of data, but are not "tidy data" and cause many problems for machine reading of data. R cannot accommodate merged cells.

One solution to deal with merged cells is to import the data with the function `readWorkbook()` from the package **openxlsx**. Set the argument `fillMergedCells = TRUE`. This gives the value in a merged cell to all cells within the merge range.

```
linelist_raw <- openxlsx::readWorkbook("linelist_raw.xlsx", fillMergedCells = TRUE)
```

6.2 Revisiting select

Two weeks ago we learned to use `select()` to select the columns we wanted to keep.

```
# linelist dataset is piped through select() command, and names() prints just the column n
linelist %>%
  select(case_id, date_onset, date_hospitalisation, fever) %>%
  names()  # display the column names
```

```
[1] "case_id"              "date_onset"           "date_hospitalisation"
[4] "fever"
```

Let's look at some more complicated scenarios when we need to think a bit deeper on how we're selecting or choosing columns in our data.

"tidyselect" helper functions

These helper functions exist to make it easy to specify columns to keep, discard, or transform. They are from the package **tidyselect**, which is included in **tidyverse** and underlies how columns are selected in **dplyr** functions.

For example, if you want to re-order the columns, `everything()` is a useful function to signify "all other columns not yet mentioned". The command below moves columns `date_onset` and `date_hospitalisation` to the beginning (left) of the dataset, but keeps all the other columns afterward. Note that `everything()` is written with empty parentheses:

```
# move date_onset and date_hospitalisation to beginning
linelist %>%
  select(date_onset, date_hospitalisation, everything()) %>%
  names()
```

```
 [1] "date_onset"           "date_hospitalisation" "case_id"
 [4] "generation"           "date_infection"       "date_outcome"
 [7] "outcome"              "gender"               "age"
[10] "age_unit"             "age_years"            "age_cat"
[13] "age_cat5"             "hospital"             "lon"
[16] "lat"                  "infector"             "source"
[19] "wt_kg"                "ht_cm"                "ct_blood"
[22] "fever"                "chills"               "cough"
[25] "aches"                "vomit"                "temp"
[28] "time_admission"       "bmi"                  "days_onset_hosp"
```

Here are other "tidyselect" helper functions that also work *within* **dplyr** functions like `select()`, `across()`, and `summarise()`:

- `everything()` - all other columns not mentioned
- `last_col()` - the last column
- `where()` - applies a function to all columns and selects those which are TRUE
- `contains()` - columns containing a character string
  - example: `select(contains("time"))`
- `starts_with()` - matches to a specified prefix

  - example: `select(starts_with("date_"))`

- `ends_with()` - matches to a specified suffix
  - example: `select(ends_with("_post"))`

- `matches()` - to apply a regular expression (regex)
  - example: `select(matches("[pt]al"))`

- `num_range()` - a numerical range like x01, x02, x03

- `any_of()` - matches IF column exists but returns no error if it is not found
  - example: `select(any_of(date_onset, date_death, cardiac_arrest))`

In addition, use normal operators such as `c()` to list several columns, `:` for consecutive columns, `!` for opposite, `&` for AND, and `|` for OR.

Use `where()` to specify logical criteria for columns. If providing a function inside `where()`, do not include the function's empty parentheses. The command below selects columns that are class Numeric.

```
# select columns that are class Numeric
linelist %>%
  select(where(is.numeric)) %>%
  names()
```

```
 [1] "generation"      "age"             "age_years"       "lon"
 [5] "lat"             "wt_kg"           "ht_cm"           "ct_blood"
 [9] "temp"            "bmi"             "days_onset_hosp"
```

Use `contains()` to select only columns in which the column name contains a specified character string. `ends_with()` and `starts_with()` provide more nuance.

```
# select columns containing certain characters
linelist %>%
  select(contains("date")) %>%
  names()
```

```
[1] "date_infection"       "date_onset"           "date_hospitalisation"
[4] "date_outcome"
```

The function `matches()` works similarly to `contains()` but can be provided a regular expression (see page on [Characters and strings]), such as multiple strings separated by OR bars within the parentheses:

```
# searched for multiple character matches
linelist %>%
  select(matches("onset|hosp|fev")) %>%   # note the OR symbol "|"
  names()
```

```
[1] "date_onset"           "date_hospitalisation" "hospital"
[4] "fever"                "days_onset_hosp"
```

6.3 Deduplication

In a later week we will learn more about how to de-duplicate data. Only a very simple row de-duplication example is presented here.

