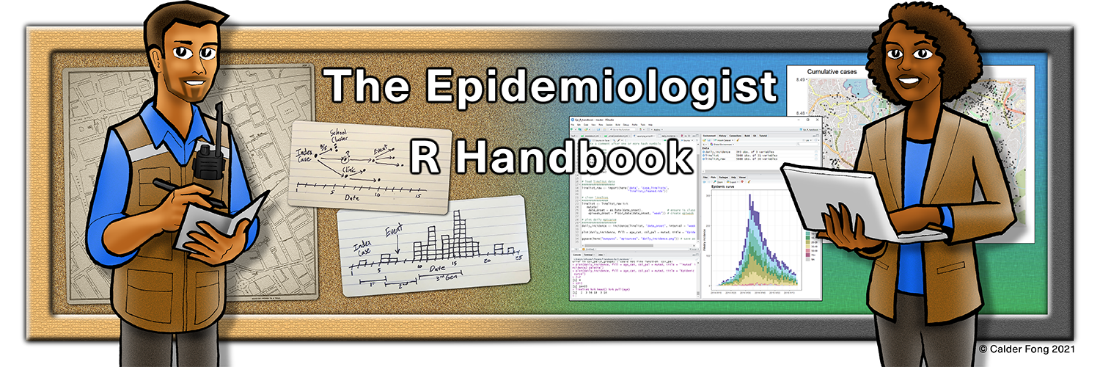
Code

* Show All Code
* Hide All Code

# The Epidemiologist R Handbook

#### the handbook team

#### 2021-06-04



## R for applied epidemiology and public health

**This handbook strives to:**

* Serve as a quick R code reference manual
* Provide task-centered examples addressing common epidemiological problems
* Assist epidemiologists transitioning to R
* Be accessible in settings with low internet-connectivity via an [**offline version**](#download-handbook-and-data)

**Written by epidemiologists, for epidemiologists**

We are applied epis from around the world, writing in our spare time to offer this resource to the community. Your encouragement and feedback is most welcome:

* Structured [**feedback form**](https://forms.gle/A5SnRVws7tPD15Js9)
* Email [**epiRhandbook@gmail.com**](mailto:epiRhandbook@gmail.com) or tweet [**@epiRhandbook**](https://twitter.com/epirhandbook)
* Submit issues to our [**Github repository**](https://github.com/epirhandbook/Epi_R_handbook)

## How to use this handbook

* Browse the pages in the Table of Contents, or use the search box
* Click the “copy” icons to copy code
* You can follow-along with [the example data](#download-handbook-and-data)
* See the “Resources” section of each page for further material

**Offline version**

See instructions in the [Download handbook and data](#download-handbook-and-data) page.

**Languages**

We want to translate this into languages other than English. If you can help, please contact us.

## Acknowledgements

This handbook is produced by a collaboration of epidemiologists from around the world drawing upon experience with organizations including local, state, provincial, and national health agencies, the World Health Organization (WHO), Médecins Sans Frontières / Doctors without Borders (MSF), hospital systems, and academic institutions.

This handbook is **not** an approved product of any specific organization. Although we strive for accuracy, we provide no guarantee of the content in this book.

### Contributors

**Editor-in-Chief:** [Neale Batra](https://www.linkedin.com/in/neale-batra/)

**Project core team:** [Neale Batra](https://www.linkedin.com/in/neale-batra/), [Alex Spina](https://github.com/aspina7), [Amrish Baidjoe](https://twitter.com/Ammer_B), Pat Keating, [Henry Laurenson-Schafer](https://github.com/henryls1), [Finlay Campbell](https://github.com/finlaycampbell)

**Authors**: [Neale Batra](https://www.linkedin.com/in/neale-batra/), [Alex Spina](https://github.com/aspina7), [Paula Blomquist](https://www.linkedin.com/in/paula-bianca-blomquist-53188186/), [Finlay Campbell](https://github.com/finlaycampbell), [Henry Laurenson-Schafer](https://github.com/henryls1), [Isaac Florence](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\www.Twitter.com\isaacatflorence), [Natalie Fischer](https://www.linkedin.com/in/nataliefischer211/), [Aminata Ndiaye](https://twitter.com/aminata_fadl), [Liza Coyer](https://www.linkedin.com/in/liza-coyer-86022040/), [Jonathan Polonsky](https://twitter.com/jonny_polonsky), [Yurie Izawa](https://ch.linkedin.com/in/yurie-izawa-a1590319), [Chris Bailey](https://twitter.com/cbailey_58?lang=en), [Daniel Molling](https://www.linkedin.com/in/daniel-molling-4005716a/), [Isha Berry](https://twitter.com/ishaberry2), [Emma Buajitti](https://twitter.com/buajitti), [Mathilde Mousset](https://mathildemousset.wordpress.com/research/), [Sara Hollis](https://www.linkedin.com/in/saramhollis/), Wen Lin

**Reviewers**: Pat Keating, Annick Lenglet, Margot Charette, Danielly Xavier, Esther Kukielka, Michelle Sloan, Aybüke Koyuncu, Rachel Burke, Kate Kelsey, [Berhe Etsay](https://www.linkedin.com/in/berhe-etsay-5752b1154/), John Rossow, Mackenzie Zendt, James Wright, Laura Haskins, [Flavio Finger](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\ffinger.github.io), Tim Taylor, [Jae Hyoung Tim Lee](https://www.linkedin.com/in/jaehyoungtlee/), [Brianna Bradley](https://www.linkedin.com/in/brianna-bradley-bb8658155), [Wayne Enanoria](https://www.linkedin.com/in/wenanoria), Manual Albela Miranda, [Molly Mantus](https://www.linkedin.com/in/molly-mantus-174550150/), Pattama Ulrich, Joseph Timothy, Adam Vaughan, Olivia Varsaneux, Lionel Monteiro, Joao Muianga

**Illustrations**: Calder Fong

### Funding and support

The handbook received supportive funding via a COVID-19 emergency capacity-building grant from [TEPHINET](https://www.tephinet.org/), the global network of Field Epidemiology Training Programs (FETPs).

Administrative support was provided by the EPIET Alumni Network ([EAN](https://epietalumni.net/)), with special thanks to Annika Wendland. EPIET is the European Programme for Intervention Epidemiology Training.

Special thanks to Médecins Sans Frontières (MSF) Operational Centre Amsterdam (OCA) for their support during the development of this handbook.

This publication was supported by Cooperative Agreement number NU2GGH001873, funded by the Centers for Disease Control and Prevention through TEPHINET, a program of The Task Force for Global Health. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the Centers for Disease Control and Prevention, the Department of Health and Human Services, The Task Force for Global Health, Inc. or TEPHINET.

### Inspiration

The multitude of tutorials and vignettes that provided knowledge for development of handbook content are credited within their respective pages.

More generally, the following sources provided inspiration for this handbook:  
[The “R4Epis” project](https://r4epis.netlify.app/) (a collaboration between MSF and RECON)  
[R Epidemics Consortium (RECON)](https://www.repidemicsconsortium.org/)  
[R for Data Science book (R4DS)](https://r4ds.had.co.nz/)  
[bookdown: Authoring Books and Technical Documents with R Markdown](https://bookdown.org/yihui/bookdown/)  
[Netlify](https://www.netlify.com) hosts this website

## Terms of Use and Contribution

### License

[](http://creativecommons.org/licenses/by-nc-sa/4.0/)  
This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-nc-sa/4.0/).

Academic courses and epidemiologist training programs are welcome to use this handbook with their students. If you have questions about your intended use, email [**epiRhandbook@gmail.com**](mailto:epiRhandbook@gmail.com).

### Citation

The Epidemiologist R Handbook.

### Contribution

If you would like to make a content contribution, please contact with us first via Github issues or by email. We are implementing a schedule for updates and are creating a contributor guide.

Please note that the epiRhandbook project is released with a [Contributor Code of Conduct](https://contributor-covenant.org/version/2/0/CODE_OF_CONDUCT.html). By contributing to this project, you agree to abide by its terms.

# I About this book

# Editorial and technical notes

In this page we describe the philosophical approach, style, and specific editorial decisions made during the creation of this handbook.

## Approach and style

The potential audience for this book is large. It will surely be used by people very new to R, and also by experienced R users looking for best practices and tips. So it must be both accessible and succinct. Therefore, our approach was to provide just enough text explanation that someone very new to R can apply the code and follow what the code is doing.

A few other points:

* This is a code reference book accompanied by relatively brief examples - not a thorough textbook on R or data science
* This is a R handbook for use within applied epidemiology - not a manual on the methods or science of applied epidemiology
* This is intended to be a living document - optimal R packages for a given task change often and we welcome discussion about which to emphasize in this handbook

### R packages

**So many choices**

One of the most challenging aspects of learning R is knowing which R package to use for a given task. It is a common occurrence to struggle through a task only later to realize - hey, there’s an R package that does all that in one command line!

In this handbook, we try to offer you at least two ways to complete each task: one tried-and-true method (probably in **base** R or **tidyverse**) and one special R package that is custom-built for that purpose. We want you to have a couple options in case you can’t download a given package or it otherwise does not work for you.

In choosing which packages to use, we prioritized R packages and approaches that have been tested and vetted by the community, minimize the number of packages used in a typical work session, that are stable (not changing very often), and that accomplish the task simply and cleanly

This handbook generally prioritizes R packages and functions from the **tidyverse**. Tidyverse is a collection of R packages designed for data science that share underlying grammar and data structures. All tidyverse packages can be installed or loaded via the **tidyverse** package. Read more at the [tidyverse website](https://www.tidyverse.org/).

When applicable, we also offer code options using **base** R - the packages and functions that come with R at installation. This is because we recognize that some of this book’s audience may not have reliable internet to download extra packages.

**Linking functions to packages explicitly**

It is often frustrating in R tutorials when a function is shown in code, but you don’t know which package it is from! We try to avoid this situation.

In the narrative text, package names are written in bold (e.g. **dplyr**) and functions are written like this: mutate(). We strive to be explicit about which package a function comes from, either by referencing the package in nearby text or by specifying the package explicitly in the code like this: dplyr::mutate(). It may look redundant, but we are doing it on purpose.

See the page on [R basics](#r-basics) to learn more about packages and functions.

### Code style

In the handbook, we frequently utilize “new lines”, making our code appear “long”. We do this for a few reasons:

* We can write explanatory comments with # that are adjacent to each little part of the code
* Generally, longer (vertical) code is easier to read
* It is easier to read on a narrow screen (no sideways scrolling needed)
* From the indentations, it can be easier to know which arguments belong to which function

As a result, code that could be written like this:

…is written like this:

R code is generally not affected by new lines or indentations. When writing code, if you initiate a new line after a comma it will apply automatic indentation patterns.

We also use lots of spaces (e.g. n = 1 instead of n=1) because it is easier to read. Be kind to the people reading your code!

### Nomenclature

In this handbook, we generally reference “columns” and “rows” instead of “variables” and “observations”. As explained in this primer on [“tidy data”](https://tidyr.tidyverse.org/articles/tidy-data.html), most epidemiological statistical datasets consist structurally of rows, columns, and values.

Variables contain the values that measure the same underlying attribute (like age group, outcome, or date of onset). Observations contain all values measured on the same unit (e.g. a person, site, or lab sample). So these aspects can be more difficult to tangibly define.

In “tidy” datasets, each column is a variable, each row is an observation, and each cell is a single value. However some datasets you encounter will not fit this mold - a “wide” format dataset may have a variable split across several columns (see an example in the [Pivoting data](#pivoting-data) page). Likewise, observations could be split across several rows.

Most of this handbook is about managing and transforming data, so referring to the concrete data structures of rows and columns is more relevant than the more abstract observations and variables. Exceptions occur primarily in pages on data analysis, where you will see more references to variables and observations.

### Notes

Here are the types of notes you may encounter in the handbook:

**NOTE:** This is a note  
**TIP:** This is a tip.  
**CAUTION:** This is a cautionary note.  
**DANGER:** This is a warning.

## Editorial decisions

Below, we track significant editorial decisions around package and function choice. If you disagree or want to offer a new tool for consideration, please join/start a conversation on our [Github page](https://github.com/epirhandbook/Epi_R_handbook).

**Table of package, function, and other editorial decisions**

| **Subject** | **Considered** | **Outcome** | **Brief rationale** |
| --- | --- | --- | --- |
| General coding approach | **tidyverse**, **data.table**, **base** | **tidyverse**, with a page on **data.table**, and mentions of **base** alternatives for readers with no internet | **tidyverse** readability, universality, most-taught |
| Package loading | library(),install.packages(), require(), **pacman** | **pacman** | Shortens and simplifies code for most multi-package install/load use-cases |
| Import and export | **rio**, many other packages | **rio** | Ease for many file types |
| Grouping for summary statistics | **dplyr** group\_by(), **stats** aggregate() | **dplyr** group\_by() | Consistent with **tidyverse** emphasis |
| Pivoting | **tidyr** (pivot functions), **reshape2** (melt/cast), **tidyr** (spread/gather) | **tidyr** (pivot functions) | **reshape2** is retired, **tidyr** uses pivot functions as of v1.0.0 |
| Clean column names | **linelist**, **janitor** | **janitor** | Consolidation of packages emphasized |
| Epiweeks | **lubridate**, **aweek**, **tsibble**, **zoo** | **lubridate** generally, the others for specific cases | **lubridate’s** flexibility, consistency, package maintenance prospects |
| ggplot labels | labs(), ggtitle()/ylab()/xlab() | labs() | all labels in one place, simplicity |
| Convert to factor | factor(), **forcats** | **forcats** | its various functions also convert to factor in same command |
| Epidemic curves | **incidence**, **ggplot2**, **EpiCurve** | **incidence2** as quick, **ggplot2** as detailed | dependability |
| Concatenation | paste(), paste0(), str\_glue(), glue() | str\_glue() | More simple syntax than paste functions; within **stringr** |

## Major revisions

| **Date** | **Major changes** |
| --- | --- |
| 10 May 2021 | Release of version 1.0.0 |

## Session info (R, RStudio, packages)

Below is the information on the versions of R, RStudio, and R packages used during this rendering of the Handbook.

# Download handbook and data

## Download offline handbook

You can download the offline version of this handbook as an HTML file so that you can view the file in your web browser even if you no longer have internet access. If you are considering offline use of the Epi R Handbook here are a few things to consider:

* When you open the file it may take a minute or two for the images and the Table of Contents to load
* The offline handbook has a slightly different layout - one very long page with Table of Contents on the left. To search for specific terms use Ctrl+f (Cmd-f)
* See the [Suggested packages](#suggested-packages-1) page to assist you with installing appropriate R packages before you lose internet connectivity
* Install our R package **epirhandbook** that contains all the example data (install process described below)

**There are two ways you can download the handbook:**

### Use download link

For quick access, **right-click** [this link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/offline_long/Epi_R_Handbook_offline.html) **and select “Save link as”**.

If on a Mac, use Cmd+click. If on a mobile, press and hold the link and select “Save link”. The handbook will download to your device. If a screen with raw HTML code appears, ensure you have followed the above instructions or try Option 2.

### Use our R package

We offer an R package called **epirhandbook**. It includes a function download\_book() that downloads the handbook file from our Github repository to your computer.

This package also contains a function get\_data() that downloads all the example data to your computer.

Run the following code to install our R package **epirhandbook** from the [Github repository appliedepi](https://github.com/appliedepi/epirhandbook). This package is not on CRAN, so use the special function p\_install\_gh() to install it from Github.

Now, load the package for use in your current R session:

Next, run the package’s function download\_book() (with empty parentheses) to download the handbook to your computer. Assuming you are in RStudio, a window will appear allowing you to select a save location.

## Download data to follow along

To “follow along” with the handbook pages, you can download the example data and outputs.

### Use our R package

The easiest approach to download all the data is to install our R package **epirhandbook**. It contains a function get\_data() that saves all the example data to a folder of your choice on your computer.

To install our R package **epirhandbook**, run the following code. This package is not on CRAN, so use the function p\_install\_gh() to install it. The input is referencing our Github organisation (“appliedepi”) and the **epirhandbook** package.

Now, load the package for use in your current R session:

Next, use the package’s function get\_data() to download the example data to your computer. Run get\_data("all") to get all the example data, or provide a specific file name and extension within the quotes to retrieve only one file.

The data have already been downloaded with the package, and simply need to be transferred out to a folder on your computer. A pop-up window will appear, allowing you to select a save folder location. We suggest you create a new “data” folder as there are about 30 files (including example data and example outputs).

Once you have used get\_data() to save a file to your computer, you will still need to import it into R. See the [Import and export](#import-and-export) page for details.

If you wish, you can review all the data used in this handbook in the [**“data” folder**](https://github.com/epirhandbook/Epi_R_handbook/tree/master/data) of our Github repository.

### Download one-by-one

This option involves downloading the data file-by-file from our Github repository via either a link or an R command specific to the file. Some file types allow a download button, while others can be downloaded via an R command.

#### Case linelist

This is a fictional Ebola outbreak, expanded by the handbook team from the ebola\_sim practice dataset in the **outbreaks** package.

* [Click to download the “raw” linelist (.xlsx)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_raw.xlsx). The “raw” case linelist is an Excel spreadsheet with messy data. Use this to follow-along with the [Cleaning data and core functions](#cleaning-data-and-core-functions) page.
* [Click to download the “clean” linelist (.rds)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds). Use this file for all other pages of this handbook that use the linelist. A .rds file is an R-specific file type that preserves column classes. This ensures you will have only minimal cleaning to do after importing the data into R.

Other related files:

* [Click to download the “clean” linelist as an Excel file](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.xlsx)
* Part of the cleaning page uses a “cleaning dictionary” (.csv file). You can load it directly into R by running the following commands:

#### Malaria count data

These data are fictional counts of malaria cases by age group, facility, and day. A .rds file is an R-specific file type that preserves column classes. This ensures you will have only minimal cleaning to do after importing the data into R.

[Click to download the malaria count data (.rds file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/malaria_facility_count_data.rds)

#### Likert-scale data

These are fictional data from a Likert-style survey, used in the page on [Demographic pyramids and Likert-scales](#demographic-pyramids-and-likert-scales). You can load these data directly into R by running the following commands:

#### Flexdashboard

Below are links to the file associated with the page on [Dashboards with R Markdown](#dashboards-with-r-markdown):

* To download the R Markdown for the outbreak dashboard, right-click this [link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/flexdashboard/outbreak_dashboard.Rmd) (Cmd+click for Mac) and select “Save link as”.
* To download the HTML dashboard, right-click this [link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/flexdashboard/outbreak_dashboard_test.html) (Cmd+click for Mac) and select “Save link as”.

#### Contact Tracing

The [Contact Tracing](#contact-tracing-1) page demonstrated analysis of contact tracing data, using example data from [Go.Data](https://github.com/WorldHealthOrganization/godata/tree/master/analytics/r-reporting). The data used in the page can be downloaded as .rds files by clicking the following links:

[Click to download the case investigation data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/cases_clean.rds?raw=true)

[Click to download the contact registration data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/contacts_clean.rds?raw=true)

[Click to download the contact follow-up data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/followups_clean.rds?raw=true)

**NOTE:** Structured contact tracing data from other software (e.g. KoBo, DHIS2 Tracker, CommCare) may look different. If you would like to contribute alternative sample data or content for this page, please [contact us](#contact_us).