The package **dplyr** offers the `distinct()` function. This function examines every row and reduce the data frame to only the unique rows. That is, it removes rows that are 100% duplicates.

When evaluating duplicate rows, it takes into account a range of columns - by default it considers all columns. As shown in the de-duplication page, you can adjust this column range so that the uniqueness of rows is only evaluated in regards to certain columns.

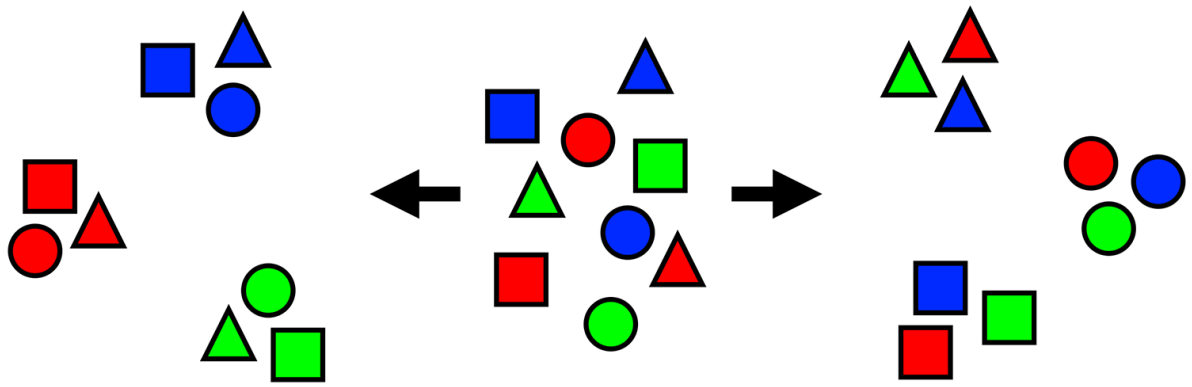
In this simple example, we just add the empty command `distinct()` to the pipe chain. This ensures there are no rows that are 100% duplicates of other rows (evaluated across all columns).

We begin with `nrow(linelist)` rows in `linelist`.

```
linelist <- linelist %>%
  distinct()
```

After de-duplication there are `nrow(linelist)` rows. Any removed rows would have been 100% duplicates of other rows.

7 Grouping data



Grouping data is a core component of data management and analysis. Grouped data statistically summarised by group, and can be plotted by group. Functions from the **dplyr** package (part of the **tidyverse**) make grouping and subsequent operations quite easy.

We will be examining:

- Grouping data with the `group_by()` function
- Un-grouping data
- `summarise()` grouped data with statistics
- The difference between `count()` and `tally()`
- `arrange()` applied to grouped data
- `filter()` applied to grouped data
- `mutate()` applied to grouped data
- `select()` applied to grouped data
- The **base** R `aggregate()` command as an alternative

Let's load a clean version of the simulated Ebola epidemic dataset.

```
linelist <- import("linelist_cleaned.rds")
```

7.1 Grouping

The function `group_by()` from **dplyr** groups the rows by the unique values in the column specified to it. If multiple columns are specified, rows are grouped by the unique combinations of values across the columns. Each unique value (or combination of values) constitutes a group. Subsequent changes to the dataset or calculations can then be performed within the context of each group.

For example, the command below takes the `linelist` and groups the rows by unique values in the column `outcome`, saving the output as a new data frame `ll_by_outcome`. The grouping column(s) are placed inside the parentheses of the function `group_by()`.

```
ll_by_outcome <- linelist %>%
  group_by(outcome)
```

Note that there is no perceptible change to the dataset after running `group_by()`, *until* another **dplyr** verb such as `mutate()`, `summarise()`, or `arrange()` is applied on the "grouped" data frame.

You can however "see" the groupings by printing the data frame. When you print a grouped data frame, you will see it has been transformed into a `tibble` class object which, when printed, displays which groupings have been applied and how many groups there are - written just above the header row.

```
# print to see which groups are active
ll_by_outcome
```

```
# A tibble: 5,888 x 30
# Groups:   outcome [3]
   case_id generation date_infection date_onset date_hospitalisation
   <chr>        <dbl> <date>         <date>     <date>
 1 5fe599           4 2014-05-08     2014-05-13 2014-05-15
 2 8689b7           4 NA             2014-05-13 2014-05-14
 3 11f8ea           2 NA             2014-05-16 2014-05-18
 4 b8812a           3 2014-05-04     2014-05-18 2014-05-20
 5 893f25           3 2014-05-18     2014-05-21 2014-05-22
 6 be99c8           3 2014-05-03     2014-05-22 2014-05-23
```

```
 7 07e3e8           4 2014-05-22     2014-05-27 2014-05-29
 8 369449           4 2014-05-28     2014-06-02 2014-06-03
 9 f393b4           4 NA             2014-06-05 2014-06-06
10 1389ca           4 NA             2014-06-05 2014-06-07
# i 5,878 more rows
# i 25 more variables: date_outcome <date>, outcome <chr>, gender <chr>,
#   age <dbl>, age_unit <chr>, age_years <dbl>, age_cat <fct>, age_cat5 <fct>,
#   hospital <chr>, lon <dbl>, lat <dbl>, infector <chr>, source <chr>,
#   wt_kg <dbl>, ht_cm <dbl>, ct_blood <dbl>, fever <chr>, chills <chr>,
#   cough <chr>, aches <chr>, vomit <chr>, temp <dbl>, time_admission <chr>,
#   bmi <dbl>, days_onset_hosp <dbl>
```

Unique groups

The groups created reflect each unique combination of values across the grouping columns.

To see the groups *and the number of rows in each group*, pass the grouped data to `tally()`. To see just the unique groups without counts you can pass to `group_keys()`.

See below that there are **three** unique values in the grouping column `outcome`: "Death", "Recover", and `NA`. See that there were `nrow(linelist %>% filter(outcome == "Death"))` deaths, `nrow(linelist %>% filter(outcome == "Recover"))` recoveries, and `nrow(linelist %>% filter(is.na(outcome)))` with no outcome recorded.

```r
linelist %>%
  group_by(outcome) %>%
  tally()
```

```
# A tibble: 3 x 2
  outcome     n
  <chr>   <int>
1 Death    2582
2 Recover  1983
3 <NA>     1323
```

You can group by more than one column. Below, the data frame is grouped by `outcome` and `gender`, and then tallied. Note how each unique combination of `outcome` and `gender` is registered as its own group - including missing values for either column.

```
linelist %>%
  group_by(outcome, gender) %>%
  tally()
```

```
# A tibble: 9 x 3
# Groups:   outcome [3]
  outcome gender     n
  <chr>   <chr>  <int>
1 Death   f       1227
2 Death   m       1228
3 Death   <NA>     127
4 Recover f        953
5 Recover m        950
6 Recover <NA>      80
7 <NA>    f        627
8 <NA>    m        625
9 <NA>    <NA>      71
```

New columns

You can also create a new grouping column *within* the `group_by()` statement. This is equivalent to calling `mutate()` before the `group_by()`. For a quick tabulation this style can be handy, but for more clarity in your code consider creating this column in its own `mutate()` step and then piping to `group_by()`.

```
# group dat based on a binary column created *within* the group_by() command
linelist %>%
  group_by(
    age_class = ifelse(age >= 18, "adult", "child")) %>%
  tally(sort = T)
```

```
# A tibble: 3 x 2
  age_class     n
  <chr>     <int>
1 child      3618
2 adult      2184
3 <NA>         86
```