**TIP:** If you are deploying Go.Data and want to connect to your instance’s API, see the Import and export page [(API section)](#import_api) and the [Go.Data Community of Practice](https://community-godata.who.int/).

#### GIS

Shapefiles have many sub-component files, each with a different file extention. One file will have the “.shp” extension, but others may have “.dbf”, “.prj”, etc.

The [GIS basics](#gis-basics) page provides links to the Humanitarian Data Exchange website where you can download the shapefiles directly as zipped files.

For example, the health facility points data can be downloaded [here](https://data.humdata.org/dataset/hotosm_sierra_leone_health_facilities). Download “hotosm\_sierra\_leone\_health\_facilities\_points\_shp.zip”. Once saved to your computer, “unzip” the folder. You will see several files with different extensions (e.g. “.shp”, “.prj”, “.shx”) - all these must be saved to the same folder on your computer. Then to import into R, provide the file path and name of the “.shp” file to st\_read() from the **sf** package (as described in the [GIS basics](#gis-basics) page).

If you follow Option 1 to download all the example data (via our R package **epirhandbook**), all the shapefiles are included.

Alternatively, you can download the shapefiles from the R Handbook Github “data” folder (see the “gis” sub-folder). However, be aware that you will need to download each sub-file individually to your computer. In Github, click on each file individually and download them by clicking on the “Download” button. Below, you can see how the shapefile “sle\_adm3” consists of many files - each of which would need to be downloaded from Github.

#### Phylogenetic trees

See the page on [Phylogenetic trees](#phylogenetic-trees-1). Newick file of phylogenetic tree constructed from whole genome sequencing of 299 Shigella sonnei samples and corresponding sample data (converted to a text file). The Belgian samples and resulting data are kindly provided by the Belgian NRC for Salmonella and Shigella in the scope of a project conducted by an ECDC EUPHEM Fellow, and will also be published in a manuscript. The international data are openly available on public databases (ncbi) and have been previously published.

* To download the “Shigella\_tree.txt” phylogenetic tree file, right-click this [link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/phylo/Shigella_tree.txt) (Cmd+click for Mac) and select “Save link as”.
* To download the “sample\_data\_Shigella\_tree.csv” with additional information on each sample, right-click this [link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/phylo/sample_data_Shigella_tree.csv) (Cmd+click for Mac) and select “Save link as”.
* To see the new, created subset-tree, right-click this [link](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/phylo/Shigella_subtree_2.txt) (Cmd+click for Mac) and select “Save link as”. The .txt file will download to your computer.

You can then import the .txt files with read.tree() from the **ape** package, as explained in the page.

#### Standardization

See the page on [Standardised rates](#standardised-rates). You can load the data directly from our Github repository on the internet into your R session with the following commands:

#### Time series and outbreak detection

See the page on [Time series and outbreak detection](#time-series-and-outbreak-detection). We use campylobacter cases reported in Germany 2002-2011, as available from the **surveillance** R package. (nb. this dataset has been adapted from the original, in that 3 months of data have been deleted from the end of 2011 for demonstration purposes)

[Click to download Campylobacter in Germany (.xlsx)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/campylobacter_germany.xlsx)

We also use climate data from Germany 2002-2011 (temperature in degrees celsius and rain fail in millimetres) . These were downloaded from the EU Copernicus satellite reanalysis dataset using the **ecmwfr** package. You will need to download all of these and import them with stars::read\_stars() as explained in the time series page.

[Click to download Germany weather 2002 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2002.nc)

[Click to download Germany weather 2003 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2003.nc)

[Click to download Germany weather 2004 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2004.nc)

[Click to download Germany weather 2005 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2005.nc)

[Click to download Germany weather 2006 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2006.nc)

[Click to download Germany weather 2007 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2007.nc)

[Click to download Germany weather 2008 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2008.nc)

[Click to download Germany weather 2009 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2009.nc)

[Click to download Germany weather 2010 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2010.nc)

[Click to download Germany weather 2011 (.nc file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/weather/germany_weather2011.nc)

#### Survey analysis

For the [survey analysis](https://epirhandbook.com/survey-analysis.html) page we use fictional mortality survey data based off MSF OCA survey templates. This fictional data was generated as part of the [“R4Epis” project](https://r4epis.netlify.app/).

[Click to download Fictional survey data (.xlsx)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/surveys/survey_data.xlsx)

[Click to download Fictional survey data dictionary (.xlsx)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/surveys/survey_dict.xlsx)

[Click to download Fictional survey population data (.xlsx)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/surveys/population.xlsx)

#### Shiny

The page on [Dashboards with Shiny](#dashboards-with-shiny) demonstrates the construction of a simple app to display malaria data.

To download the R files that produce the Shiny app:

You can [click here to download the app.R file that contains both the UI and Server code for the Shiny app.](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/malaria_app/app.R)

You can [click here to download the facility\_count\_data.rds file](https://github.com/epirhandbook/Epi_R_handbook/blob/master/data/malaria_app/data/facility_count_data.rds) that contains malaria data for the Shiny app. Note that you may need to store it within a “data” folder for the here() file paths to work correctly.

You can [click here to download the global.R file](https://github.com/epirhandbook/Epi_R_handbook/blob/master/data/malaria_app/global.R) that should run prior to the app opening, as explained in the page.

You can [click here to download the plot\_epicurve.R file](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/malaria_app/funcs/plot_epicurve.R) that is sourced by global.R. Note that you may need to store it within a “funcs” folder for the here() file paths to work correctly.

# II Basics

# R Basics

Welcome!

This page reviews the essentials of R. It is not intended to be a comprehensive tutorial, but it provides the basics and can be useful for refreshing your memory. The section on [Resource for learning](#learning) links to more comprehensive tutorials.

Parts of this page have been adapted with permission from the [R4Epis project](https://r4epis.netlify.app/).

See the page on [Transition to R](#transition-to-r) for tips on switching to R from STATA, SAS, or Excel.

## Why use R?

As stated on the [R project website](https://www.r-project.org/about.html), R is a programming language and environment for statistical computing and graphics. It is highly versatile, extendable, and community-driven.

**Cost**

R is free to use! There is a strong ethic in the community of free and open-source material.

**Reproducibility**

Conducting your data management and analysis through a programming language (compared to Excel or another primarily point-click/manual tool) enhances **reproducibility**, makes **error-detection** easier, and eases your workload.

**Community**

The R community of users is enormous and collaborative. New packages and tools to address real-life problems are developed daily, and vetted by the community of users. As one example, [R-Ladies](https://rladies.org/) is a worldwide organization whose mission is to promote gender diversity in the R community, and is one of the largest organizations of R users. It likely has a chapter near you!

## Key terms

**RStudio** - RStudio is a Graphical User Interface (GUI) for easier use of **R**. Read more [in the RStudio section](#rstudio).

**Objects** - 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. Read more [in the Objects section](#objects).

**Functions** - A function is a code operation that accept inputs and returns a transformed output. Read more [in the Functions section](#functions).

**Packages** - An R package is a shareable bundle of functions. Read more [in the Packages section](#packages).

**Scripts** - A script is the document file that hold your commands. Read more [in the Scripts section](#scripts)

## Resources for learning

### Resources within RStudio

**Help documentation**

Search the RStudio “Help” tab for documentation on R packages and specific functions. This is within the pane that also contains Files, Plots, and Packages (typically in the lower-right pane). As a shortcut, you can also type the name of a package or function into the R console after a question-mark to open the relevant Help page. Do not include parentheses.

For example: ?filter or ?diagrammeR.

**Interactive tutorials**

There are several ways to learn R interactively within RStudio.

RStudio itself offers a Tutorial pane that is powered by the [**learnr**](https://blog.rstudio.com/2020/02/25/rstudio-1-3-integrated-tutorials/) R package. Simply install this package and open a tutorial via the new “Tutorial” tab in the upper-right RStudio pane (which also contains Environment and History tabs).

The R package [**swirl**](https://swirlstats.com/) offers interactive courses in the R Console. Install and load this package, then run the command swirl() (empty parentheses) in the R console. You will see prompts appear in the Console. Respond by typing in the Console. It will guide you through a course of your choice.

### Cheatsheets

There are many PDF “cheatsheets” available on the [RStudio website](https://rstudio.com/resources/cheatsheets/), for example:

* Factors with **forcats** package
* Dates and times with **lubridate** package
* Strings with **stringr** package
* iterative opertaions with **purrr** package
* Data import
* Data transformation cheatsheet with **dplyr** package
* R Markdown (to create documents like PDF, Word, Powerpoint…)
* Shiny (to build interactive web apps)
* Data visualization with **ggplot2** package
* Cartography (GIS)
* **leaflet** package (interactive maps)
* Python with R (**reticulate** package)

This is an online R resource specifically for [Excel users](https://jules32.github.io/r-for-excel-users/)

### Twitter

R has a vibrant twitter community where you can learn tips, shortcuts, and news - follow these accounts:

* Follow us! [@epiRhandbook](https://twitter.com/epirhandbook)
* R Function A Day [@rfuntionaday](https://twitter.com/rfunctionaday) is an incredible resource
* R for Data Science [@rstats4ds](https://twitter.com/rstats4ds?lang=en)
* RStudio [@RStudio](https://twitter.com/rstudio?lang=en)
* RStudio Tips [@rstudiotips](https://twitter.com/rstudiotips)
* R-Bloggers [@Rbloggers](https://twitter.com/Rbloggers)
* R-ladies [@RLadiesGlobal](https://twitter.com/RLadiesGlobal)
* Hadley Wickham [@hadleywickham](https://twitter.com/hadleywickham?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor)

Also:

**#epitwitter** and **#rstats**

### Free online resources

A definitive text is the [R for Data Science](https://r4ds.had.co.nz/) book by Garrett Grolemund and Hadley Wickham

The [R4Epis](https://r4epis.netlify.app/) project website aims to “develop standardised data cleaning, analysis and reporting tools to cover common types of outbreaks and population-based surveys that would be conducted in an MSF emergency response setting.” You can find R basics training materials, templates for RMarkdown reports on outbreaks and surveys, and tutorials to help you set them up.

### Languages other than English

[Materiales de RStudio en Español](https://www.rstudio.com/collections/espanol/)

[Introduction à R et au tidyverse (Francais)](https://juba.github.io/tidyverse/index.html)

## Installation

### R and RStudio

**How to install R**

Visit this website <https://www.r-project.org/> and download the latest version of R suitable for your computer.

**How to install RStudio**

Visit this website <https://rstudio.com/products/rstudio/download/> and download the latest free Desktop version of RStudio suitable for your computer.

**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](#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. Alternatively, you can use the **installr** package (on Windows) by running installr::updateR(). This will open dialog boxes to help you download the latest R version and update your packages to the new R version. More details can be found in the **installr** [documentation](https://www.r-project.org/nosvn/pandoc/installr.html).

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.

To see which versions of R, RStudio, or packages were used when this Handbook as made, see the page on [Editorial and technical notes](#editorial-and-technical-notes).

### Other software you may need to install

* TinyTeX (for compiling an RMarkdown document to PDF)
* Pandoc (for compiling RMarkdown documents)
* RTools (for building packages for R)
* phantomjs (for saving still images of animated networks, such as transmission chains)

#### TinyTex

TinyTex is a custom LaTeX distribution, useful when trying to produce PDFs from R.  
See <https://yihui.org/tinytex/> for more informaton.

To install TinyTex from R:

#### Pandoc

Pandoc is a document converter, a separate software from R. **It comes bundled with RStudio and should not need to be downloaded.** It helps the process of converting Rmarkdown documents to formats like .pdf and adding complex functionality.

#### RTools

RTools is a collection of software for building packages for R

Install from this website: <https://cran.r-project.org/bin/windows/Rtools/>

#### phantomjs

This is often used to take “screenshots” of webpages. For example when you make a transmission chain with **epicontacts** package, an HTML file is produced that is interactive and dynamic. If you want a static image, it can be useful to use the [**webshot**](https://wch.github.io/webshot/articles/intro.html) package to automate this process. This will require the external program “phantomjs”. You can install phantomjs via the **webshot** package with the command webshot::install\_phantomjs().

## RStudio

### RStudio orientation

**First, open RStudio.** As their icons can look very similar, be sure you are opening RStudio and not R.

For RStudio to work you must also have R installed on the computer (see above for installation instructions).

**RStudio** is an interface (GUI) for easier use of **R**. You can think of R as being the engine of a vehicle, doing the crucial work, and RStudio as the body of the vehicle (with seats, accessories, etc.) that helps you actually use the engine to move forward! You can see the complete RStudio user-interface cheatsheet (PDF) [here](https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf)

By default RStudio displays four rectangle panes.

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

**The Source Pane**  
This pane, by default in the upper-left, is a space to edit, run, and save your [scripts](#scripts). Scripts contain the commands you want to run. This pane can also display datasets (data frames) for viewing.

For Stata users, this pane is similar to your Do-file and Data Editor windows.

**The R Console Pane**

The R Console, by default the left or lower-left pane in R Studio, is the home of the R “engine”. This is where the commands are actually run and non-graphic outputs and error/warning messages appear. You can directly enter and run commands in the R Console, but realize that these commands are not saved as they are when running commands from a script.

If you are familiar with Stata, the R Console is like the Command Window and also the Results Window.

**The Environment Pane**  
This pane, by default in the upper-right, is most often used to see brief summaries of [objects](#objects) in the R Environment in the current session. These objects could include imported, modified, or created datasets, parameters you have defined (e.g. a specific epi week for the analysis), or vectors or lists you have defined during analysis (e.g. names of regions). You can click on the arrow next to a data frame name to see its variables.

In Stata, this is most similar to the Variables Manager window.

This pane also contains History where can see commands that you can previously. It also has a “Tutorial” tab where you can complete interactive R tutorials if you have the **learnr** package installed. It also has a “Connections” pane for external connections, and can have a “Git” pane if you choose to interface with Github.

**Plots, Viewer, Packages, and Help Pane**  
The lower-right pane includes several important tabs. Typical plot graphics including maps will display in the Plot pane. Interactive or HTML outputs will display in the Viewer pane. The Help pane can display documentation and help files. The Files pane is a browser which can be used to open or delete files. The Packages pane allows you to see, install, update, delete, load/unload R packages, and see which version of the package you have. To learn more about packages see the [packages section](#packages) below.

This pane contains the Stata equivalents of the Plots Manager and Project Manager windows.

### RStudio settings

Change RStudio settings and appearance in the Tools drop-down menu, by selecting Global Options. There you can change the default settings, including appearance/background color.

**Restart**

If your R freezes, you can re-start R by going to the Session menu and clicking “Restart R”. This avoids the hassle of closing and opening RStudio. Everything in your R environment will be removed when you do this.

### Keyboard shortcuts

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](https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf).

| **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 |

**TIP:** 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.

## 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](#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 handbook, functions are written in code-text with open parentheses, like this: filter(). As explained in the [packages](#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:

The object provided to a function also can be a column in a dataset (see the [Objects](#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.

**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.

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.

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.

**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](#objectstructure) 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.

### Writing Functions

R is a language that is oriented around functions, so you should feel empowered to write your own functions. Creating functions brings several advantages:

* To facilitate modular programming - the separation of code in to independent and manageable pieces
* Replace repetitive copy-and-paste, which can be error prone
* Give pieces of code memorable names

How to write a function is covered in-depth in the [Writing functions](#writing-functions-1) page.

## 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). This becomes especially important if working with [R on network drives](#r-on-network-drives).

#### 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.

Are you worried about viruses and security when downloading a package from CRAN? Read [this article](https://support.rstudio.com/hc/en-us/articles/360042593974-R-and-R-Package-Security) on the topic.

#### How to install and load

In this handbook, we suggest using the **pacman** package (short for “package manager”). It offers a convenient function p\_load() which will install a package if necessary and load it for use in the current R session.

The syntax quite simple. Just list the names of the packages within the p\_load() parentheses, separated by commas. This command will install the **rio**, **tidyverse**, and **here** packages if they are not yet installed, and will load them for use. This makes the p\_load() approach convenient and concise if sharing scripts with others. Note that package names are case-sensitive.

Note that we have used the syntax pacman::p\_load() which explicitly writes the package name (**pacman**) prior to the function name (p\_load()), connected by two colons ::. This syntax is useful because it also loads the **pacman** package (assuming it is already installed).

There are alternative **base** R functions that you will see often. 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().

Note: this command installs a package, but does not load it for use in the current session.

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.

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.

**Install from Github**

Sometimes, you need to install a package that is not yet available from CRAN. Or perhaps the package is available on CRAN but you want the development version with new features not yet offered in the more stable published CRAN version. These are often hosted on the website [github.com](https://github.com/) in a free, public-facing code “repository”. Read more about Github in the handbook page on [Version control and collaboration with Git and Github](#version-control-and-collaboration-with-).

To download R packages from Github, you can use the function p\_load\_gh() from **pacman**, which will install the package if necessary, and load it for use in your current R session. Alternatives to install include using the **remotes** or **devtools** packages. Read more about all the **pacman** functions in the [package documentation](https://cran.r-project.org/web/packages/pacman/pacman.pdf).

To install from Github, you have to provide more information. You must provide:

1. The Github ID of the repository owner
2. The name of the repository that contains the package
3. (optional) The name of the “branch” (specific development version) you want to download

In the examples below, the first word in the quotation marks is the Github ID of the repository owner, after the slash is the name of the repository (the name of the package).

If you want to install from a “branch” (version) other than the main branch, add the branch name after an “@”, after the repository name.

If there is no difference between the Github version and the version on your computer, no action will be taken. You can “force” a re-install by instead using p\_load\_current\_gh() with the argument update = TRUE. Read more about **pacman** in this [online vignette](http://trinker.github.io/pacman/vignettes/Introduction_to_pacman.html)

**Install from ZIP or TAR**

You could install the package from a URL:

Or, download it to your computer in a zipped file:

Option 1: using install\_local() from the **remotes** package

Option 2: using install.packages() from **base** R, providing the file path to the ZIP file and setting type = "source and repos = NULL.

### 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.

### 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 to 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 p\_delete() from **pacman**, or remove.packages() from **base** R. 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.

See the dependencies of a package with p\_depends(), and see which packages depend on it with p\_depends\_reverse()

### 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 (e.g. within p\_load()), and **start a new R session**

### Detach / unload

To detach (unload) a package, use this command, with the correct package name and only one colon. Note that this may not resolve masking.

### Install older version

See this [guide](https://support.rstudio.com/hc/en-us/articles/219949047-Installing-older-versions-of-packages) to install an older version of a particular package.

### Suggested packages

See the page on [Suggested packages](#suggested-packages-1) for a listing of packages we recommend for everyday epidemiology.

## 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).

* 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. We advocate for the **tidyverse** [style guide](https://style.tidyverse.org/). There are also packages such as **styler** and **lintr** which help you conform to this 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 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](#reports-with-r-markdown) documents.

### R notebooks

There is no difference between writing in a Rmarkdown vs an R notebook. However the execution of the document differs slightly. See this [site](http://uc-r.github.io/r_notebook) for more details.

### Shiny

Shiny apps/websites are contained within one script, which must be named app.R. This file has three components:

1. A user interface (ui)
2. A server function
3. A call to the shinyApp function

See the handbook page on [Dashboards with Shiny](#dashboards-with-shiny), or this online tutorial: [Shiny tutorial](https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/)

In older times, the above file was split into two files (*ui.R* and *server.R*)

### Code folding

You can collapse portions of code to make your script easier to read.