Add/drop grouping columns

By default, if you run `group_by()` on data that are already grouped, the old groups will be removed and the new one(s) will apply. If you want to add new groups to the existing ones, include the argument `.add = TRUE`.

```
# Grouped by outcome
by_outcome <- linelist %>%
  group_by(outcome)

# Add grouping by gender in addition
by_outcome_gender <- by_outcome %>%
  group_by(gender, .add = TRUE)
```

** Keep all groups**

If you group on a column of class factor there may be levels of the factor that are not currently present in the data. If you group on this column, by default those non-present levels are dropped and not included as groups. To change this so that all levels appear as groups (even if not present in the data), set `.drop = FALSE` in your `group_by()` command.

7.2 Un-group

Data that have been grouped will remain grouped until specifically ungrouped via `ungroup()`. If you forget to ungroup, it can lead to incorrect calculations! Below is an example of removing all groupings:

```
linelist %>%
  group_by(outcome, gender) %>%
  tally() %>%
  ungroup()
```

You can also remove grouping for only specific columns, by placing the column name inside `ungroup()`.

```
linelist %>%
  group_by(outcome, gender) %>%
  tally() %>%
  ungroup(gender) # remove the grouping by gender, leave grouping by outcome
```

> Tip
>
> The verb `count()` automatically ungroups the data after counting.

7.3 Summarise

See the **dplyr** section of the [Descriptive tables] page for a detailed description of how to produce summary tables with `summarise()`. Here we briefly address how its behavior changes when applied to grouped data.

The **dplyr** function `summarise()` (or `summarize()`) takes a data frame and converts it into a *new* summary data frame, with columns containing summary statistics that you define. On an ungrouped data frame, the summary statistics will be calculated from all rows. Applying `summarise()` to grouped data produces those summary statistics *for each group.*

The syntax of `summarise()` is such that you provide the name(s) of the **new** summary column(s), an equals sign, and then a statistical function to apply to the data, as shown below. For example, `min()`, `max()`, `median()`, or `sd()`. Within the statistical function, list the column to be operated on and any relevant argument (e.g. `na.rm = TRUE`). You can use `sum()` to count the number of rows that meet a logical criteria (with double equals `==`).

Below is an example of `summarise()` applied *without grouped data.* The statistics returned are produced from the entire dataset.

```
# summary statistics on ungrouped linelist
linelist %>%
  summarise(
    n_cases  = n(),
    mean_age = mean(age_years, na.rm=T),
    max_age  = max(age_years, na.rm=T),
    min_age  = min(age_years, na.rm=T),
    n_males  = sum(gender == "m", na.rm=T))
```

```
  n_cases mean_age max_age min_age n_males
1    5888 16.01831      84       0    2803
```

In contrast, below is the same `summarise()` statement applied to grouped data. The statistics are calculated for each `outcome` group. Note how grouping columns will carry over into the new data frame.

```r
# summary statistics on grouped linelist
linelist %>%
  group_by(outcome) %>%
  summarise(
    n_cases  = n(),
    mean_age = mean(age_years, na.rm=T),
    max_age  = max(age_years, na.rm=T),
    min_age  = min(age_years, na.rm=T),
    n_males    = sum(gender == "m", na.rm=T))
```

```
# A tibble: 3 x 6
  outcome n_cases mean_age max_age min_age n_males
  <chr>     <int>    <dbl>   <dbl>   <dbl>   <int>
1 Death      2582     15.9      76       0    1228
2 Recover    1983     16.1      84       0     950
3 <NA>       1323     16.2      69       0     625
```

7.4 Counts and tallies

`count()` and `tally()` provide similar functionality but are different. Read more about the distinction between `tally()` and `count()` here

`tally()`

`tally()` is shorthand for `summarise(n = n())`, and *does not* group data. Thus, to achieve grouped tallys it must follow a `group_by()` command. You can add `sort = TRUE` to see the largest groups first.

```r
linelist %>%
  tally()
```

```
     n
1 5888
```

```r
linelist %>%
  group_by(outcome) %>%
  tally(sort = TRUE)
```

```
# A tibble: 3 x 2
  outcome     n
  <chr>   <int>
1 Death    2582
2 Recover  1983
3 <NA>     1323
```

`count()`

In contrast, `count()` does the following:

1) applies `group_by()` on the specified column(s)

2) applies `summarise()` and returns column `n` with the number of rows per group

3) applies `ungroup()`

```
linelist %>%
  count(outcome)
```

```
  outcome    n
1   Death 2582
2 Recover 1983
3    <NA> 1323
```

Just like with `group_by()` you can create a new column within the `count()` command:

```
linelist %>%
  count(age_class = ifelse(age >= 18, "adult", "child"), sort = T)
```

```
  age_class    n
1     child 3618
2     adult 2184
3      <NA>   86
```

`count()` can be called multiple times, with the functionality "rolling up". For example, to summarise the number of hospitals present for each gender, run the following. Note, the name of the final column is changed from default "n" for clarity (with `name =`).

```r
linelist %>%
  # produce counts by unique outcome-gender groups
  count(gender, hospital) %>%
  # gather rows by gender (3) and count number of hospitals per gender (6)
  count(gender, name = "hospitals per gender" )
```

```
  gender hospitals per gender
1      f                    6
2      m                    6
3   <NA>                    6
```

Add counts

In contrast to `count()` and `summarise()`, you can use `add_count()` to *add* a new column `n` with the counts of rows per group *while retaining all the other data frame columns*.

This means that a group's count number, in the new column `n`, will be printed in each row of the group. For demonstration purposes, we add this column and then re-arrange the columns for easier viewing. See the section below on filter on group size for another example.

```r
linelist %>%
  as_tibble() %>%                   # convert to tibble for nicer printing
  add_count(hospital) %>%           # add column n with counts by hospital
  select(hospital, n, everything()) # re-arrange for demo purposes
```

```
# A tibble: 5,888 x 31
   hospital                     n case_id generation date_infection date_onset
   <chr>                    <int> <chr>        <dbl> <date>         <date>
 1 Other                      885 5fe599           4 2014-05-08     2014-05-13
 2 Missing                   1469 8689b7           4 NA             2014-05-13
 3 St. Mark's Maternity Hosp~  422 11f8ea           2 NA             2014-05-16
 4 Port Hospital             1762 b8812a           3 2014-05-04     2014-05-18
 5 Military Hospital          896 893f25           3 2014-05-18     2014-05-21
 6 Port Hospital             1762 be99c8           3 2014-05-03     2014-05-22
 7 Missing                   1469 07e3e8           4 2014-05-22     2014-05-27
 8 Missing                   1469 369449           4 2014-05-28     2014-06-02
 9 Missing                   1469 f393b4           4 NA             2014-06-05
10 Missing                   1469 1389ca           4 NA             2014-06-05
# i 5,878 more rows
# i 25 more variables: date_hospitalisation <date>, date_outcome <date>,
```

```
#   outcome <chr>, gender <chr>, age <dbl>, age_unit <chr>, age_years <dbl>,
#   age_cat <fct>, age_cat5 <fct>, lon <dbl>, lat <dbl>, infector <chr>,
#   source <chr>, wt_kg <dbl>, ht_cm <dbl>, ct_blood <dbl>, fever <chr>,
#   chills <chr>, cough <chr>, aches <chr>, vomit <chr>, temp <dbl>,
#   time_admission <chr>, bmi <dbl>, days_onset_hosp <dbl>
```