To do this, create a text header with #, write your header, and follow it with at least 4 of either dashes (-), hashes (#) or equals (=). When you have done this, a small arrow will appear in the “gutter” to the left (by the row number). You can click this arrow and the code below until the next header will collapse and a dual-arrow icon will appear in its place.

To expand the code, either click the arrow in the gutter again, or the dual-arrow icon. There are also keyboard shortcuts as explained in the [RStudio section](#rstudio) of this page.

By creating headers with #, you will also activate the Table of Contents at the bottom of your script (see below) that you can use to navigate your script. You can create sub-headers by adding more # symbols, for example # for primary, ## for seconary, and ### for tertiary headers.

Below are two versions of an example script. On the left is the original with commented headers. On the right, four dashes have been written after each header, making them collapsible. Two of them have been collapsed, and you can see that the Table of Contents at the bottom now shows each section.

Other areas of code that are automatically eligible for folding include “braced” regions with brackets { } such as function definitions or conditional blocks (if else statements). You can read more about code folding at the RStudio [site](https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections).

## Working directory

The working directory is the root folder location used by R for your work - where R looks for and saves files by default. By default, it will save new files and outputs to this location, and will look for files to import (e.g. datasets) here as well.

The working directory appears in grey text at the top of the RStudio Console pane. You can also print the current working directory by running getwd() (leave the parentheses empty).

### Recommended approach

**See the page on** [**R projects**](#r-projects) **for details on our recommended approach to managing your working directory.**  
A common, efficient, and trouble-free way to manage your working directory and file paths is to combine these 3 elements in an [R project](#r-projects)-oriented workflow:

1. An R Project to store all your files (see page on [R projects](#r-projects))
2. The **here** package to locate files (see page on [Import and export](#import-and-export))
3. The **rio** package to import/export files (see page on [Import and export](#import-and-export))

### Set by command

Until recently, many people learning R were taught to begin their scripts with a setwd() command. Please instead consider using an [R project](#r-projects)-oriented workflow and read the [reasons for not using setwd()](https://www.tidyverse.org/blog/2017/12/workflow-vs-script/). In brief, your work becomes specific to your computer, file paths used to import and export files become “brittle”, and this severely hinders collaboration and use of your code on any other computer. There are easy alternatives!

As noted above, although we do not recommend this approach in most circumstances, you can use the command setwd() with the desired folder file path in quotations, for example:

**DANGER:** Setting a working directory with setwd() can be “brittle” if the file path is specific to one computer. Instead, use file paths relative to an R Project root directory (with the **here** package).

### Set manually

To set the working directory manually (the point-and-click equivalent of setwd()), click the Session drop-down menu and go to “Set Working Directory” and then “Choose Directory”. This will set the working directory for that specific R session. Note: if using this approach, you will have to do this manually each time you open RStudio.

### Within an R project

If using an R project, the working directory will default to the R project root folder that contains the “.rproj” file. This will apply if you open RStudio by clicking open the R Project (the file with “.rproj” extension).

### Working directory in an R markdown

In an R markdown script, the default working directory is the folder the Rmarkdown file (.Rmd) is saved within. If using an R project and **here** package, this does not apply and the working directory will be here() as explained in the [R projects](#r-projects) page.

If you want to change the working directory of a stand-alone R markdown (not in an R project), if you use setwd() this will only apply to that specific code chunk. To make the change for all code chunks in an R markdown, edit the setup chunk to add the root.dir = parameter, such as below:

It is much easier to just use the R markdown within an R project and use the **here** package.

### Providing file paths

Perhaps the most common source of frustration for an R beginner (at least on a Windows machine) is typing in a file path to import or export data. There is a thorough explanation of how to best input file paths in the [Import and export](#import-and-export) page, but here are a few key points:

**Broken paths**

Below is an example of an “absolute” or “full address” file path. These will likely break if used by another computer. One exception is if you are using a shared/network drive.

C:/Users/Name/Document/Analytic Software/R/Projects/Analysis2019/data/March2019.csv

**Slash direction**

If typing in a file path, be aware the direction of the slashes. Use forward slashes (/) to separate the components (“data/provincial.csv”). For Windows users, the default way that file paths are displayed is with back slashes (\) - so you will need to change the direction of each slash. If you use the **here** package as described in the [R projects](#r-projects) page the slash direction is not an issue.

**Relative file paths**

We generally recommend providing “relative” filepaths instead - that is, the path relative to the root of your R Project. You can do this using the **here** package as explained in the [R projects](#r-projects) page. A relativel filepath might look like this:

Even if using relative file paths within an R project, you can still use absolute paths to import/export data outside your R project.

## 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

This section is adapted from the [*R4Epis project*](https://r4epis.netlify.app/training/r_basics/objects/).  
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.

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

**CAUTION:** **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**.

The following command will re-define the value of current\_week:

**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](https://renkun.me/2014/01/28/difference-between-assignment-operators-in-r/).

**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.

You can read more about importing and exporting datasets with the section on [Import and export](#import-and-export).

**CAUTION:** A quick note on naming of 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).

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

**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.

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](#cleaning-data-and-core-functions). You can read more about piping in the Piping section below.

### 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](http://venus.ifca.unican.es/Rintro/dataStruct.html). It shows some common data structures and their names. Not included in this image is spatial data, which is discussed in the [GIS basics](#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](#working-with-dates-1) 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 |
| 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()**. Note: 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

class(linelist$age) # class should be numeric

class(linelist$gender) # class should be 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.

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 - Note: see section on [dates](#dates) for details |
| factor() | Converts to factor - Note: re-defining order of value levels requires extra arguments |

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](https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/).

### 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:

**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”.

By typing the name of the dataframe followed by $ you will also see a drop-down menu of all columns in the data frame. You can scroll through them using your arrow key, select one with your Enter key, and avoid spelling mistakes!

**ADVANCED TIP:** Some more complex objects (e.g. a list, or an epicontacts object) may have multiple levels which can be accessed through multiple dollar signs. For example epicontacts$linelist$date\_onset

### 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.

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

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

Note that you can also achieve the above row/column indexing on data frames and tibbles using **dplyr** syntax (functions filter() for rows, and select() for columns). Read more about these core functions in the [Cleaning data and core functions](#cleaning-data-and-core-functions) page.

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().

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](https://r4ds.had.co.nz/vectors.html#lists-of-condiments) is humorous and helpful.

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

Now we extract, using various methods:

### Remove objects

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

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

## 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](https://style.tidyverse.org/pipes.html)

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

Here is another [link](https://cfss.uchicago.edu/notes/pipes/#:~:text=Pipes%20are%20an%20extremely%20useful,code%20and%20combine%20multiple%20operations) 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](https://magrittr.tidyverse.org/).

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:

**%<>%**  
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:

### 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:

Combine all functions together - this is difficult to read:

## Key operators and functions

This section details operators in R, such as:

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

### 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.

**<<-**

If [Writing functions](#writing-functions-1), or using R in an interactive way with sourced scripts, then you may need to use this assignment operator <<- (from **base** R). This operator is used to define an object in a higher ‘parent’ R Environment. See this [online reference](https://stat.ethz.ch/R-manual/R-devel/library/base/html/assignOps.html).

**%<>%**

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, as shown below in two equivalent examples:

The above is equivalent to the below:

**%<+%**

This is used to add data to phylogenetic trees with the **ggtree** package. See the page on [Phylogenetic trees](#phylogenetic-trees-1) or this online [resource book](https://yulab-smu.top/treedata-book/).

### 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](#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:

| **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). If you import data that records missing data in another way (e.g. 99, “Missing”, or .), you may want to re-code those values to NA. How to do this is addressed in the [Import and export](#import-and-export) page.

**To test whether a value is NA, use the special function is.na()**, which returns TRUE or FALSE.

Read more about missing, infinite, NULL, and impossible values in the page on [Missing data](#missing-data). Learn how to convert missing values when importing data in the page on [Import and export](#import-and-export).

### 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) |
| 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.

#### Scientific notation

The likelihood of scientific notation being used depends on the value of the scipen option.

From the documentation of ?options: scipen is a penalty to be applied when deciding to print numeric values in fixed or exponential notation. Positive values bias towards fixed and negative towards scientific notation: fixed notation will be preferred unless it is more than ‘scipen’ digits wider.

If it is set to a low number (e.g. 0) it will be “turned on” always. To “turn off” scientific notation in your R session, set it to a very high number, for example:

#### Rounding

**DANGER:** round() uses “banker’s rounding” which rounds up from a .5 only if the upper number is even. Use round\_half\_up() from **janitor** to consistently round halves up to the nearest whole number. See [this explanation](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html#explore-records-with-duplicated-values-for-specific-combinations-of-variables-with-get_dupes)

#### Statistical functions

**CAUTION:** 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

**DANGER:** If providing a vector of numbers to one of the above functions, be sure to wrap the numbers within c() .

#### 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.

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

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

Note: If you want to detect a partial string, perhaps using str\_detect() from **stringr**, it will not accept a character vector like c("1", "Yes", "yes", "y"). Instead, it must be given a regular expression - one condensed string with OR bars, such as “1|Yes|yes|y”. For example, str\_detect(hospitalized, "1|Yes|yes|y"). See the page on [Characters and strings](#characters-and-strings) for more information.

You can convert a character vector to a named regular expression with this command:

## Errors & warnings

This section explains:

* The difference between errors and warnings
* General syntax tips for writing R code
* Code assists

Common errors and warnings and troubleshooting tips can be found in the page on [Errors and help].

### 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
* R is case-sensitive, meaning Variable\_A is different from variable\_A

### 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.

# Transition to R

Below, we provide some advice and resources if you are transitioning to R.

R was introduced in the late 1990s and has since grown dramatically in scope. Its capabilities are so extensive that commercial alternatives have reacted to R developments in order to stay competitive! ([read this article comparing R, SPSS, SAS, STATA, and Python](https://www.inwt-statistics.com/read-blog/comparison-of-r-python-sas-spss-and-stata.html)).

Moreover, R is much easier to learn than it was 10 years ago. Previously, R had a reputation of being difficult for beginners. It is now much easier with friendly user-interfaces like RStudio, intuitive code like the **tidyverse**, and many tutorial resources.

**Do not be intimidated - come discover the world of R!**

## From Excel

Transitioning from Excel directly to R is a very achievable goal. It may seem daunting, but you can do it!

It is true that someone with strong Excel skills can do very advanced activities in Excel alone - even using scripting tools like VBA. Excel is used across the world and is an essential tool for an epidemiologist. However, complementing it with R can dramatically improve and expand your work flows.

### Benefits

You will find that using R offers immense benefits in time saved, more consistent and accurate analysis, reproducibility, shareability, and faster error-correction. Like any new software there is a learning “curve” of time you must invest to become familiar. The dividends will be significant and immense scope of new possibilities will open to you with R.

Excel is a well-known software that can be easy for a beginner to use to produce simple analysis and visualizations with “point-and-click”. In comparison, it can take a couple weeks to become comfortable with R functions and interface. However, R has evolved in recent years to become much more friendly to beginners.

Many Excel workflows rely on memory and on repetition - thus, there is much opportunity for error. Furthermore, generally the data cleaning, analysis methodology, and equations used are hidden from view. It can require substantial time for a new colleague to learn what an Excel workbook is doing and how to troubleshoot it. With R, all the steps are explicitly written in the script and can be easily viewed, edited, corrected, and applied to other datasets.

**To begin your transition from Excel to R you must adjust your mindset in a few important ways:**

### Tidy data

Use machine-readable “tidy” data instead of messy “human-readable” data. These are the three main requirements for “tidy” data, as explained in this tutorial on [“tidy” data in R](https://r4ds.had.co.nz/tidy-data.html):

* Each variable must have its own column
* Each observation must have its own row
* Each value must have its own cell

To Excel users - think of the role that [Excel “tables”](https://exceljet.net/excel-tables) play in standardizing data and making the format more predictable.

An example of “tidy” data would be the case linelist used throughout this handbook - each variable is contained within one column, each observation (one case) has it’s own row, and every value is in just one cell. Below you can view the first 50 rows of the linelist:

The main reason one encounters non-tidy data is because many Excel spreadsheets are designed to prioritize easy reading by humans, not easy reading by machines/software.

To help you see the difference, below are some fictional examples of **non-tidy data** that prioritize human-readability over machine-readability:

Problems: In the spreadsheet above, there are merged cells which are not easily digested by R. Which row should be considered the “header” is not clear. A color-based dictionary is to the right side and cell values are represented by colors - which is also not easily interpreted by R (nor by humans with color-blindness!). Furthermore, different pieces of information are combined into one cell (multiple partner organizations working in one area, or the status “TBC” in the same cell as “Partner D”).

Problems: In the spreadsheet above, there are numerous extra empty rows and columns within the dataset - this will cause cleaning headaches in R. Furthermore, the GPS coordinates are spread across two rows for a given treatment center. As a side note - the GPS coordinates are in two different formats!

“Tidy” datasets may not be as readable to a human eye, but they make data cleaning and analysis much easier! Tidy data can be stored in various formats, for example “long” or “wide”"(see page on [Pivoting data](#pivoting-data)), but the principles above are still observed.

### Functions

The R word “function” might be new, but the concept exists in Excel too as formulas. Formulas in Excel also require precise syntax (e.g. placement of semicolons and parentheses). All you need to do is learn a few new functions and how they work together in R.

### Scripts

Instead of clicking buttons and dragging cells you will be writing every step and procedure into a “script”. Excel users may be familiar with “VBA macros” which also employ a scripting approach.

The R script consists of step-by-step instructions. This allows any colleague to read the script and easily see the steps you took. This also helps de-bug errors or inaccurate calculations. See the [R basics](#r-basics) section on scripts for examples.

Here is an example of an R script:

### Excel-to-R resources

Here are some links to tutorials to help you transition to R from Excel:

* [R vs. Excel](https://www.northeastern.edu/graduate/blog/r-vs-excel/)
* [RStudio course in R for Excel users](https://rstudio-conf-2020.github.io/r-for-excel/)

### R-Excel interaction

R has robust ways to import Excel workbooks, work with the data, export/save Excel files, and work with the nuances of Excel sheets.

It is true that some of the more aesthetic Excel formatting can get lost in translation (e.g. italics, sideways text, etc.). If your work flow requires passing documents back-and-forth between R and Excel while retaining the original Excel formatting, try packages such as **openxlsx**.

## From Stata

**Coming to R from Stata**

Many epidemiologists are first taught how to use Stata, and it can seem daunting to move into R. However, if you are a comfortable Stata user then the jump into R is certainly more manageable than you might think. While there are some key differences between Stata and R in how data can be created and modified, as well as how analysis functions are implemented – after learning these key differences you will be able to translate your skills.

Below are some key translations between Stata and R, which may be handy as your review this guide.

**General notes**

| **STATA** | **R** |
| --- | --- |
| You can only view and manipulate one dataset at a time | You can view and manipulate multiple datasets at the same time, therefore you will frequently have to specify your dataset within the code |
| Online community available through <https://www.statalist.org/> | Online community available through [RStudio](https://community.rstudio.com/), [StackOverFlow](https://stackoverflow.com/questions/tagged/r), and [R-bloggers](https://www.r-bloggers.com/) |
| Point and click functionality as an option | Minimal point and click functionality |
| Help for commands available by help [command] | Help available by [function]? or search in the Help pane |
| Comment code using \* or /// or /\* TEXT \*/ | Comment code using # |
| Almost all commands are built-in to Stata. New/user-written functions can be installed as **ado** files using **ssc install** [package] | R installs with **base** functions, but typical use involves installing other packages from CRAN (see page on [R basics](#r-basics)) |
| Analysis is usually written in a **do** file | Analysis written in an R script in the RStudio source pane. R markdown scripts are an alternative. |

**Working directory**

| **STATA** | **R** |
| --- | --- |
| Working directories involve absolute filepaths (e.g. “C:/usename/documents/projects/data/”) | Working directories can be either absolute, or relative to a project root folder by using the **here** package (see [Import and export](#import-and-export)) |
| See current working directory with **pwd** | Use getwd() or here() (if using the **here** package), with empty parentheses |
| Set working directory with **cd** “folder location” | Use setwd(“folder location”), or set\_here("folder location) (if using **here** package) |

**Importing and viewing data**

| **STATA** | **R** |
| --- | --- |
| Specific commands per file type | Use import() from **rio** package for almost all filetypes. Specific functions exist as alternatives (see [Import and export](#import-and-export)) |
| Reading in csv files is done by **import delimited** “filename.csv” | Use import("filename.csv") |
| Reading in xslx files is done by **import excel** “filename.xlsx” | Use import("filename.xlsx") |
| Browse your data in a new window using the command **browse** | View a dataset in the RStudio source pane using View(dataset). You need to specify your dataset name to the function in R because multiple datasets can be held at the same time. Note capital “V” in this function |
| Get a high-level overview of your dataset using **summarize**, which provides the variable names and basic information | Get a high-level overview of your dataset using summary(dataset) |

**Basic data manipulation**

| **STATA** | **R** |
| --- | --- |
| Dataset columns are often referred to as “variables” | More often referred to as “columns” or sometimes as “vectors” or “variables” |
| No need to specify the dataset | In each of the below commands, you need to specify the dataset - see the page on [Cleaning data and core functions](#cleaning-data-and-core-functions) for examples |
| New variables are created using the command **generate** varname = | Generate new variables using the function mutate(varname = ). See page on [Cleaning data and core functions](#cleaning-data-and-core-functions) for details on all the below **dplyr** functions. |
| Variables are renamed using **rename** old\_name new\_name | Columns can be renamed using the function rename(new\_name = old\_name) |
| Variables are dropped using **drop** varname | Columns can be removed using the function select() with the column name in the parentheses following a minus sign |
| Factor variables can be labeled using a series of commands such as **label define** | Labeling values can done by converting the column to Factor class and specifying levels. See page on [Factors](#factors). Column names are not typically labeled as they are in Stata. |

**Descriptive analysis**

| **STATA** | **R** |
| --- | --- |
| Tabulate counts of a variable using **tab** varname | Provide the dataset and column name to table() such as table(dataset$colname). Alternatively, use count(varname) from the **dplyr** package, as explained in [Grouping data](#grouping-data) |
| Cross-tabulaton of two variables in a 2x2 table is done with **tab** varname1 varname2 | Use table(dataset$varname1, dataset$varname2 or count(varname1, varname2) |

While this list gives an overview of the basics in translating Stata commands into R, it is not exhaustive. There are many other great resources for Stata users transitioning to R that could be of interest:

* <https://dss.princeton.edu/training/RStata.pdf>
* <https://clanfear.github.io/Stata_R_Equivalency/docs/r_stata_commands.html>
* <http://r4stats.com/books/r4stata/>

## From SAS

**Coming from SAS to R**

SAS is commonly used at public health agencies and academic research fields. Although transitioning to a new language is rarely a simple process, understanding key differences between SAS and R may help you start to navigate the new language using your native language. Below outlines the key translations in data management and descriptive analysis between SAS and R.