Add totals

To easily add total *sum* rows or columns after using `tally()` or `count()` you can use the `tabyl` function from `janitor`. This package also offers functions like `adorn_totals()` and `adorn_percentages()` to add totals and convert to show percentages. Below is a brief example:

```
linelist %>%                                    # case linelist
  tabyl(age_cat, gender) %>%                    # cross-tabulate counts of two columns
  adorn_totals(where = "row") %>%               # add a total row
  adorn_percentages(denominator = "col") %>%  # convert to proportions with column denomin
  adorn_pct_formatting() %>%                    # convert proportions to percents
  adorn_ns(position = "front") %>%              # display as: "count (percent)"
  adorn_title(                                  # adjust titles
    row_name = "Age Category",
    col_name = "Gender")
```

	Gender					
Age Category	f		m		NA_	
0-4	640	(22.8%)	416	(14.8%)	39	(14.0%)
5-9	641	(22.8%)	412	(14.7%)	42	(15.1%)
10-14	518	(18.5%)	383	(13.7%)	40	(14.4%)
15-19	359	(12.8%)	364	(13.0%)	20	(7.2%)
20-29	468	(16.7%)	575	(20.5%)	30	(10.8%)
30-49	179	(6.4%)	557	(19.9%)	18	(6.5%)
50-69	2	(0.1%)	91	(3.2%)	2	(0.7%)
70+	0	(0.0%)	5	(0.2%)	1	(0.4%)
<NA>	0	(0.0%)	0	(0.0%)	86	(30.9%)
Total	2,807	(100.0%)	2,803	(100.0%)	278	(100.0%)

To add more complex totals rows that involve summary statistics other than *sums*, see this section of the Descriptive Tables page.

7.4.1 Arranging grouped data

Using the **dplyr** verb `arrange()` to order the rows in a data frame behaves the same when the data are grouped, *unless* you set the argument `.by_group =TRUE`. In this case the rows are ordered first by the grouping columns and then by any other columns you specify to `arrange()`.

7.4.2 Filter on grouped data

`filter()`

When applied in conjunction with functions that evaluate the data frame (like `max()`, `min()`, `mean()`), these functions will now be applied to the groups. For example, if you want to filter and keep rows where patients are above the median age, this will now apply per group - filtering to keep rows above the *group's* median age.

Slice rows per group

The **dplyr** function `slice()`, which filters rows based on their position in the data, can also be applied per group. Remember to account for sorting the data within each group to get the desired "slice".

For example, to retrieve only the latest 5 admissions from each hospital:

1) Group the linelist by column `hospital`

2) Arrange the records from latest to earliest `date_hospitalisation` *within each hospital group*

3) Slice to retrieve the first 5 rows from each hospital

```
linelist %>%
  group_by(hospital) %>%
  arrange(hospital, date_hospitalisation) %>%
  slice_head(n = 5) %>%
  arrange(hospital) %>%                          # for display
  select(case_id, hospital, date_hospitalisation)  # for display
```

```
# A tibble: 30 x 3
# Groups:   hospital [6]
   case_id hospital          date_hospitalisation
```

```
   <chr>   <chr>             <date>
 1 20b688  Central Hospital  2014-05-06
 2 d58402  Central Hospital  2014-05-10
 3 b8f2fd  Central Hospital  2014-05-13
 4 acf422  Central Hospital  2014-05-28
 5 275cc7  Central Hospital  2014-05-28
 6 d1fafd  Military Hospital 2014-04-17
 7 974bc1  Military Hospital 2014-05-13
 8 6a9004  Military Hospital 2014-05-13
 9 09e386  Military Hospital 2014-05-14
10 865581  Military Hospital 2014-05-15
# i 20 more rows
```

`slice_head()` - selects n rows from the top
`slice_tail()` - selects n rows from the end
`slice_sample()` - randomly selects n rows
`slice_min()` - selects n rows with highest values in `order_by =` column, use `with_ties = TRUE` to keep ties
`slice_max()` - selects n rows with lowest values in `order_by =` column, use `with_ties = TRUE` to keep ties

See the [De-duplication] page for more examples and detail on `slice()`.

Filter on group size

The function `add_count()` adds a column `n` to the original data giving the number of rows in that row's group.

Shown below, `add_count()` is applied to the column `hospital`, so the values in the new column `n` reflect the number of rows in that row's hospital group. Note how values in column `n` are repeated. In the example below, the column name `n` could be changed using `name =` within `add_count()`. For demonstration purposes we re-arrange the columns with `select()`.

```r
linelist %>%
  as_tibble() %>%
  add_count(hospital) %>%          # add "number of rows admitted to same hospital as this
  select(hospital, n, everything())
```

```
# A tibble: 5,888 x 31
   hospital                     n case_id generation date_infection date_onset
   <chr>                    <int> <chr>        <dbl> <date>         <date>
 1 Other                      885 5fe599           4 2014-05-08     2014-05-13
```

```
 2 Missing                      1469 8689b7           4 NA              2014-05-13
 3 St. Mark's Maternity Hosp~    422 11f8ea           2 NA              2014-05-16
 4 Port Hospital                1762 b8812a           3 2014-05-04      2014-05-18
 5 Military Hospital             896 893f25           3 2014-05-18      2014-05-21
 6 Port Hospital                1762 be99c8           3 2014-05-03      2014-05-22
 7 Missing                      1469 07e3e8           4 2014-05-22      2014-05-27
 8 Missing                      1469 369449           4 2014-05-28      2014-06-02
 9 Missing                      1469 f393b4           4 NA              2014-06-05
10 Missing                      1469 1389ca           4 NA              2014-06-05
# i 5,878 more rows
# i 25 more variables: date_hospitalisation <date>, date_outcome <date>,
#   outcome <chr>, gender <chr>, age <dbl>, age_unit <chr>, age_years <dbl>,
#   age_cat <fct>, age_cat5 <fct>, lon <dbl>, lat <dbl>, infector <chr>,
#   source <chr>, wt_kg <dbl>, ht_cm <dbl>, ct_blood <dbl>, fever <chr>,
#   chills <chr>, cough <chr>, aches <chr>, vomit <chr>, temp <dbl>,
#   time_admission <chr>, bmi <dbl>, days_onset_hosp <dbl>
```

It then becomes easy to filter for case rows who were hospitalized at a "small" hospital, say, a hospital that admitted fewer than 500 patients:

```
linelist %>%
  add_count(hospital) %>%
  filter(n < 500)
```

7.4.3 Mutate on grouped data

To retain all columns and rows (not summarise) and *add a new column containing group statistics*, use `mutate()` after `group_by()` instead of `summarise()`.

This is useful if you want group statistics in the original dataset *with all other columns present* - e.g. for calculations that compare one row to its group.

For example, this code below calculates the difference between a row's delay-to-admission and the median delay for their hospital. The steps are:

1) Group the data by hospital

2) Use the column `days_onset_hosp` (delay to hospitalisation) to create a new column containing the mean delay at the hospital of *that row*

3) Calculate the difference between the two columns

We `select()` only certain columns to display, for demonstration purposes.

```
linelist %>%
  # group data by hospital (no change to linelist yet)
  group_by(hospital) %>%

  # new columns
  mutate(
    # mean days to admission per hospital (rounded to 1 decimal)
    group_delay_admit = round(mean(days_onset_hosp, na.rm=T), 1),

    # difference between row's delay and mean delay at their hospital (rounded to 1 decima
    diff_to_group     = round(days_onset_hosp - group_delay_admit, 1)) %>%

  # select certain rows only - for demonstration/viewing purposes
  select(case_id, hospital, days_onset_hosp, group_delay_admit, diff_to_group)
```

```
# A tibble: 5,888 x 5
# Groups:   hospital [6]
   case_id hospital              days_onset_hosp group_delay_admit diff_to_group
   <chr>   <chr>                           <dbl>             <dbl>         <dbl>
 1 5fe599  Other                               2               2             0
 2 8689b7  Missing                             1               2.1          -1.1
 3 11f8ea  St. Mark's Maternity~               2               2.1          -0.1
 4 b8812a  Port Hospital                       2               2.1          -0.1
 5 893f25  Military Hospital                   1               2.1          -1.1
 6 be99c8  Port Hospital                       1               2.1          -1.1
 7 07e3e8  Missing                             2               2.1          -0.1
 8 369449  Missing                             1               2.1          -1.1
 9 f393b4  Missing                             1               2.1          -1.1
10 1389ca  Missing                             2               2.1          -0.1
# i 5,878 more rows
```

7.4.4 Select on grouped data

The verb `select()` works on grouped data, but the grouping columns are always included (even if not mentioned in `select()`). If you do not want these grouping columns, use `ungroup()` first.