**General notes**

| **SAS** | **R** |
| --- | --- |
| Online community available through [SAS Customer Support](https://support.sas.com/en/support-home.html) | Online community available through RStudio, StackOverFlow, and R-bloggers |
| Help for commands available by help [command] | Help available by [function]? or search in the Help pane |
| Comment code using \* TEXT ; or /\* TEXT \*/ | Comment code using # |
| Almost all commands are built-in. Users can write new functions using SAS macro, SAS/IML, SAS Component Language (SCL), and most recently, procedures Proc Fcmp and Proc Proto | R installs with **base** functions, but typical use involves installing other packages from CRAN (see page on [R basics](#r-basics)) |
| Analysis is usually conducted by writing a SAS program in the Editor window. | Analysis written in an R script in the RStudio source pane. R markdown scripts are an alternative. |

**Working directory**

| **SAS** | **R** |
| --- | --- |
| Working directories can be either absolute, or relative to a project root folder by defining the root folder using %let rootdir=/root path; %include “&rootdir/subfoldername/filename” | Working directories can be either absolute, or relative to a project root folder by using the **here** package (see [Import and export](#import-and-export)) |
| See current working directory with %put %sysfunc(getoption(work)); | Use getwd() or here() (if using the **here** package), with empty parentheses |
| Set working directory with libname “folder location” | Use setwd(“folder location”), or set\_here("folder location) if using **here** package |

**Importing and viewing data**

| **SAS** | **R** |
| --- | --- |
| Use Proc Import procedure or using Data Step Infile statement. | Use import() from **rio** package for almost all filetypes. Specific functions exist as alternatives (see [Import and export](#import-and-export)) |
| Reading in csv files is done by using Proc Import datafile=”filename.csv” out=work.filename dbms=CSV; run; OR using [Data Step Infile statement](http://support.sas.com/techsup/technote/ts673.pdf) | Use import("filename.csv") |

Reading in xslx files is done by using Proc Import datafile=”filename.xlsx” out=work.filename dbms=xlsx; run; OR using [Data Step Infile statement](http://support.sas.com/techsup/technote/ts673.pdf)|Use import(“filename.xlsx”) Browse your data in a new window by opening the Explorer window and select desired library and the dataset|View a dataset in the RStudio source pane using View(dataset). You need to specify your dataset name to the function in R because multiple datasets can be held at the same time. Note capital “V” in this function

**Basic data manipulation**

| **SAS** | **R** |
| --- | --- |
| Dataset columns are often referred to as “variables” | More often referred to as “columns” or sometimes as “vectors” or “variables” |
| No special procedures are needed to create a variable. New variables are created simply by typing the new variable name, followed by an equal sign, and then an expression for the value | Generate new variables using the function mutate(). See page on [Cleaning data and core functions](#cleaning-data-and-core-functions) for details on all the below **dplyr** functions. |
| Variables are renamed using rename \*old\_name=new\_name\* | Columns can be renamed using the function rename(new\_name = old\_name) |
| Variables are kept using \*\*keep\*\*=varname | Columns can be selected using the function select() with the column name in the parentheses |
| Variables are dropped using \*\*drop\*\*=varname | Columns can be removed using the function select() with the column name in the parentheses following a minus sign |
| Factor variables can be labeled in the Data Step using Label statement | Labeling values can done by converting the column to Factor class and specifying levels. See page on [Factors](#factors). Column names are not typically labeled. |
| Records are selected using Where or If statement in the Data Step. Multiple selection conditions are separated using “and” command. | Records are selected using the function filter() with multiple selection conditions separated either by an AND operator (&) or a comma |
| Datasets are combined using Merge statement in the Data Step. The datasets to be merged need to be sorted first using Proc Sort procedure. | **dplyr** package offers a few functions for merging datasets. See page [Joining Data](#joining-data) for details. |

**Descriptive analysis**

| **SAS** | **R** |
| --- | --- |
| Get a high-level overview of your dataset using Proc Summary procedure, which provides the variable names and descriptive statistics | Get a high-level overview of your dataset using summary(dataset) or skim(dataset) from the **skimr** package |
| Tabulate counts of a variable using proc freq data=Dataset; Tables varname; Run; | See the page on [Descriptive tables](#descriptive-tables). Options include table() from **base** R, and tabyl() from **janitor** package, among others. Note you will need to specify the dataset and column name as R holds multiple datasets. |
| Cross-tabulation of two variables in a 2x2 table is done with proc freq data=Dataset; Tables rowvar\*colvar; Run; | Again, you can use table(), tabyl() or other options as described in the [Descriptive tables](#descriptive-tables) page. |

**Some useful resources:**

[R for SAS and SPSS Users (2011)](https://www.amazon.com/SAS-SPSS-Users-Statistics-Computing/dp/1461406846/ref=sr_1_1?dchild=1&gclid=EAIaIQobChMIoqLOvf6u7wIVAhLnCh1c9w_DEAMYASAAEgJLIfD_BwE&hvadid=241675955927&hvdev=c&hvlocphy=9032185&hvnetw=g&hvqmt=e&hvrand=16854847287059617468&hvtargid=kwd-44746119007&hydadcr=16374_10302157&keywords=r+for+sas+users&qid=1615698213&sr=8-1)

[SAS and R, Second Edition (2014)](https://www.amazon.com/SAS-Management-Statistical-Analysis-Graphics-dp-1466584491/dp/1466584491/ref=dp_ob_title_bk)

## Data interoperability

See the [Import and export](#import-and-export) page for details on how the R package **rio** can import and export files such as STATA .dta files, SAS .xpt and.sas7bdat files, SPSS .por and.sav files, and many others.

# Suggested packages

Below is a long list of suggested packages for common epidemiological work in R. You can copy this code, run it, and all of these packages will install from CRAN and load for use in the current R session. If a package is already installed, it will be loaded for use only.

You can modify the code with # symbols to exclude any packages you do not want.

Of note:

* Install the **pacman** package first before running the below code. You can do this with install.packages("pacman"). In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use in the current R session. You can also load packages that are already installed with library() from **base** R.
* In the code below, packages that are included when installing/loading another package are indicated by an indent and hash. For example how **ggplot2** is listed under **tidyverse**.
* If multiple packages have functions with the same name, masking can occur when the function from the more recently-loaded package takes precedent. Read more in the [R basics](#r-basics) page. Consider using the package **conflicted** to manage such conflicts.
* See the [R basics](#r-basics) section on packages for more information on **pacman** and masking.

To see the versions of R, RStudio, and R packages used during the production of this handbook, see the page on [Editorial and technical notes](#editorial-and-technical-notes).

## Packages from CRAN

## Packages from Github

Below are commmands to install two packages directly from Github repositories.

* The development version of **epicontacts** contains the ability to make transmission trees with an temporal x-axis
* The **epirhandbook** package contains all the example data for this handbook and can be used to download the offline version of the handbook.

# R projects

An R project enables your work to be bundled in a portable, self-contained folder. Within the project, all the relevant scripts, data files, figures/outputs, and history are stored in sub-folders and importantly - the working directory is the project’s root folder.

## Suggested use

A common, efficient, and trouble-free way to use R is to combine these 3 elements. One discrete work project is hosted within one R project. Each element is described in the sections below.

1. An **R project**
   * A self-contained working environment with folders for data, scripts, outputs, etc.
2. The **here** package for relative filepaths
   * Filepaths are written relative to the root folder of the R project - see [Import and export](#import-and-export) for more information
3. The **rio** package for importing/exporting
   * import() and export() handle any file type by by its extension (e.g. .csv, .xlsx, .png)

## Creating an R project

To create an R project, select “New Project” from the File menu.

* If you want to create a new folder for the project, select “New directory” and indicate where you want it to be created.
* If you want to create the project within an existing folder, click “Existing directory” and indicate the folder.
* If you want to clone a Github repository, select the third option “Version Control” and then “Git”. See the page on [Version control and collaboration with Git and Github](#version-control-and-collaboration-with-) for further details.

The R project you create will come in the form of a folder containing a .Rproj file. This file is a shortcut and likely the primary way you will open your project. You can also open a project by selecting “Open Project” from the File menu. Alternatively on the far upper right side of RStudio you will see an R project icon and a drop-down menu of available R projects.

To exit from an R project, either open a new project, or close the project (File - Close Project).

### Switch projects

To switch between projects, click the R project icon and drop-down menu at the very top-right of RStudio. You will see options to Close Project, Open Project, and a list of recent projects.

### Settings

It is generally advised that you start RStudio each time with a “clean slate” - that is, with your workspace **not** preserved from your previous session. This will mean that your objects and results will not persist session-to-session (you must re-create them by running your scripts). This is good, because it will force you to write better scripts and avoid errors in the long run.

To set RStudio to have a “clean slate” each time at start-up:

* Select “Project Options” from the Tools menu.
* In the “General” tab, set RStudio to **not** restore .RData into workspace at startup, and to **not** save workspace to .RData on exit.

### Organization

It is common to have subfolders in your project. Consider having folders such as “data”, “scripts”, “figures”, “presentations”. You can add folders in the typical way you would add a new folder for your computer. Alternatively, see the page on [Directory interactions](#directory-interactions) to learn how to create new folders with R commands.

### Version control

Consider a version control system. It could be something as simple as having dates on the names of scripts (e.g. “transmission\_analysis\_2020-10-03.R”) and an “archive” folder. Consider also having commented header text at the top of each script with a description, tags, authors, and change log.

A more complicated method would involve using Github or a similar platform for version control. See the page on [Version control and collaboration with Git and Github](#version-control-and-collaboration-with-).

One tip is that you can search across an entire project or folder using the “Find in Files” tool (Edit menu). It can search and even replace strings across multiple files.

## Examples

Below are some examples of import/export/saving using here() from within an R projct. Read more about using the **here** package in the [Import and export](#import-and-export) page.

Importing *linelist\_raw.xlsx* from the “data” folder in your R project

Exporting the R object *linelist* as “my\_linelist.rds” to the “clean” folder within the “data” folder in your R project.

Saving the most recently printed plot as “epicurve\_2021-02-15.png” within the “epicurves” folder in “outputs” folder in your R project.

## Resources

RStudio webpage on [using R projects](https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects)

# Import and export

In this page we describe ways to locate, import, and export files:

* Use of the **rio** package to flexibly import() and export() many types of files
* Use of the **here** package to locate files relative to an R project root - to prevent complications from file paths that are specific to one computer
* Specific import scenarios, such as:
  + Specific Excel sheets
  + Messy headers and skipping rows
  + From Google sheets
  + From data posted to websites
  + With APIs
  + Importing the most recent file
* Manual data entry
* R-specific file types such as RDS and RData
* Exporting/saving files and plots

## Overview

When you import a “dataset” into R, you are generally creating a new data frame object in your R environment and defining it as an imported file (e.g. Excel, CSV, TSV, RDS) that is located in your folder directories at a certain file path/address.

You can import/export many types of files, including those created by other statistical programs (SAS, STATA, SPSS). You can also connect to relational databases.

R even has its own data formats:

* An RDS file (.rds) stores a single R object such as a data frame. These are useful to store cleaned data, as they maintain R column classes. Read more in [this section](#import_rds).
* An RData file (.Rdata) can be used to store multiple objects, or even a complete R workspace. Read more in [this section](#import_rdata).

## The ****rio**** package

The R package we recommend is: **rio**. The name “rio” is an abbreviation of “R I/O” (input/output).

Its functions import() and export() can handle many different file types (e.g. .xlsx, .csv, .rds, .tsv). When you provide a file path to either of these functions (including the file extension like “.csv”), **rio** will read the extension and use the correct tool to import or export the file.

The alternative to using **rio** is to use functions from many other packages, each of which is specific to a type of file. For example, read.csv() (**base** R), read.xlsx() (**openxlsx** package), and write\_csv() (**readr** pacakge), etc. These alternatives can be difficult to remember, whereas using import() and export() from **rio** is easy.

**rio**’s functions import() and export() use the appropriate package and function for a given file, based on its file extension. See the end of this page for a complete table of which packages/functions **rio** uses in the background. It can also be used to import STATA, SAS, and SPSS files, among dozens of other file types.

Import/export of shapefiles requires other packages, as detailed in the page on [GIS basics](#gis-basics).

## The ****here**** package

The package **here** and its function here() make it easy to tell R where to find and to save your files - in essence, it builds file paths.

Used in conjunction with an R project, **here** allows you to describe the location of files in your R project in relation to the R project’s root directory (the top-level folder). This is useful when the R project may be shared or accessed by multiple people/computers. It prevents complications due to the unique file paths on different computers (e.g. "C:/Users/Laura/Documents..." by “starting” the file path in a place common to all users (the R project root).

This is how here() works within an R project:

* When the **here** package is first loaded within the R project, it places a small file called “.here” in the root folder of your R project as a “benchmark” or “anchor”
* In your scripts, to reference a file in the R project’s sub-folders, you use the function here() to build the file path in relation to that anchor
* To build the file path, write the names of folders beyond the root, within quotes, separated by commas, finally ending with the file name and file extension as shown below
* here() file paths can be used for both importing and exporting

For example, below, the function import() is being provided a file path constructed with here().

The command here("data", "linelists", "ebola\_linelist.xlsx") is actually providing the full file path that is unique to the user’s computer:

"C:/Users/Laura/Documents/my\_R\_project/data/linelists/ebola\_linelist.xlsx"

The beauty is that the R command using here() can be successfully run on any computer accessing the R project.

**TIP:** If you are unsure where the “.here” root is set to, run the function here() with empty parentheses.

Read more about the **here** package [at this link](https://here.r-lib.org/).

## File paths

When importing or exporting data, you must provide a file path. You can do this one of three ways:

1. Recommended: provide a “relative” file path with the **here** package
2. Provide the “full” / “absolute” file path
3. Manual file selection

### “Relative” file paths

In R, “relative” file paths consist of the file path relative to the root of an R project. They allow for more simple file paths that can work on different computers (e.g. if the R project is on a shared drive or is sent by email). As described [above](#here), relative file paths are facilitated by use of the **here** package.

An example of a relative file path constructed with here() is below. We assume the work is in an R project that contains a sub-folder “data” and within that a subfolder “linelists”, in which there is the .xlsx file of interest.

### “Absolute” file paths

Absolute or “full” file paths can be provided to functions like import() but they are “fragile” as they are unique to the user’s specific computer and therefore not recommended.

Below is an example of an absolute file path, where in Laura’s computer there is a folder “analysis”, a sub-folder “data” and within that a sub-folder “linelists”, in which there is the .xlsx file of interest.

A few things to note about absolute file paths:

* **Avoid using absolute file paths** as they will break if the script is run on a different computer
* Use forward slashes (/), as in the example above (note: this is NOT the default for Windows file paths)
* File paths that begin with double slashes (e.g. “//…”) will likely **not be recognized by R** and will produce an error. Consider moving your work to a “named” or “lettered” drive that begins with a letter (e.g. “J:” or “C:”). See the page on [Directory interactions](#directory-interactions) for more details on this issue.

One scenario where absolute file paths may be appropriate is when you want to import a file from a shared drive that has the same full file path for all users.

**TIP:** To quickly convert all \ to /, highlight the code of interest, use Ctrl+f (in Windows), check the option box for “In selection”, and then use the replace functionality to convert them.

### Select file manually

You can import data manually via one of these methods:

1. Environment RStudio Pane, click “Import Dataset”, and select the type of data
2. Click File / Import Dataset / (select the type of data)
3. To hard-code manual selection, use the base R command file.choose() (leaving the parentheses empty) to trigger appearance of a **pop-up window** that allows the user to manually select the file from their computer. For example:

**TIP:** The **pop-up window** may appear BEHIND your RStudio window.

## Import data

To use import() to import a dataset is quite simple. Simply provide the path to the file (including the file name and file extension) in quotes. If using here() to build the file path, follow the instructions above. Below are a few examples:

Importing a csv file that is located in your “working directory” or in the R project root folder:

Importing the first sheet of an Excel workbook that is located in “data” and “linelists” sub-folders of the R project (the file path built using here()):

Importing a data frame (a .rds file) using an absolute file path:

### Specific Excel sheets

By default, if you provide an Excel workbook (.xlsx) to import(), the workbook’s first sheet will be imported. If you want to import a specific **sheet**, include the sheet name to the which = argument. For example:

If using the here() method to provide a relative pathway to import(), you can still indicate a specific sheet by adding the which = argument after the closing parentheses of the here() function.

To export a data frame from R to a specific Excel sheet and have the rest of the Excel workbook remain unchanged, you will have to import, edit, and export with an alternative package catered to this purpose such as **openxlsx**. See more information in the page on [Directory interactions](#directory-interactions) or [at this github page](https://ycphs.github.io/openxlsx/).

If your Excel workbook is .xlsb (binary format Excel workbook) you may not be able to import it using **rio**. Consider re-saving it as .xlsx, or using a package like **readxlsb** which is built for [this purpose](https://cran.r-project.org/web/packages/readxlsb/vignettes/read-xlsb-workbook.html).

### Missing values

You may want to designate which value(s) in your dataset should be considered as missing. As explained in the page on [Missing data](#missing-data), the value in R for missing data is NA, but perhaps the dataset you want to import uses 99, “Missing”, or just empty character space "" instead.

Use the na = argument for import() and provide the value(s) within quotes (even if they are numbers). You can specify multiple values by including them within a vector, using c() as shown below.

Here, the value “99” in the imported dataset is considered missing and converted to NA in R.

Here, any of the values “Missing”, "" (empty cell), or " " (single space) in the imported dataset are converted to NA in R.

### Skip rows

Sometimes, you may want to avoid importing a row of data. You can do this with the argument skip = if using import() from **rio** on a .xlsx or .csv file. Provide the number of rows you want to skip.

Unfortunately skip = only accepts one integer value, not a range (e.g. “2:10” does not work). To skip import of specific rows that are not consecutive from the top, consider importing multiple times and using bind\_rows() from **dplyr**. See the example below of skipping only row 2.

### Manage a second header row

Sometimes, your data may have a second row, for example if it is a “data dictionary” row as shown below. This situation can be problematic because it can result in all columns being imported as class “character”.

Below is an example of this kind of dataset (with the first row being the data dictionary).

#### Remove the second header row

To drop the second header row, you will likely need to import the data twice.

1. Import the data in order to store the correct column names
2. Import the data again, skipping the first two rows (header and second rows)
3. Bind the correct names onto the reduced dataframe

The exact argument used to bind the correct column names depends on the type of data file (.csv, .tsv, .xlsx, etc.). This is because **rio** is using a different function for the different file types (see table above).

**For Excel files:** (col\_names =)

**For CSV files:** (col.names =)

**Backup option** - changing column names as a separate command

#### Make a data dictionary

Bonus! If you do have a second row that is a data dictionary, you can easily create a proper data dictionary from it. This tip is adapted from this [post](https://alison.rbind.io/post/2018-02-23-read-multiple-header-rows/).

#### Combine the two header rows

In some cases when your raw dataset has two header rows (or more specifically, the 2nd row of data is a secondary header), you may want to “combine” them or add the values in the second header row into the first header row.

The command below will define the data frame’s column names as the combination (pasting together) of the first (true) headers with the value immediately underneath (in the first row).

### Google sheets

You can import data from an online Google spreadsheet with the **googlesheet4** package and by authenticating your access to the spreadsheet.

Below, a demo Google sheet is imported and saved. This command may prompt confirmation of authentification of your Google account. Follow prompts and pop-ups in your internet browser to grant Tidyverse API packages permissions to edit, create, and delete your spreadsheets in Google Drive.

The sheet below is “viewable for anyone with the link” and you can try to import it.

The sheet can also be imported using only the sheet ID, a shorter part of the URL:

Another package, **googledrive** offers useful functions for writing, editing, and deleting Google sheets. For example, using the gs4\_create() and sheet\_write() functions found in this package.

Here are some other helpful online tutorials:  
[basic Google sheets importing tutorial](https://arbor-analytics.com/post/getting-your-data-into-r-from-google-sheets/)  
[more detailed tutorial](https://googlesheets4.tidyverse.org/articles/googlesheets4.html)  
[interaction between the googlesheets4 and tidyverse](https://googlesheets4.tidyverse.org/articles/articles/drive-and-sheets.html)

## Multiple files - import, export, split, combine

See the page on [Iteration, loops, and lists](#iteration-loops-and-lists) for examples of how to import and combine multiple files, or multiple Excel workbook files. That page also has examples on how to split a data frame into parts and export each one separately, or as named sheets in an Excel workbook.

## Import from Github

Importing data directly from Github into R can be very easy or can require a few steps - depending on the file type. Below are some approaches:

### CSV files

It can be easy to import a .csv file directly from Github into R with an R command.

1. Go to the Github repo, locate the file of interest, and click on it
2. Click on the “Raw” button (you will then see the “raw” csv data, as shown below)
3. Copy the URL (web address)
4. Place the URL in quotes within the import() R command

### XLSX files

You may not be able to view the “Raw” data for some files (e.g. .xlsx, .rds, .nwk, .shp)

1. Go to the Github repo, locate the file of interest, and click on it
2. Click the “Download” button, as shown below
3. Save the file on your computer, and import it into R

### Shapefiles

Shapefiles have many sub-component files, each with a different file extention. One file will have the “.shp” extension, but others may have “.dbf”, “.prj”, etc. To download a shapefile from Github, you will need to download each of the sub-component files individually, and save them in the same folder on your computer. In Github, click on each file individually and download them by clicking on the “Download” button.

Once saved to your computer you can import the shapefile as shown in the [GIS basics](#gis-basics) page using st\_read() from the **sf** package. You only need to provide the filepath and name of the “.shp” file - as long as the other related files are within the same folder on your computer.

Below, you can see how the shapefile “sle\_adm3” consists of many files - each of which must be downloaded from Github.

## Manual data entry

### Entry by rows

Use the tribble function from the **tibble** package from the tidyverse ([online tibble reference](https://tibble.tidyverse.org/reference/tribble.html)).

Note how column headers start with a tilde (~). Also note that each column must contain only one class of data (character, numeric, etc.). You can use tabs, spacing, and new rows to make the data entry more intuitive and readable. Spaces do not matter between values, but each row is represented by a new line of code. For example:

And now we display the new dataset:

### Entry by columns

Since a data frame consists of vectors (vertical columns), the **base** approach to manual dataframe creation in R expects you to define each column and then bind them together. This can be counter-intuitive in epidemiology, as we usually think about our data in rows (as above).

**CAUTION:** All vectors must be the same length (same number of values).

The vectors can then be bound together using the function data.frame():

And now we display the new dataset:

### Pasting from clipboard

If you copy data from elsewhere and have it on your clipboard, you can try one of the two ways below:

From the **clipr** package, you can use read\_clip\_tbl() to import as a data frame, or just just read\_clip() to import as a character vector. In both cases, leave the parentheses empty.

You can also easily export to your system’s clipboard with **clipr**. See the section below on Export.

Alternatively, you can use the the read.table() function from **base** R with file = "clipboard") to import as a data frame:

## Import most recent file

Often you may receive daily updates to your datasets. In this case you will want to write code that imports the most recent file. Below we present two ways to approach this:

* Selecting the file based on the date in the file name
* Selecting the file based on file metadata (last modification)

### Dates in file name

This approach depends on three premises:

1. You trust the dates in the file names
2. The dates are numeric and appear in generally the same format (e.g. year then month then day)
3. There are no other numbers in the file name

We will explain each step, and then show you them combined at the end.

First, use dir() from **base** R to extract just the file names for each file in the folder of interest. See the page on [Directory interactions](#directory-interactions) for more details about dir(). In this example, the folder of interest is the folder “linelists” within the folder “example” within “data” within the R project.

Once you have this vector of names, you can extract the dates from them by applying str\_extract() from **stringr** using this regular expression. It extracts any numbers in the file name (including any other characters in the middle such as dashes or slashes). You can read more about **stringr** in the [Strings and characters] page.

Assuming the dates are written in generally the same date format (e.g. Year then Month then Day) and the years are 4-digits, you can use **lubridate**’s flexible conversion functions (ymd(), dmy(), or mdy()) to convert them to dates. For these functions, the dashes, spaces, or slashes do not matter, only the order of the numbers. Read more in the [Working with dates](#working-with-dates-1) page.

The **base** R function which.max() can then be used to return the index position (e.g. 1st, 2nd, 3rd, …) of the maximum date value. The latest file is correctly identified as the 6th file - “case\_linelist\_2020-10-08.xlsx”.

If we condense all these commands, the complete code could look like below. Note that the . in the last line is a placeholder for the piped object at that point in the pipe sequence. At that point the value is simply the number 6. This is placed in double brackets to extract the 6th element of the vector of file names produced by dir().

You can now use this name to finish the relative file path, with here():

And you can now import the latest file:

### Use the file info

If your files do not have dates in their names (or you do not trust those dates), you can try to extract the last modification date from the file metadata. Use functions from the package **fs** to examine the metadata information for each file, which includes the last modification time and the file path.

Below, we provide the folder of interest to **fs**’s dir\_info(). In this case, the folder of interest is in the R project in the folder “data”, the sub-folder “example”, and its sub-folder “linelists”. The result is a data frame with one line per file and columns for modification\_time, path, etc. You can see a visual example of this in the page on [Directory interactions](#directory-interactions).

We can sort this data frame of files by the column modification\_time, and then keep only the top/latest row (file) with **base** R’s head(). Then we can extract the file path of this latest file only with the **dplyr** function pull() on the column path. Finally we can pass this file path to import(). The imported file is saved as latest\_file.

## APIs

An “Automated Programming Interface” (API) can be used to directly request data from a website. APIs are a set of rules that allow one software application to interact with another. The client (you) sends a “request” and receives a “response” containing content. The R packages **httr** and **jsonlite** can facilitate this process.

Each API-enabled website will have its own documentation and specifics to become familiar with. Some sites are publicly available and can be accessed by anyone. Others, such as platforms with user IDs and credentials, require authentication to access their data.

Needless to say, it is necessary to have an internet connection to import data via API. We will briefly give examples of use of APIs to import data, and link you to further resources.

Note: recall that data may be posted\* on a website without an API, which may be easier to retrieve. For example a posted CSV file may be accessible simply by providing the site URL to import() as described in the section on [importing from Github](#import_github).\*

### HTTP request

The API exchange is most commonly done through an HTTP request. HTTP is Hypertext Transfer Protocol, and is the underlying format of a request/response between a client and a server. The exact input and output may vary depending on the type of API but the process is the same - a “Request” (often HTTP Request) from the user, often containing a query, followed by a “Response”, containing status information about the request and possibly the requested content.

Here are a few components of an HTTP request:

* The URL of the API endpoint
* The “Method” (or “Verb”)
* Headers
* Body

The HTTP request “method” is the action your want to perform. The two most common HTTP methods are GET and POST but others could include PUT, DELETE, PATCH, etc. When importing data into R it is most likely that you will use GET.

After your request, your computer will receive a “response” in a format similar to what you sent, including URL, HTTP status (Status 200 is what you want!), file type, size, and the desired content. You will then need to parse this response and turn it into a workable data frame within your R environment.

### Packages

The **httr** package works well for handling HTTP requests in R. It requires little prior knowledge of Web APIs and can be used by people less familiar with software development terminology. In addition, if the HTTP response is .json, you can use **jsonlite** to parse the response.

### Publicly-available data

Below is an example of an HTTP request, borrowed from a tutorial from [the Trafford Data Lab](https://www.trafforddatalab.io/open_data_companion/#A_quick_introduction_to_APIs). This site has several other resources to learn and API exercises.

Scenario: We want to import a list of fast food outlets in the city of Trafford, UK. The data can be accessed from the API of the Food Standards Agency, which provides food hygiene rating data for the United Kingdom.

Here are the parameters for our request:

* HTTP verb: GET
* API endpoint URL: <http://api.ratings.food.gov.uk/Establishments>
* Selected parameters: name, address, longitude, latitude, businessTypeId, ratingKey, localAuthorityId
* Headers: “x-api-version”, 2
* Data format(s): JSON, XML
* Documentation: <http://api.ratings.food.gov.uk/help>

The R code would be as follows:

You can now clean and use the response data frame, which contains one row per fast food facility.

### Authentication required

Some APIs require authentication - for you to prove who you are, so you can access restricted data. To import these data, you may need to first use a POST method to provide a username, password, or code. This will return an access token, that can be used for subsequent GET method requests to retrieve the desired data.

Below is an example of querying data from Go.Data, which is an outbreak investigation tool. Go.Data uses an API for all interactions between the web front-end and smartphone applications used for data collection. Go.Data is used throughout the world. Because outbreak data are sensitive and you should only be able to access data for your outbreak, authentication is required.

Below is some sample R code using **httr** and **jsonlite** for connecting to the Go.Data API to import data on contact follow-up from your outbreak.

**CAUTION:** If you are importing large amounts of data from an API requiring authentication, it may time-out. To avoid this, retrieve access\_token again before each API GET request and try using filters or limits in the query.

**TIP:** The fromJSON() function in the **jsonlite** package does not fully un-nest the first time it’s executed, so you will likely still have list items in your resulting tibble. You will need to further un-nest for certain variables; depending on how nested your .json is. To view more info on this, view the documentation for the **jsonlite** package, such as the [flatten() function](https://rdrr.io/cran/jsonlite/man/flatten.html).

For more details, View documentation on [LoopBack Explorer](https://loopback.io/doc/en/lb4/index.html), the [Contact Tracing](#contact-tracing-1) page or API tips on [Go.Data Github repository](https://worldhealthorganization.github.io/godata/api-docs)

You can read more about the httr package [here](https://httr.r-lib.org/articles/quickstart.html)

This section was also informed by [this tutorial](https://www.dataquest.io/blog/r-api-tutorial/) and [this tutorial](https://medium.com/@traffordDataLab/querying-apis-in-r-39029b73d5f1).

## Export

### With ****rio**** package

With **rio**, you can use the export() function in a very similar way to import(). First give the name of the R object you want to save (e.g. linelist) and then in quotes put the file path where you want to save the file, including the desired file name and file extension. For example:

This saves the data frame linelist as an Excel workbook to the working directory/R project root folder:

You could save the same data frame as a csv file by changing the extension. For example, we also save it to a file path constructed with here():

### To clipboard

To export a data frame to your computer’s “clipboard” (to then paste into another software like Excel, Google Spreadsheets, etc.) you can use write\_clip() from the **clipr** package.

## RDS files

Along with .csv, .xlsx, etc, you can also export/save R data frames as .rds files. This is a file format specific to R, and is very useful if you know you will work with the exported data again in R.

The classes of columns are stored, so you don’t have do to cleaning again when it is imported (with an Excel or even a CSV file this can be a headache!). It is also a smaller file, which is useful for export and import if your dataset is large.

For example, if you work in an Epidemiology team and need to send files to a GIS team for mapping, and they use R as well, just send them the .rds file! Then all the column classes are retained and they have less work to do.

## Rdata files and lists

.Rdata files can store multiple R objects - for example multiple data frames, model results, lists, etc. This can be very useful to consolidate or share a lot of your data for a given project.

In the below example, multiple R objects are stored within the exported file “my\_objects.Rdata”:

Note: if you are trying to import a list, use import\_list() from **rio** to import it with the complete original structure and contents.

## Saving plots

Instructions on how to save plots, such as those created by ggplot(), are discussed in depth in the [ggplot basics](#ggplot-basics) page.

In brief, run ggsave("my\_plot\_filepath\_and\_name.png") after printing your plot. You can either provide a saved plot object to the plot = argument, or only specify the destination file path (with file extension) to save the most recently-displayed plot. You can also control the width =, height =, units =, and dpi =.

How to save a network graph, such as a transmission tree, is addressed in the page on [Transmission chains](#transmission-chains).

## Resources

The [R Data Import/Export Manual](https://cran.r-project.org/doc/manuals/r-release/R-data.html)  
[R 4 Data Science chapter on data import](https://r4ds.had.co.nz/data-import.html#data-import)  
[ggsave() documentation](https://ggplot2.tidyverse.org/reference/ggsave.html)

Below is a table, taken from the **rio** online [vignette](https://cran.r-project.org/web/packages/rio/vignettes/rio.html). For each type of data it shows: the expected file extension, the package **rio** uses to import or export the data, and whether this functionality is included in the default installed version of **rio**.

| **Format** | **Typical Extension** | **Import Package** | **Export Package** | **Installed by Default** |
| --- | --- | --- | --- | --- |
| Comma-separated data | .csv | data.table fread() | data.table | Yes |
| Pipe-separated data | .psv | data.table fread() | data.table | Yes |
| Tab-separated data | .tsv | data.table fread() | data.table | Yes |
| SAS | .sas7bdat | haven | haven | Yes |
| SPSS | .sav | haven | haven | Yes |
| Stata | .dta | haven | haven | Yes |
| SAS | XPORT | .xpt | haven | haven |
| SPSS Portable | .por | haven |  | Yes |
| Excel | .xls | readxl |  | Yes |
| Excel | .xlsx | readxl | openxlsx | Yes |
| R syntax | .R | base | base | Yes |
| Saved R objects | .RData, .rda | base | base | Yes |
| Serialized R objects | .rds | base | base | Yes |
| Epiinfo | .rec | foreign |  | Yes |
| Minitab | .mtp | foreign |  | Yes |
| Systat | .syd | foreign |  | Yes |
| “XBASE” | database files | .dbf | foreign | foreign |
| Weka Attribute-Relation File Format | .arff | foreign | foreign | Yes |
| Data Interchange Format | .dif | utils |  | Yes |
| Fortran data | no recognized extension | utils |  | Yes |
| Fixed-width format data | .fwf | utils | utils | Yes |
| gzip comma-separated data | .csv.gz | utils | utils | Yes |
| CSVY (CSV + YAML metadata header) | .csvy | csvy | csvy | No |
| EViews | .wf1 | hexView |  | No |
| Feather R/Python interchange format | .feather | feather | feather | No |
| Fast Storage | .fst | fst | fst | No |
| JSON | .json | jsonlite | jsonlite | No |
| Matlab | .mat | rmatio | rmatio | No |
| OpenDocument Spreadsheet | .ods | readODS | readODS | No |
| HTML Tables | .html | xml2 | xml2 | No |
| Shallow XML documents | .xml | xml2 | xml2 | No |
| YAML | .yml | yaml | yaml | No |
| Clipboard default is tsv |  | clipr | clipr | No |

# III Data Management

# Cleaning data and core functions

This page demonstrates common steps used in the process of “cleaning” a dataset, and also explains the use of many essential R data management functions.

To demonstrate data cleaning, this page begins by importing a raw case linelist dataset, and proceeds step-by-step through the cleaning process. In the R code, this manifests as a “pipe” chain, which references the “pipe” operator %>% that passes a dataset from one operation to the next.

### Core functions

This handbook emphasizes use of the functions from the [**tidyverse**](https://www.tidyverse.org/) family of R packages. The essential R functions demonstrated in this page are listed below.

Many of these functions belong to the [**dplyr**](https://dplyr.tidyverse.org/) R package, which provides “verb” functions to solve data manipulation challenges (the name is a reference to a "data frame-[plier](https://www.thefreedictionary.com/plier#:~:text=also%20ply%C2%B7er%20(pl%C4%AB%E2%80%B2,holding%2C%20bending%2C%20or%20cutting.)%22). **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** |
| 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 |
| clean\_variable\_spelling() | re-code/clean values using a data dictionary | **linelist** |
| which() | apply logical criteria; return indices | **base** R |

If you want to see how these functions compare to Stata or SAS commands, see the page on [Transition to R](#transition-to-r).

You may encounter an alternative data management framework from the **data.table** R package with operators like := and frequent use of brackets [ ]. This approach and syntax is briefly explained in the [Data Table](#data-table) page.

### Nomenclature

In this handbook, we generally reference “columns” and “rows” instead of “variables” and “observations”. As explained in this primer on [“tidy data”](https://tidyr.tidyverse.org/articles/tidy-data.html), most epidemiological statistical datasets consist structurally of rows, columns, and values.

Variables contain the values that measure the same underlying attribute (like age group, outcome, or date of onset). Observations contain all values measured on the same unit (e.g. a person, site, or lab sample). So these aspects can be more difficult to tangibly define.

In “tidy” datasets, each column is a variable, each row is an observation, and each cell is a single value. However some datasets you encounter will not fit this mold - a “wide” format dataset may have a variable split across several columns (see an example in the [Pivoting data](#pivoting-data) page). Likewise, observations could be split across several rows.

Most of this handbook is about managing and transforming data, so referring to the concrete data structures of rows and columns is more relevant than the more abstract observations and variables. Exceptions occur primarily in pages on data analysis, where you will see more references to variables and observations.

## Cleaning pipeline

**This page proceeds through typical cleaning steps, adding them sequentially to a cleaning pipe chain.**

In epidemiological analysis and data processing, cleaning steps are often performed sequentially, linked together. In R, this often manifests as a cleaning “pipeline”, where the raw dataset is passed or “piped” from one cleaning step to another.

Such chains utilize **dplyr** “verb” functions and the **magrittr** pipe operator %>%. This pipe begins with the “raw” data (“linelist\_raw.xlsx”) and ends with a “clean” R data frame (linelist) that can be used, saved, exported, etc.

In a cleaning pipeline the order of the steps is important. Cleaning steps might include:

* Importing of data
* Column names cleaned or changed
* De-duplication
* Column creation and transformation (e.g. re-coding or standardising values)
* Rows filtered or added

## Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

## Import data

### Import

Here we import the “raw” case linelist Excel file using the import() function from the package **rio**. The **rio** package flexibly handles many types of files (e.g. .xlsx, .csv, .tsv, .rds. See the page on [Import and export](#import-and-export) for more information and tips on unusual situations (e.g. skipping rows, setting missing values, importing Google sheets, etc).

If you want to follow along, [click to download the “raw” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_raw.xlsx) (as .xlsx file).

If your dataset is large and takes a long time to import, it can be useful to have the import command be separate from the pipe chain and the “raw” saved as a distinct file. This also allows easy comparison between the original and cleaned versions.

Below we import the raw Excel file and save it as the data frame linelist\_raw. We assume the file is located in your working directory or R project root, and so no sub-folders are specified in the file path.

You can view the first 50 rows of the the data frame below. Note: the **base** R function head(n) allow you to view just the first n rows in the R console.

### Review

You can use the function skim() from the package **skimr** to get an overview of the entire dataframe (see page on [Descriptive tables](#descriptive-tables) for more info). Columns are summarised by class/type such as character, numeric. Note: “POSIXct” is a type of raw date class (see [Working with dates](#working-with-dates-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 (e.g. axis or legend titles of a plot, or column headers in a printed table - see the [scales section of the ggplot tips page](#ggplot_tips_scales) and [Tables for presentation](#tables-for-presentation) pages). If you want to assign column labels in the data, read more online [here](https://cran.r-project.org/web/packages/expss/vignettes/labels-support.html) and [here](https://cran.r-project.org/web/packages/labelled/vignettes/intro_labelled.html).