7.5 Appending Datasets

We often need to combine multiple sources of data. Later on we'll see more complex methods for combining data based on matching ID's or other values. However, to start we'll look at the case where we want to add additional observations to a dataset. You can also think of this as "appending" or "adding" rows.

Bind rows

To bind rows of one data frame to the bottom of another data frame, use `bind_rows()` from **dplyr**. It is very inclusive, so any column present in either data frame will be included in the output. A few notes:

- Unlike the **base** R version `row.bind()`, **dplyr**'s `bind_rows()` does not require that the order of columns be the same in both data frames. As long as the column names are spelled identically, it will align them correctly.

- You can optionally specify the argument `.id =`. Provide a character column name. This will produce a new column that serves to identify which data frame each row originally came from.

- You can use `bind_rows()` on a `list` of similarly-structured data frames to combine them into one data frame. See an example in the [Iteration, loops, and lists] page involving the import of multiple linelists with **purrr**.

One common example of row binding is to bind a "total" row onto a descriptive table made with **dplyr**'s `summarise()` function. Below we create a table of case counts and median CT values by hospital with a total row.

The function `summarise()` is used on data grouped by hospital to return a summary data frame by hospital. But the function `summarise()` does not automatically produce a "totals" row, so we create it by summarising the data *again*, but with the data not grouped by hospital. This produces a second data frame of just one row. We can then bind these data frames together to achieve the final table.

See other worked examples like this in the [Descriptive tables] and [Tables for presentation] pages.

```
# Create core table
###################
hosp_summary <- linelist %>%
  group_by(hospital) %>%                          # Group data by hospital
  summarise(                                      # Create new summary columns of indicators
```

```
    cases = n(),                                  # Number of rows per hospital-outcome gr
    ct_value_med = median(ct_blood, na.rm=T))     # median CT value per group
```

Here is the `hosp_summary` data frame:

```
# A tibble: 6 x 3
  hospital                              cases ct_value_med
  <chr>                                 <int>        <dbl>
1 Central Hospital                        454           22
2 Military Hospital                       896           21
3 Missing                                1469           21
4 Other                                   885           22
5 Port Hospital                          1762           22
6 St. Mark's Maternity Hospital (SMMH)    422           22
```

Create a data frame with the "total" statistics (*not grouped by hospital*). This will return just one row.

```
# create totals
################
totals <- linelist %>%
  summarise(
    cases = n(),                                # Number of rows for whole dataset
    ct_value_med = median(ct_blood, na.rm=T))  # Median CT for whole dataset
```

And below is that `totals` data frame. Note how there are only two columns. These columns are also in `hosp_summary`, but there is one column in `hosp_summary` that is not in `totals` (`hospital`).

```
  cases ct_value_med
1  5888           22
```

Now we can bind the rows together with `bind_rows()`.

```
# Bind data frames together
combined <- bind_rows(hosp_summary, totals)
```

Now we can view the result. See how in the final row, an empty `NA` value fills in for the column `hospital` that was not in `hosp_summary`.

```
# A tibble: 7 x 3
  hospital                              cases ct_value_med
  <chr>                                 <int>        <dbl>
1 Central Hospital                        454           22
2 Military Hospital                       896           21
3 Missing                                1469           21
4 Other                                   885           22
5 Port Hospital                          1762           22
6 St. Mark's Maternity Hospital (SMMH)    422           22
7 <NA>                                   5888           22
```