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…)

The columns names of linelist\_raw are printed below using names() from **base** R. We can see that initially:

* Some names contain spaces (e.g. infection date)
* Different naming patterns are used for dates (date onset vs. infection date)
* There must have been a merged header across the two last columns in the .xlsx. We know this because the name of two merged columns (“merged\_header”) was assigned by R to the first column, and the second column was assigned a placeholder name “…28” (as it was then empty and is the 28th column).

**NOTE:** To reference a column name that includes spaces, surround the name with back-ticks, for example: linelist$` '\x60infection date\x60'`. note that on your keyboard, the back-tick (`) is different from the single quotation mark (’).

### Labels

Some other statistical software such as SAS have variable labels

### Automatic cleaning

The function clean\_names() from the package **janitor** standardizes column names and makes them unique by doing the following:

* Converts all names to consist of only underscores, numbers, and letters
* Accented characters are transliterated to ASCII (e.g. german o with umlaut becomes “o”, spanish “enye” becomes “n”)
* Capitalization preference for the new column names can be specified using the case = argument (“snake” is default, alternatives include “sentence”, “title”, “small\_camel”…)
* You can specify specific name replacements by providing a vector to the replace = argument (e.g. replace = c(onset = "date\_of\_onset"))
* Here is an online [vignette](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html#cleaning)

Below, the cleaning pipeline begins by using clean\_names() on the raw linelist.

**NOTE:** The last column name “…28” was changed to “x28”.

### Manual name cleaning

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:

#### Rename by column position

You can also rename by column position, instead of column name, for example:

#### 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](#grouping-data) and [Descriptive tables](#descriptive-tables) pages. These functions also uses the format new\_name = old\_name. Here is an example:

### Other challenges

#### 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. As explained in [Transition to R](#transition-to-r), 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.

Remind people doing data entry that **human-readable data is not the same as machine-readable data**. Strive to train users about the principles of [**tidy data**](https://r4ds.had.co.nz/tidy-data.html). If at all possible, try to change procedures so that data arrive in a tidy format without merged cells.

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

When using **rio**’s import() function, the value in a merged cell will be assigned to the first cell and subsequent cells will be empty.

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.

**DANGER:** If column names are merged with readWorkbook(), you will end up with duplicate column names, which you will need to fix manually - R does not work well with duplicate column names! You can re-name them by referencing their position (e.g. column 5), as explained in the section on manual column name cleaning.

## 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.

**CAUTION:** In the examples below, the linelist data frame is modified with select() and displayed, but not saved. This is for demonstration purposes. The modified column names are printed by piping the data frame to names().

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

### 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).

### “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:

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.

Use contains() to select only columns in which the column name contains a specified character string. ends\_with() and starts\_with() provide more nuance.

The function matches() works similarly to contains() but can be provided a regular expression (see page on [Characters and strings](#characters-and-strings)), such as multiple strings separated by OR bars within the parentheses:

**CAUTION:** If a column name that you specifically provide does not exist in the data, it can return an error and stop your code. Consider using any\_of() to cite columns that may or may not exist, especially useful in negative (remove) selections.

Only one of these columns exists, but no error is produced and the code continues without stopping your cleaning chain.

### 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.

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

### 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.

#### Add to the pipe chain

In the linelist\_raw, there are a few columns we do not need: row\_num, merged\_header, and x28. We remove them with a select() command in the cleaning pipe chain:

## Deduplication

See the handbook page on [De-duplication](#de-duplication) for extensive options on 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.

After de-duplication there are nrow(linelist) rows. Any removed rows would have been 100% duplicates of other rows.

Below, the distinct() command is added to the cleaning pipe chain:

## Column creation and transformation

**We recommend using the dplyr function mutate() to add a new column, or to modify an existing one.**

Below is an example of creating a new column with mutate(). The syntax is: mutate(new\_column\_name = value or transformation)

In Stata, this is similar to the command generate, but R’s 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.

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.

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](#characters-and-strings).

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

**TIP:** 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](#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!

The class of the date\_onset column is also character! To perform analyses, these dates must be recognized as dates!

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:

In a similar way, you can use as.character() and as.logical(). To convert to class Factor, you can use factor() from **base** R or as\_factor() from **forcats**. Read more about this in the [Factors](#factors) page.

You must be careful when converting to class Date. Several methods are explained on the page [Working with dates](#working-with-dates-1). Typically, the raw date values must all be in the same format for conversion to work correctly (e.g “MM/DD/YYYY”, or “DD MM YYYY”). After converting to class Date, check your data to confirm that each value was converted correctly.

### Grouped data

If your data frame is already grouped (see page on [Grouping data](#grouping-data)), mutate() may behave differently than if the data frame is not grouped. Any summarizing functions, like mean(), median(), max(), etc. will calculate by group, not by all the rows.

Read more about using mutate () on grouped dataframes in this [tidyverse mutate documentation](https://dplyr.tidyverse.org/reference/mutate.html).

### Transform multiple columns

Often to write concise code you want to apply the same transformation to multiple columns at once. A transformation can be applied to multiple columns at once using the across() function from the package **dplyr** (also contained within **tidyverse** package). across() can be used with any **dplyr** function, but is commonly used within select(), mutate(), filter(), or summarise(). See how it is applied to summarise() in the page on [Descriptive tables](#descriptive-tables).

Specify the columns to the argument .cols = and the function(s) to apply to .fns =. Any additional arguments to provide to the .fns function can be included after a comma, still within across().

#### across() column selection

Specify the columns to the argument .cols =. You can name them individually, or use “tidyselect” helper functions. Specify the function to .fns =. Note that using the function mode demonstrated below, the function is written without its parentheses ( ).

Here the transformation as.character() is applied to specific columns named within across().

The “tidyselect” helper functions are available to assist you in specifying columns. They are detailed above in the section on Selecting and re-ordering columns, and they include: everything(), last\_col(), where(), starts\_with(), ends\_with(), contains(), matches(), num\_range() and any\_of().

Here is an example of how one would change **all columns** to character class:

Convert to character all columns where the name contains the string “date” (note the placement of commas and parentheses):

Below, an example of mutating the columns that are currently class POSIXct (a raw datetime class that shows timestamps) - in other words, where the function is.POSIXct() evaluates to TRUE. Then we want to apply the function as.Date() to these columns to convert them to a normal class Date.

* Note that within across() we also use the function where() as is.POSIXct is evaluating to either TRUE or FALSE.
* Note that is.POSIXct() is from the package **lubridate**. Other similar “is” functions like is.character(), is.numeric(), and is.logical() are from **base R**

#### across() functions

You can read the documentation with ?across for details on how to provide functions to across(). A few summary points: there are several ways to specify the function(s) to perform on a column and you can even define your own functions:

* You can provide the function name alone (e.g. mean or as.character)
* You can provide the function in **purrr**-style (e.g. ~ mean(.x, na.rm = TRUE)) (see [this page](#iteration-loops-and-lists))
* You can specify multiple functions by providing a list (e.g. list(mean = mean, n\_miss = ~ sum(is.na(.x))).
  + If you provide multiple functions, multiple transformed columns will be returned per input column, with unique names in the format col\_fn. You can adjust how the new columns are named with the .names = argument using **glue** syntax (see page on [Characters and strings](#characters-and-strings)) where {.col} and {.fn} are shorthand for the input column and function.

Here are a few online resources on using across(): [creator Hadley Wickham’s thoughts/rationale](https://www.tidyverse.org/blog/2020/04/dplyr-1-0-0-colwise/)

### coalesce()

This **dplyr** function finds the first non-missing value at each position. It “fills-in” missing values with the first available value in an order you specify.

Here is an example outside the context of a data frame: Let us say you have two vectors, one containing the patient’s village of detection and another containing the patient’s village of residence. You can use coalesce to pick the first non-missing value for each index:

This works the same if you provide data frame columns: for each row, the function will assign the new column value with the first non-missing value in the columns you provided (in order provided).

This is an example of a “row-wise” operation. For more complicated row-wise calculations, see the section below on Row-wise calculations.

### Cumulative math

If you want a column to reflect the cumulative sum/mean/min/max etc as assessed down the rows of a dataframe to that point, use the following functions:

cumsum() returns the cumulative sum, as shown below:

This can be used in a dataframe when making a new column. For example, to calculate the cumulative number of cases per day in an outbreak, consider code like this:

Below are the first 10 rows:

See the page on [Epidemic curves](#epidemic-curves) for how to plot cumulative incidence with the epicurve.

See also:  
cumsum(), cummean(), cummin(), cummax(), cumany(), cumall()

### Using ****base**** R

To define a new column (or re-define a column) using **base** R, write the name of data frame, connected with $, to the new column (or the column to be modified). Use the assignment operator <- to define the new value(s). Remember that when using **base** R you must specify the data frame name before the column name every time (e.g. dataframe$column). Here is an example of creating the bmi column using **base** R:

### Add to pipe chain

**Below, a new column is added to the pipe chain and some classes are converted.**

## 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.

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.

**Here is another example re-coding multiple values within one column.**

In linelist the values in the column “hospital” must be cleaned. There are several different spellings and many missing values.

The recode() command below re-defines the column “hospital” as the current column “hospital”, but with the specified recode changes. Don’t forget commas after each!

Now we see the spellings in the hospital column have been corrected and consolidated:

**TIP:** The number of spaces before and after an equals sign does not matter. Make your code easier to read by aligning the = for all or most rows. Also, consider adding a hashed comment row to clarify for future readers which side is OLD and which side is NEW.

**TIP:** Sometimes a blank character value exists in a dataset (not recognized as R’s value for missing - NA). You can reference this value with two quotation marks with no space inbetween ("").

### 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”.

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’”.

#### 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”.

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.

**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:

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. Thus, it is best to write the most specific criteria first, and the most general last.

Along those lines, in your final statement, 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.

**DANGER:** **Vvalues on the right-side must all be the same class** - either numeric, character, date, logical, etc. To assign missing (NA), you may need to use special variations of NA such as NA\_character\_, NA\_real\_ (for numeric or POSIX), and as.Date(NA). Read more in [Working with dates](#working-with-dates-1).

### Missing values

Below are special functions for handling missing values in the context of data cleaning.

See the page on [Missing data](#missing-data) for more detailed tips on identifying and handling missing values. For example, the is.na() function which logically tests for missingness.

**replace\_na()**

To change missing values (NA) to a specific value, such as “Missing”, use the **dplyr** function replace\_na() within mutate(). Note that this is used in the same manner as recode above - the name of the variable must be repeated within replace\_na().

**fct\_explicit\_na()**

This is a function from the **forcats** package. The **forcats** package handles columns of class Factor. Factors are R’s way to handle ordered values such as c("First", "Second", "Third") or to set the order that values (e.g. hospitals) appear in tables and plots. See the page on [Factors](#factors).

If your data are class Factor and you try to convert NA to “Missing” by using replace\_na(), you will get this error: invalid factor level, NA generated. You have tried to add “Missing” as a value, when it was not defined as a possible level of the factor, and it was rejected.

The easiest way to solve this is to use the **forcats** function fct\_explicit\_na() which converts a column to class factor, and converts NA values to the character “(Missing)”.

A slower alternative would be to add the factor level using fct\_expand() and then convert the missing values.

**na\_if()**

To convert a specific value to NA, use **dplyr**’s na\_if(). The command below performs the opposite operation of replace\_na(). In the example below, any values of “Missing” in the column hospital are converted to NA.

Note: na\_if() **cannot be used for logic criteria** (e.g. “all values > 99”) - use replace() or case\_when() for this:

### Cleaning dictionary

Use the R package **linelist** and it’s function clean\_variable\_spelling() to clean a data frame with a cleaning dictionary. **linelist** is a package developed by [RECON](https://github.com/reconhub/linelist) - the R Epidemics Consortium.

1. Create a cleaning dictionary with 3 columns:
   * A “from” column (the incorrect value)
   * A “to” column (the correct value)
   * A column specifying the column for the changes to be applied (or “.global” to apply to all columns)

Note: .global dictionary entries will be overridden by column-specific dictionary entries.

1. Import the dictionary file into R. This example can be downloaded via instructions on the [Download handbook and data](#download-handbook-and-data) page.
2. Pass the raw linelist to clean\_variable\_spelling(), specifying to wordlists = the cleaning dictionary data frame. The spelling\_vars = argument can be used to specify which column in the dictionary refers to the columns (3rd by default), or can be set to NULL to have the dictionary apply to all character and factor columns. Note this function can take a long time to run.

Now scroll to the right to see how values have changed - particularly gender (lowercase to uppercase), and all the symptoms columns have been transformed from yes/no to 1/0.

Note that your column names in the cleaning dictionary must correspond to the names at this point in your cleaning script. See this [online reference for the linelist package](https://www.repidemicsconsortium.org/linelist/reference/clean_data.html) for more details.

#### Add to pipe chain

**Below, some new columns and column transformations are added to the pipe chain.**

## 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.

First, examine the distribution of your data, to make appropriate cut-points. See the page on [ggplot basics](#ggplot-basics).

**CAUTION:** Sometimes, numeric variables will import as class “character”. This occurs if there are non-numeric characters in some of the values, for example an entry of “2 months” for age, or (depending on your R locale settings) if a comma is used in the decimals place (e.g. “4,5” to mean four and one half years)..

### age\_categories()

With the **epikit** package, you can use the age\_categories() function to easily categorize and label numeric columns (note: this function can be applied to non-age numeric variables too). As a bonum, the output column is automatically an ordered factor.

Here are the required inputs:

* A numeric vector (column)
* The breakers = argument - provide a numeric vector of break points for the new groups

First, the simplest example:

The break values you specify are by default the lower bounds - that is, they are included in the “higher” group / the groups are “open” on the lower/left side. As shown below, you can add 1 to each break value to achieve groups that are open at the top/right.

You can adjust how the labels are displayed with separator =. The default is “-”

You can adjust how the top numbers are handled, with the ceiling = arguemnt. To set an upper cut-off set ceiling = TRUE. In this use, the highest break value provided is a “ceiling” and a category “XX+” is not created. Any values above highest break value (or to upper =, if defined) are categorized as NA. Below is an example with ceiling = TRUE, so that there is no category of XX+ and values above 70 (the highest break value) are assigned as NA.

Alternatively, instead of breakers =, you can provide all of lower =, upper =, and by =:

* lower = The lowest number you want considered - default is 0
* upper = The highest number you want considered
* by = The number of years between groups

See the function’s Help page for more details (enter ?age\_categories in the R console).

### cut()

cut() is a **base** R alternative to age\_categories(), but I think you will see why age\_categories() was developed to simplify this process. Some notable differences from age\_categories() are:

* You do not need to install/load another package
* You can specify whether groups are open/closed on the right/left
* You must provide accurate labels yourself
* If you want 0 included in the lowest group you must specify this

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:

**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).

**Re-labeling NA values**

You may want to assign NA values a label such as “Missing”. Because the new column is class Factor (restricted values), you cannot simply mutate it with replace\_na(), as this value will be rejected. Instead, use fct\_explicit\_na() from **forcats** as explained in the [Factors](#factors) page.

**Quickly make breaks and labels**

For a fast way to make breaks and label vectors, use something like below. See the [R basics](#r-basics) page for references on seq() and rep().

Read more about cut() in its Help page by entering ?cut in the R console.

### 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

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 (see the [Descriptive tables](#descriptive-tables) page). Note how they are not exactly 10% in each group.

### Evenly-sized groups

Another tool to make numeric groups is the the **dplyr** function ntile(), which attempts to break your data into n evenly-sized groups - but be aware that unlike with *quantile()* the same value could appear in more than one group. Provide the numeric vector and then the number of groups. The values in the new column created is just group “numbers” (e.g. 1 to 10), not the range of values themselves as when using cut().

### case\_when()

It is possible to use the **dplyr** function case\_when() to create categories from a numeric column, but it is easier to use age\_categories() from **epikit** or cut() because these will create an ordered factor automatically.

If using case\_when(), please review the proper use as described earlier in the Re-code values section of this page. Also be aware that all right-hand side values must be of the same class. Thus, if you want NA on the right-side you should either write “Missing” or use the special NA value NA\_character\_.

### Add to pipe chain

Below, code to create two categorical age columns is added to the cleaning pipe chain:

## Add rows

### One-by-one

Adding rows one-by-one manually is tedious but can be done with add\_row() from **dplyr**. Remember that each column must contain values of only one class (either character, numeric, logical, etc.). So adding a row requires nuance to maintain this.

Use .before and .after. to specify the placement of the row you want to add. .before = 3 will put the new row before the current 3rd row. The default behavior is to add the row to the end. Columns not specified will be left empty (NA).

The new row number may look strange (“…23”) but the row numbers in the pre-existing rows have changed. So if using the command twice, examine/test the insertion carefully.

If a class you provide is off you will see an error like this:

Error: Can't combine ..1$infection date <date> and ..2$infection date <character>.

(when inserting a row with a date value, remember to wrap the date in the function as.Date() like as.Date("2020-10-10")).

### Bind rows

To combine datasets together by binding the rows of one dataframe to the bottom of another data frame, you can use bind\_rows() from **dplyr**. This is explained in more detail in the page [Joining data](#joining-data).

## 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).

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

### Filter out missing values

It is fairly common to want to filter out rows that have missing values. Resist the urge to write filter(!is.na(column) & !is.na(column)) and instead use the **tidyr** function that is custom-built for this purpose: drop\_na(). If run with empty parentheses, it removes rows with any missing values. Alternatively, you can provide names of specific columns to be evaluated for missingness, or use the “tidyselect” helper functions described [above](#clean_tidyselect).

See the page on [Missing data](#missing-data) for many techniques to analyse and manage missingness in your data.

### Filter by row number

In a data frame or tibble, each row will usually have a “row number” that (when seen in R Viewer) appears to the left of the first column. It is not itself a true column in the data, but it can be used in a filter() statement.

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().

You can also convert the row numbers to a true column by piping your data frame to the **tibble** function rownames\_to\_column() (do not put anything in the parentheses).

### 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.**

#### 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!**

**DANGER:** 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.

(See the page on [*Working with dates*](#working-with-dates-1) for more information on working with dates and the package ***lubridate***)

#### Design the filter

Examine a cross-tabulation to make sure we exclude only the correct rows:

What other criteria can we filter on to remove the first outbreak (in 2012 & 2013) from the dataset? We see that:

* The first epidemic in 2012 & 2013 occurred at Hospital A, Hospital B, and that there were also 10 cases at Port Hospital.
* Hospitals A & B did not have cases in the second epidemic, but Port Hospital did.

We want to exclude:

* The nrow(linelist %>% filter(hospital %in% c("Hospital A", "Hospital B") | date\_onset < as.Date("2013-06-01"))) rows with onset in 2012 and 2013 at either hospital A, B, or Port:
  + Exclude nrow(linelist %>% filter(date\_onset < as.Date("2013-06-01"))) rows with onset in 2012 and 2013
  + Exclude nrow(linelist %>% filter(hospital %in% c('Hospital A', 'Hospital B') & is.na(date\_onset))) rows from Hospitals A & B with missing onset dates
  + Do **not** exclude nrow(linelist %>% filter(!hospital %in% c('Hospital A', 'Hospital B') & is.na(date\_onset))) other rows with missing onset dates.

We start with a linelist of nrow(linelist)`. Here is our filter statement:

When we re-make the cross-tabulation, we see that Hospitals A & B are removed completely, and the 10 Port Hospital cases from 2012 & 2013 are removed, and all other values are the same - just as we wanted.

Multiple statements can be included within one filter command (separated by commas), or you can always pipe to a separate filter() command for clarity.

Note: some readers may notice that it would be easier to just filter by *date\_hospitalisation* because it is 100% complete with no missing values. This is true. But *date\_onset* is used for purposes of demonstrating a complex filter.

### 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.

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

### Quickly review records

Often you want to quickly review a few records, for only a few columns. The **base** R function View() will print a data frame for viewing in your RStudio.

View the linelist in RStudio:

Here are two examples of viewing specific cells (specific rows, and specific columns):

**With dplyr functions filter() and select():**

Within View(), pipe the dataset to filter() to keep certain rows, and then to select() to keep certain columns. For example, to review onset and hospitalization dates of 3 specific cases:

You can achieve the same with **base** R syntax, using brackets [ ] to subset you want to see.

#### Add to pipe chain

## Row-wise calculations

If you want to perform a calculation within a row, you can use rowwise() from **dplyr**. See this online vignette on [row-wise calculations](https://cran.r-project.org/web/packages/dplyr/vignettes/rowwise.html).  
For example, this code applies rowwise() and then creates a new column that sums the number of the specified symptom columns that have value “yes”, for each row in the linelist. The columns are specified within sum() by name within a vector c(). rowwise() is essentially a special kind of group\_by(), so it is best to use ungroup() when you are done (page on [Grouping data](#grouping-data)).

As you specify the column to evaluate, you may want to use the “tidyselect” helper functions described in the select() section of this page. You just have to make one adjustment (because you are not using them within a **dplyr** function like select() or summarise()).

Put the column-specification criteria within the **dplyr** function c\_across(). This is because c\_across ([documentation](https://dplyr.tidyverse.org/reference/c_across.html)) is designed to work with rowwise() specifically. For example, the following code:

* Applies rowwise() so the following operation (sum()) is applied within each row (not summing entire columns)
* Creates new column num\_NA\_dates, defined for each row as the number of columns (with name containing “date”) for which is.na() evaluated to TRUE (they are missing data).
* ungroup() to remove the effects of rowwise() for subsequent steps

You could also provide other functions, such as max() to get the latest or most recent date for each row:

## 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](#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](#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:

# Working with dates

Working with dates in R requires more attention than working with other object classes. Below, we offer some tools and example to make this process less painful. Luckily, dates can be wrangled easily with practice, and with a set of helpful packages such as **lubridate**.

Upon import of raw data, R often interprets dates as character objects - this means they cannot be used for general date operations such as making time series and calculating time intervals. To make matters more difficult, there are many ways a date can be formatted and you must help R know which part of a date represents what (month, day, hour, etc.).

Dates in R are their own class of object - the Date class. It should be noted that there is also a class that stores objects with date and time. Date time objects are formally referred to as POSIXt, POSIXct, and/or POSIXlt classes (the difference isn’t important). These objects are informally referred to as datetime classes.

* It is important to make R recognize when a column contains dates.
* Dates are an object class and can be tricky to work with.
* Here we present several ways to convert date columns to Date class.

## Preparation

### Load packages

This code chunk shows the loading of packages required for this page. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

We import the dataset of cases from a simulated Ebola epidemic. If you want to download the data to follow along step-by-step, see instruction in the [Download handbook and data](#download-handbook-and-data) page. We assume the file is in the working directory so no sub-folders are specified in this file path.

## Current date

You can get the current “system” date or system datetime of your computer by doing the following with **base** R.

With the **lubridate** package these can also be returned with today() and now(), respectively. date() returns the current date and time with weekday and month names.

## Convert to Date

After importing a dataset into R, date column values may look like “1989/12/30”, “05/06/2014”, or “13 Jan 2020”. In these cases, R is likely still treating these values as Character values. R must be told that these values are dates… and what the format of the date is (which part is Day, which is Month, which is Year, etc).

Once told, R converts these values to class Date. In the background, R will store the dates as numbers (the number of days from its “origin” date 1 Jan 1970). You will not interface with the date number often, but this allows for R to treat dates as continuous variables and to allow special operations such as calculating the distance between dates.

By default, values of class Date in R are displayed as YYYY-MM-DD. Later in this section we will discuss how to change the display of date values.

Below we present two approaches to converting a column from character values to class Date.

**TIP:** You can check the current class of a column with **base** R function class(), like class(linelist$date\_onset).

### ****base**** R

as.Date() is the standard, **base** R function to convert an object or column to class Date (note capitalization of “D”).

Use of as.Date() requires that:

* You specify the ***existing*** format of the raw character date or the origin date if supplying dates as numbers (see section on Excel dates)
* If used on a character column, all date values must have the same exact format (if this is not the case, try guess\_dates() from the **linelist** package)

**First**, check the class of your column with class() from **base** R. If you are unsure or confused about the class of your data (e.g. you see “POSIXct”, etc.) it can be easiest to first convert the column to class Character with as.character(), and then convert it to class Date.

**Second**, within the as.Date() function, use the format = argument to tell R the current format of the character date components - which characters refer to the month, the day, and the year, and how they are separated. If your values are already in one of R’s standard date formats (“YYYY-MM-DD” or “YYYY/MM/DD”) the format = argument is not necessary.

To format =, provide a character string (in quotes) that represents the current date format using the special “strptime” abbreviations below. For example, if your character dates are currently in the format “DD/MM/YYYY”, like “24/04/1968”, then you would use format = "%d/%m/%Y" to convert the values into dates. **Putting the format in quotation marks is necessary. And don’t forget any slashes or dashes!**

Most of the strptime abbreviations are listed below. You can see the complete list by running ?strptime.

%d = Day number of month (5, 17, 28, etc.)  
%j = Day number of the year (Julian day 001-366)  
%a = Abbreviated weekday (Mon, Tue, Wed, etc.)  
%A = Full weekday (Monday, Tuesday, etc.) %w = Weekday number (0-6, Sunday is 0)  
%u = Weekday number (1-7, Monday is 1)  
%W = Week number (00-53, Monday is week start)  
%U = Week number (01-53, Sunday is week start)  
%m = Month number (e.g. 01, 02, 03, 04)  
%b = Abbreviated month (Jan, Feb, etc.)  
%B = Full month (January, February, etc.)  
%y = 2-digit year (e.g. 89)  
%Y = 4-digit year (e.g. 1989)  
%h = hours (24-hr clock)  
%m = minutes  
%s = seconds %z = offset from GMT  
%Z = Time zone (character)

**TIP:** The format = argument of as.Date() is not telling R the format you want the dates to be, but rather how to identify the date parts as they are before you run the command.

**TIP:** Be sure that in the format = argument you use the date-part separator (e.g. /, -, or space) that is present in your dates.

Once the values are in class Date, R will by default display them in the standard format, which is YYYY-MM-DD.

### ****lubridate****

Converting character objects to dates can be made easier by using the **lubridate** package. This is a **tidyverse** package designed to make working with dates and times more simple and consistent than in **base** R. For these reasons, **lubridate** is often considered the gold-standard package for dates and time, and is recommended whenever working with them.

The **lubridate** package provides several different helper functions designed to convert character objects to dates in an intuitive, and more lenient way than specifying the format in as.Date(). These functions are specific to the rough date format, but allow for a variety of separators, and synonyms for dates (e.g. 01 vs Jan vs January) - they are named after abbreviations of date formats.

The ymd() function flexibly converts date values supplied as **year, then month, then day**.

The mdy() function flexibly converts date values supplied as **month, then day, then year**.

The dmy() function flexibly converts date values supplied as **day, then month, then year**.

If using piping, the conversion of a character column to dates with **lubridate** might look like this:

Once complete, you can run class() to verify the class of the column

Once the values are in class Date, R will by default display them in the standard format, which is YYYY-MM-DD.

Note that the above functions work best with 4-digit years. 2-digit years can produce unexpected results, as lubridate attempts to guess the century.

To convert a 2-digit year into a 4-digit year (all in the same century) you can convert to class character and then combine the existing digits with a pre-fix using str\_glue() from the **stringr** package (see [Characters and strings](#characters-and-strings)). Then convert to date.

### Combine columns

You can use the **lubridate** functions make\_date() and make\_datetime() to combine multiple numeric columns into one date column. For example if you have numeric columns onset\_day, onset\_month, and onset\_year in the data frame linelist:

## Excel dates

In the background, most software store dates as numbers. R stores dates from an origin of 1st January, 1970. Thus, if you run as.numeric(as.Date("1970-01-01)) you will get 0.

Microsoft Excel stores dates with an origin of either December 30, 1899 (Windows) or January 1, 1904 (Mac), depending on your operating system. See this [Microsoft guidance](https://docs.microsoft.com/en-us/office/troubleshoot/excel/1900-and-1904-date-system) for more information.

Excel dates often import into R as these numeric values instead of as characters. If the dataset you imported from Excel shows dates as numbers or characters like “41369”… use as.Date() (or **lubridate**’s as\_date() function) to convert, but **instead of supplying a “format” as above, supply the Excel origin date** to the argument origin =.

This will not work if the Excel date is stored in R as a character type, so be sure to ensure the number is class Numeric!

**NOTE:** You should provide the origin date in R’s default date format (“YYYY-MM-DD”).

## Messy dates

The function guess\_dates() from the **linelist** package attempts to read a “messy” date column containing dates in many different formats and convert the dates to a standard format. You can [read more online about guess\_dates()](https://www.repidemicsconsortium.org/linelist/reference/guess_dates.html). If guess\_dates() is not yet available on CRAN for R 4.0.2, try install via pacman::p\_load\_gh("reconhub/linelist").

For example guess\_dates would see a vector of the following character dates “03 Jan 2018”, “07/03/1982”, and “08/20/85” and convert them to class Date as: 2018-01-03, 1982-03-07, and 1985-08-20.

Some optional arguments for guess\_dates() that you might include are:

* error\_tolerance - The proportion of entries which cannot be identified as dates to be tolerated (defaults to 0.1 or 10%)
* last\_date - the last valid date (defaults to current date)
* first\_date - the first valid date. Defaults to fifty years before the last\_date.

## Working with date-time class

As previously mentioned, R also supports a datetime class - a column that contains date **and** time information. As with the Date class, these often need to be converted from character objects to datetime objects.

### Convert dates with times

A standard datetime object is formatted with the date first, which is followed by a time component - for example 01 Jan 2020, 16:30. As with dates, there are many ways this can be formatted, and there are numerous levels of precision (hours, minutes, seconds) that can be supplied.

Luckily, **lubridate** helper functions also exist to help convert these strings to datetime objects. These functions are extensions of the date helper functions, with \_h (only hours supplied), \_hm (hours and minutes supplied), or \_hms (hours, minutes, and seconds supplied) appended to the end (e.g. dmy\_hms()). These can be used as shown:

Convert datetime with only hours to datetime object

Convert datetime with hours and minutes to datetime object

Convert datetime with hours, minutes, and seconds to datetime object

You can supply time zone but it is ignored. See section later in this page on time zones.

When working with a data frame, time and date columns can be combined to create a datetime column using str\_glue() from **stringr** package and an appropriate **lubridate** function. See the page on [Characters and strings](#characters-and-strings) for details on **stringr**.

In this example, the linelist data frame has a column in format “hours:minutes”. To convert this to a datetime we follow a few steps:

1. Create a “clean” time of admission column with missing values filled-in with the column median. We do this because **lubridate** won’t operate on missing values. Combine it with the column date\_hospitalisation, and then use the function ymd\_hm() to convert.

### Convert times alone

If your data contain only a character time (hours and minutes), you can convert and manipulate them as times using strptime() from **base** R. For example, to get the difference between two of these times:

Note however that without a date value provided, it assumes the date is today. To combine a string date and a string time together see how to use **stringr** in the section just above. Read more about strptime() [here](https://rdrr.io/r/base/strptime.html).

To convert single-digit numbers to double-digits (e.g. to “pad” hours or minutes with leading zeros to achieve 2 digits), see this [“Pad length” section of the Characters and strings page](#str_pad).

### Extract time

You can extract elements of a time with hour(), minute(), or second() from **lubridate**.

Here is an example of extracting the hour, and then classifing by part of the day. We begin with the column time\_admission, which is class Character in format “HH:MM”. First, the strptime() is used as described above to convert the characters to datetime class. Then, the hour is extracted with hour(), returning a number from 0-24. Finally, a column time\_period is created using logic with case\_when() to classify rows into Morning/Afternoon/Evening/Night based on their hour of admission.

To learn more about case\_when() see the page on [Cleaning data and core functions](#cleaning-data-and-core-functions).

## Working with dates

lubridate can also be used for a variety of other functions, such as **extracting aspects of a date/datetime**, **performing date arithmetic**, or **calculating date intervals**

Here we define a date to use for the examples:

### Extract date components

You can extract common aspects such as month, day, weekday:

You can also extract time components from a datetime object or column. This can be useful if you want to view the distribution of admission times.

There are several options to retrieve weeks. See the section on Epidemiological weeks below.

Note that if you are seeking to display a date a certain way (e.g. “Jan 2020” or “Thursday 20 March” or “Week 20, 1977”) you can do this more flexibly as described in the section on Date display.

### Date math

You can add certain numbers of days or weeks using their respective function from **lubridate**.

### Date intervals

The difference between dates can be calculated by:

1. Ensure both dates are of class date
2. Use subtraction to return the “difftime” difference between the two dates
3. If necessary, convert the result to numeric class to perform subsequent mathematical calculations

Below the interval between two dates is calculated and displayed. You can find intervals by using the subtraction “minus” symbol on values that are class Date. Note, however that the class of the returned value is “difftime” as displayed below, and must be converted to numeric.

To do subsequent operations on a “difftime”, convert it to numeric with as.numeric().

This can all be brought together to work with data - for example:

In a data frame context, if either of the above dates is missing, the operation will fail for that row. This will result in an NA instead of a numeric value. When using this column for calculations, be sure to set the na.rm = argument to TRUE. For example:

## Date display

Once dates are the correct class, you often want them to display differently, for example to display as “Monday 05 January” instead of “2018-01-05”. You may also want to adjust the display in order to then group rows by the date elements displayed - for example to group by month-year.

### format()

Adjust date display with the **base** R function format(). This function accepts a character string (in quotes) specifying the desired output format in the “%” strptime abbreviations (the same syntax as used in as.Date()). Below are most of the common abbreviations.

Note: using format() will convert the values to class Character, so this is generally used towards the end of an analysis or for display purposes only! You can see the complete list by running ?strptime.

%d = Day number of month (5, 17, 28, etc.)  
%j = Day number of the year (Julian day 001-366)  
%a = Abbreviated weekday (Mon, Tue, Wed, etc.)  
%A = Full weekday (Monday, Tuesday, etc.)  
%w = Weekday number (0-6, Sunday is 0)  
%u = Weekday number (1-7, Monday is 1)  
%W = Week number (00-53, Monday is week start)  
%U = Week number (01-53, Sunday is week start)  
%m = Month number (e.g. 01, 02, 03, 04)  
%b = Abbreviated month (Jan, Feb, etc.)  
%B = Full month (January, February, etc.)  
%y = 2-digit year (e.g. 89)  
%Y = 4-digit year (e.g. 1989)  
%h = hours (24-hr clock)  
%m = minutes  
%s = seconds  
%z = offset from GMT  
%Z = Time zone (character)

An example of formatting today’s date:

Note that if using str\_glue(), be aware of that within the expected double quotes " you should only use single quotes (as above).

### Month-Year

To convert a Date column to Month-year format, we suggest you use the function as.yearmon() from the **zoo** package. This converts the date to class “yearmon” and retains the proper ordering. In contrast, using format(column, "%Y %B") will convert to class Character and will order the values alphabetically (incorrectly).

Below, a new column yearmonth is created from the column date\_onset, using the as.yearmon() function. The default (correct) ordering of the resulting values are shown in the table.

In contrast, you can see how only using format() does achieve the desired display format, but not the correct ordering.

Note: if you are working within a ggplot() and want to adjust how dates are displayed only, it may be sufficient to provide a strptime format to the date\_labels = argument in scale\_x\_date() - you can use "%b %Y" or "%Y %b". See the [ggplot tips](#ggplot-tips) page.

**zoo** also offers the function as.yearqtr(), and you can use scale\_x\_yearmon() when using ggplot().

## Epidemiological weeks

### ****lubridate****

See the page on [Grouping data](#grouping-data) for more extensive examples of grouping data by date. Below we briefly describe grouping data by weeks.

We generally recommend using the floor\_date() function from **lubridate**, with the argument unit = "week". This rounds the date down to the “start” of the week, as defined by the argument week\_start =. The default week start is 1 (for Mondays) but you can specify any day of the week as the start (e.g. 7 for Sundays). floor\_date() is versitile and can be used to round down to other time units by setting unit = to “second”, “minute”, “hour”, “day”, “month”, or “year”.

The returned value is the start date of the week, in Date class. Date class is useful when plotting the data, as it will be easily recognized and ordered correctly by ggplot().

If you are only interested in adjusting dates to display by week in a plot, see the section in this page on Date display. For example when plotting an epicurve you can format the date display by providing the desired strptime “%” nomenclature. For example, use “%Y-%W” or “%Y-%U” to return the year and week number (given Monday or Sunday week start, respectively).

### Weekly counts

See the page on [Grouping data](#grouping-data) for a thorough explanation of grouping data with count(), group\_by(), and summarise(). A brief example is below.

1. Create a new ‘week’ column with mutate(), using floor\_date() with unit = "week"
2. Get counts of rows (cases) per week with count(); filter out any cases with missing date
3. Finish with complete() from **tidyr** to ensure that all weeks appear in the data - even those with no rows/cases. By default the count values for any “new” rows are NA, but you can make them 0 with the fill = argument, which expects a named list (below, n is the name of the counts column).

Here are the first rows of the resulting data frame:

### Epiweek alternatives

Note that **lubridate** also has functions week(), epiweek(), and isoweek(), each of which has slightly different start dates and other nuances. Generally speaking though, floor\_date() should be all that you need. Read the details for these functions by entering ?week into the console or reading the documentation [here](https://www.rdocumentation.org/packages/lubridate/versions/1.7.4/topics/week).

You might consider using the package **aweek** to set epidemiological weeks. You can read more about it [on the RECON website](https://www.repidemicsconsortium.org/aweek/). It has the functions date2week() and week2date() in which you can set the week start day with week\_start = "Monday". This package is easiest if you want “week”-style outputs (e.g. “2020-W12”). Another advantage of **aweek** is that when date2week() is applied to a date column, the returned column (week format) is automatically of class Factor and includes levels for all weeks in the time span (this avoids the extra step of complete() described above). However, **aweek** does not have the functionality to round dates to other time units such as months, years, etc.

Another alternative for time series which also works well to show a a “week” format (“2020 W12”) is yearweek() from the package **tsibble**, as demonstrated in the page on [Time series and outbreak detection](#time-series-and-outbreak-detection).

## Converting dates/time zones

When data is present in different time time zones, it can often be important to standardise this data in a unified time zone. This can present a further challenge, as the time zone component of data must be coded manually in most cases.

In R, each datetime object has a timezone component. By default, all datetime objects will carry the local time zone for the computer being used - this is generally specific to a location rather than a named timezone, as time zones will often change in locations due to daylight savings time. It is not possible to accurately compensate for time zones without a time component of a date, as the event a date column represents cannot be attributed to a specific time, and therefore time shifts measured in hours cannot be reasonably accounted for.

To deal with time zones, there are a number of helper functions in lubridate that can be used to change the time zone of a datetime object from the local time zone to a different time zone. Time zones are set by attributing a valid tz database time zone to the datetime object. A list of these can be found here - if the location you are using data from is not on this list, nearby large cities in the time zone are available and serve the same purpose.

<https://en.wikipedia.org/wiki/List_of_tz_database_time_zones>

This may seem largely abstract, and is often not needed if the user isn’t working across time zones.

## Lagging and leading calculations

lead() and lag() are functions from the **dplyr** package which help find previous (lagged) or subsequent (leading) values in a vector - typically a numeric or date vector. This is useful when doing calculations of change/difference between time units.

Let’s say you want to calculate the difference in cases between a current week and the previous one. The data are initially provided in weekly counts as shown below.

**When using lag() or lead() the order of rows in the dataframe is very important! - pay attention to whether your dates/numbers are ascending or descending**

First, create a new column containing the value of the previous (lagged) week.

* Control the number of units back/forward with n = (must be a non-negative integer)
* Use default = to define the value placed in non-existing rows (e.g. the first row for which there is no lagged value). By default this is NA.
* Use order\_by = TRUE if your the rows are not ordered by your reference column

Next, create a new column which is the difference between the two cases columns:

You can read more about lead() and lag() in the documentation [here](https://dplyr.tidyverse.org/reference/lead-lag.html) or by entering ?lag in your console.

## Resources

**lubridate** [tidyverse page](https://lubridate.tidyverse.org/)  
**lubridate** RStudio [cheatsheet](https://rawgit.com/rstudio/cheatsheets/master/lubridate.pdf)  
R for Data Science page on [dates and times](https://r4ds.had.co.nz/dates-and-times.html)  
[Online tutorial](https://www.statmethods.net/input/dates.html) [Date formats](https://www.r-bloggers.com/2013/08/date-formats-in-r/)

# Characters and strings

This page demonstrates use of the **stringr** package to evaluate and handle character values (“strings”).

1. Combine, order, split, arrange - str\_c(), str\_glue(), str\_order(), str\_split()
2. Clean and standardise
   * Adjust length - str\_pad(), str\_trunc(), str\_wrap()
   * Change case - str\_to\_upper(), str\_to\_title(), str\_to\_lower(), str\_to\_sentence()
3. Evaluate and extract by position - str\_length(), str\_sub(), word()
4. Patterns
   * Detect and locate - str\_detect(), str\_subset(), str\_match(), str\_extract()
   * Modify and replace - str\_sub(), str\_replace\_all()
5. Regular expressions (“regex”)

For ease of display most examples are shown acting on a short defined character vector, however they can easily be adapted to a column within a data frame.

This [stringr vignette](https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html) provided much of the inspiration for this page.

## Preparation

### Load packages

Install or load the **stringr** and other **tidyverse** packages.

### Import data

In this page we will occassionally reference the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Unite, split, and arrange

This section covers:

* Using str\_c(), str\_glue(), and unite() to combine strings
* Using str\_order() to arrange strings
* Using str\_split() and separate() to split strings

### Combine strings

To combine or concatenate multiple strings into one string, we suggest using str\_c from **stringr**. If you have distinct character values to combine, simply provide them as unique arguments, separated by commas.

The argument sep = inserts a character value between each of the arguments you provided (e.g. inserting a comma, space, or newline "\n")

The argument collapse = is relevant if you are inputting multiple vectors as arguments to str\_c(). It is used to separate the elements of what would be an output vector, such that the output vector only has one long character element.

The example below shows the combination of two vectors into one (first names and last names). Another similar example might be jurisdictions and their case counts. In this example:

* The sep = value appears between each first and last name
* The collapse = value appears between each person

Note: Depending on your desired display context, when printing such a combined string with newlines, you may need to wrap the whole phrase in cat() for the newlines to print properly:

### Dynamic strings

Use str\_glue() to insert dynamic R code into a string. This is a very useful function for creating dynamic plot captions, as demonstrated below.

* All content goes between double quotation marks str\_glue("")
* Any dynamic code or references to pre-defined values are placed within curly brackets {} within the double quotation marks. There can be many curly brackets in the same str\_glue() command.
* To display character quotes ’’, use single quotes within the surrounding double quotes (e.g. when providing date format - see example below)
* Tip: You can use \n to force a new line
* Tip: You use format() to adjust date display, and use Sys.Date() to display the current date

A simple example, of a dynamic plot caption:

An alternative format is to use placeholders within the brackets and define the code in separate arguments at the end of the str\_glue() function, as below. This can improve code readability if the text is long.

**Pulling from a data frame**

Sometimes, it is useful to pull data from a data frame and have it pasted together in sequence. Below is an example data frame. We will use it to to make a summary statement about the jurisdictions and the new and total case counts.

Use str\_glue\_data(), which is specially made for taking data from data frame rows:

**Combine strings across rows**

If you are trying to “roll-up” values in a data frame column, e.g. combine values from multiple rows into just one row by pasting them together with a separator, see the section of the [De-duplication](#de-duplication) page on [“rolling-up” values](#str_rollup).

**Data frame to one line**

You can make the statement appear in one line using str\_c() (specifying the data frame and column names), and providing sep = and collapse = arguments.

You could add the pre-fix text “New Cases:” to the beginning of the statement by wrapping with a separate str\_c() (if “New Cases:” was within the original str\_c() it would appear multiple times).

### Unite columns

Within a data frame, bringing together character values from multiple columns can be achieved with unite() from **tidyr**. This is the opposite of separate().

Provide the name of the new united column. Then provide the names of the columns you wish to unite.

* By default, the separator used in the united column is underscore \_, but this can be changed with the sep = argument.
* remove = removes the input columns from the data frame (TRUE by default)
* na.rm = removes missing values while uniting (FALSE by default)

Below, we define a mini-data frame to demonstrate with:

Here is the example data frame:

Below, we unite the three symptom columns:

### Split

To split a string based on a pattern, use str\_split(). It evaluates the string(s) and returns a list of character vectors consisting of the newly-split values.

The simple example below evaluates one string and splits it into three. By default it returns an object of class list with one element (a character vector) for each string initially provided. If simplify = TRUE it returns a character matrix.

In this example, one string is provided, and the function returns a list with one element - a character vector with three values.

If the output is saved, you can then access the nth split value with bracket syntax. To access a specific value you can use syntax like this: the\_returned\_object[[1]][2], which would access the second value from the first evaluated string (“fever”). See the [R basics](#r-basics) page for more detail on accessing elements.

If multiple strings are provided by str\_split(), there will be more than one element in the returned list.

To return a “character matrix” instead, which may be useful if creating data frame columns, set the argument simplify = TRUE as shown below:

You can also adjust the number of splits to create with the n = argument. For example, this restricts the number of splits to 2. Any further commas remain within the second values.

Note - the same outputs can be achieved with *str\_split\_fixed()*, in which you do not give the *simplify* argument, but must instead designate the number of columns (*n*).

### Split columns

If you are trying to split data frame column, it is best to use the separate() function from **dplyr**. It is used to split one character column into other columns.

Let’s say we have a simple data frame df (defined and united in the [unite section](#str_unite)) containing a case\_ID column, one character column with many symptoms, and one outcome column. Our goal is to separate the symptoms column into many columns - each one containing one symptom.

Assuming the data are piped into separate(), first provide the column to be separated. Then provide into = as a vector c( ) containing the new columns names, as shown below.

* sep = the separator, can be a character, or a number (interpreted as the character position to split at)
* remove = FALSE by default, removes the input column
* convert = FALSE by default, will cause string “NA”s to become NA
* extra = this controls what happens if there are more values created by the separation than new columns named.
  + extra = "warn" means you will see a warning but it will drop excess values (**the default**)
  + extra = "drop" means the excess values will be dropped with no warning
  + **extra = "merge" will only split to the number of new columns listed in into - this setting will preserve all your data**

An example with extra = "merge" is below - no data is lost. Two new columns are defined but any third symptoms are left in the second new column:

When the default extra = "drop" is used below, a warning is given but the third symptoms are lost:

**CAUTION:** If you do not provide enough into values for the new columns, your data may be truncated.

### Arrange alphabetically

Several strings can be sorted by alphabetical order. str\_order() returns the order, while str\_sort() returns the strings in that order.

To use a different alphabet, add the argument locale =. See the full list of locales by entering stringi::stri\_locale\_list() in the R console.

### base R functions

It is common to see **base** R functions paste() and paste0(), which concatenate vectors after converting all parts to character. They act similarly to str\_c() but the syntax is arguably more complicated - in the parentheses each part is separated by a comma. The parts are either character text (in quotes) or pre-defined code objects (no quotes). For example:

sep = and collapse = arguments can be specified. paste() is simply paste0() with a default sep = " " (one space).

## Clean and standardise

### Change case

Often one must alter the case/capitalization of a string value, for example names of jursidictions. Use str\_to\_upper(), str\_to\_lower(), and str\_to\_title(), from **stringr**, as shown below:

Using \*base\*\* R, the above can also be achieved with toupper(), tolower().

**Title case**

Transforming the string so each word is capitalized can be achieved with str\_to\_title():

Use toTitleCase() from the **tools** package to achieve more nuanced capitalization (words like “to”, “the”, and “of” are not capitalized).

You can also use str\_to\_sentence(), which capitalizes only the first letter of the string.

### Pad length

Use str\_pad() to add characters to a string, to a minimum length. By default spaces are added, but you can also pad with other characters using the pad = argument.

For example, to pad numbers with leading zeros (such as for hours or minutes), you can pad the number to minimum length of 2 with pad = "0".

### Truncate

str\_trunc() sets a maximum length for each string. If a string exceeds this length, it is truncated (shortened) and an ellipsis (…) is included to indicate that the string was previously longer. Note that the ellipsis is counted in the length. The ellipsis characters can be changed with the argument ellipsis =. The optional side = argument specifies which where the ellipsis will appear within the truncated string (“left”, “right”, or “center”).

### Standardize length

Use str\_trunc() to set a maximum length, and then use str\_pad() to expand the very short strings to that truncated length. In the example below, 6 is set as the maximum length (one value is truncated), and then one very short value is padded to achieve length of 6.

### Remove leading/trailing whitespace

Use str\_trim() to remove spaces, newlines (\n) or tabs (\t) on sides of a string input. Add "right" "left", or "both" to the command to specify which side to trim (e.g. str\_trim(x, "right").

### Remove repeated whitespace within

Use str\_squish() to remove repeated spaces that appear inside a string. For example, to convert double spaces into single spaces. It also removes spaces, newlines, or tabs on the outside of the string like str\_trim().

Enter ?str\_trim, ?str\_pad in your R console to see further details.

### Wrap into paragraphs

Use str\_wrap() to wrap a long unstructured text into a structured paragraph with fixed line length. Provide the ideal character length for each line, and it applies an algorithm to insert newlines (\n) within the paragraph, as seen in the example below.

The **base** function cat() can be wrapped around the above command in order to print the output, displaying the new lines added.

## Handle by position

### Extract by character position

Use str\_sub() to return only a part of a string. The function takes three main arguments:

1. the character vector(s)
2. start position
3. end position

A few notes on position numbers:

* If a position number is positive, the position is counted starting from the left end of the string.
* If a position number is negative, it is counted starting from the right end of the string.
* Position numbers are inclusive.
* Positions extending beyond the string will be truncated (removed).

Below are some examples applied to the string “pneumonia”:

### Extract by word position

To extract the nth ‘word’, use word(), also from **stringr**. Provide the string(s), then the first word position to extract, and the last word position to extract.

By default, the separator between ‘words’ is assumed to be a space, unless otherwise indicated with sep = (e.g. sep = "\_" when words are separated by underscores.

### Replace by character position

str\_sub() paired with the assignment operator (<-) can be used to modify a part of a string:

An example applied to multiple strings (e.g. a column). Note the expansion in length of “HIV”.

### Evaluate length

Alternatively, use nchar() from **base** R

## Patterns

Many **stringr** functions work to detect, locate, extract, match, replace, and split based on a specified pattern.

### Detect a pattern

Use str\_detect() as below to detect presence/absence of a pattern within a string. First provide the string or vector to search in (string =), and then the pattern to look for (pattern =). Note that by default the search is case sensitive!

The argument negate = can be included and set to TRUE if you want to know if the pattern is NOT present.

To ignore case/capitalization, wrap the pattern within regex(), and within regex() add the argument ignore\_case = TRUE (or T as shorthand).

When str\_detect() is applied to a character vector or a data frame column, it will return TRUE or FALSE for each of the values.

If you need to count the TRUEs, simply sum() the output. This counts the number TRUE.

To search inclusive of multiple terms, include them separated by OR bars (|) within the pattern = argument, as shown below:

If you need to build a long list of search terms, you can combine them using str\_c() and sep = |, then define this is a character object, and then reference the vector later more succinctly. The example below includes possible occupation search terms for front-line medical providers.

This command returns the number of occupations which contain any one of the search terms for front-line medical providers (occupation\_med\_frontline):

**Base R string search functions**

The **base** function grepl() works similarly to str\_detect(), in that it searches for matches to a pattern and returns a logical vector. The basic syntax is grepl(pattern, strings\_to\_search, ignore.case = FALSE, ...). One advantage is that the ignore.case argument is easier to write (there is no need to involve the regex() function).

Likewise, the **base** functions sub() and gsub() act similarly to str\_replace(). Their basic syntax is: gsub(pattern, replacement, strings\_to\_search, ignore.case = FALSE). sub() will replace the first instance of the pattern, whereas gsub() will replace all instances of the pattern.

#### Convert commas to periods

Here is an example of using gsub() to convert commas to periods in a vector of numbers. This could be useful if your data come from parts of the world other than the United States or Great Britain.

The inner gsub() which acts first on lengths is converting any periods to no space "“. The period character”." has to be “escaped” with two slashes to actually signify a period, because “.” in regex means “any character”. Then, the result (with only commas) is passed to the outer gsub() in which commas are replaced by periods.

### Replace all

Use str\_replace\_all() as a “find and replace” tool. First, provide the strings to be evaluated to string =, then the pattern to be replaced to pattern =, and then the replacement value to replacement =. The example below replaces all instances of “dead” with “deceased”. Note, this IS case sensitive.

Notes:

* To replace a pattern with NA, use str\_replace\_na().
* The function str\_replace() replaces only the first instance of the pattern within each evaluated string.

### Detect within logic

**Within case\_when()**

str\_detect() is often used within case\_when() (from **dplyr**). Let’s say occupations is a column in the linelist. The mutate() below creates a new column called is\_educator by using conditional logic via case\_when(). See the page on data cleaning to learn more about case\_when().

As a reminder, it may be important to add exclusion criteria to the conditional logic (negate = F):

### Locate pattern position

To locate the first position of a pattern, use str\_locate(). It outputs a start and end position.

Like other str functions, there is an "\_all" version (str\_locate\_all()) which will return the positions of all instances of the pattern within each string. This outputs as a list.

### Extract a match

str\_extract\_all() returns the matching patterns themselves, which is most useful when you have offered several patterns via “OR” conditions. For example, looking in the string vector of occupations (see previous tab) for either “teach”, “prof”, or “tutor”.

str\_extract\_all() returns a list which contains all matches for each evaluated string. See below how occupation 3 has two pattern matches within it.

str\_extract() extracts only the first match in each evaluated string, producing a character vector with one element for each evaluated string. It returns NA where there was no match. The NAs can be removed by wrapping the returned vector with na.exclude(). Note how the second of occupation 3’s matches is not shown.

### Subset and count

Aligned functions include str\_subset() and str\_count().

str\_subset() returns the actual values which contained the pattern:

str\_count() returns a vector of numbers: the **number of times** a search term appears in each evaluated value.

### Regex groups

UNDER CONSTRUCTION

## Special characters

**Backslash \ as escape**

The backslash \ is used to “escape” the meaning of the next character. This way, a backslash can be used to have a quote mark display within other quote marks (\") - the middle quote mark will not “break” the surrounding quote marks.

Note - thus, if you want to display a backslash, you must escape it’s meaning with another backslash. So you must write two backslashes \\ to display one.

**Special characters**

| **Special character** | **Represents** |
| --- | --- |
| "\\" | backslash |
| "\n" | a new line (newline) |
| "\"" | double-quote within double quotes |
| '\'' | single-quote within single quotes |
| "\“| grave accent”| carriage return“| tab”| vertical tab"` | backspace |

Run ?"'" in the R Console to display a complete list of these special characters (it will appear in the RStudio Help pane).

## Regular expressions (regex)

## Regex and special characters

Regular expressions, or “regex”, is a concise language for describing patterns in strings. If you are not familiar with it, a regular expression can look like an alien language. Here we try to de-mystify this language a little bit.

Much of this section is adapted from [*this tutorial*](https://towardsdatascience.com/a-gentle-introduction-to-regular-expressions-with-r-df5e897ca432) and [*this cheatsheet*](https://evoldyn.gitlab.io/evomics-2018/ref-sheets/R_strings.pdf). We selectively adapt here knowing that this handbook might be viewed by people without internet access to view the other tutorials.

A regular expression is often applied to extract specific patterns from “unstructured” text - for example medical notes, chief complaints, patient history, or other free text columns in a data frame

There are four basic tools one can use to create a basic regular expression:

1. Character sets
2. Meta characters
3. Quantifiers
4. Groups

**Character sets**

Character sets, are a way of expressing listing options for a character match, within brackets. So any a match will be triggered if any of the characters within the brackets are found in the string. For example, to look for vowels one could use this character set: “[aeiou]”. Some other common character sets are:

| **Character set** | **Matches for** |
| --- | --- |
| "[A-Z]" | any single capital letter |
| "[a-z]" | any single lowercase letter |
| "[0-9]" | any digit |
| [:alnum:] | any alphanumeric character |
| [:digit:] | any numeric digit |
| [:alpha:] | any letter (upper or lowercase) |
| [:upper:] | any uppercase letter |
| [:lower:] | any lowercase letter |

Character sets can be combined within one bracket (no spaces!), such as "[A-Za-z]" (any upper or lowercase letter), or another example "[t-z0-5]" (lowercase t through z OR number 0 through 5).

**Meta characters**

Meta characters are shorthand for character sets. Some of the important ones are listed below:

| **Meta character** | **Represents** |
| --- | --- |
| "\\s" | a single space |
| "\\w" | any single alphanumeric character (A-Z, a-z, or 0-9) |
| "\\d" | any single numeric digit (0-9) |

**Quantifiers**

Typically you do not want to search for a match on only one character. Quantifiers allow you to designate the length of letters/numbers to allow for the match.

Quantifiers are numbers written within curly brackets { } after the character they are quantifying, for example,

* "A{2}" will return instances of **two** capital A letters.
* "A{2,4}" will return instances of **between two and four** capital A letters (do not put spaces!).
* "A{2,}" will return instances of **two or more** capital A letters.
* "A+" will return instances of **one or more** capital A letters (group extended until a different character is encountered).
* Precede with an \* asterisk to return **zero or more** matches (useful if you are not sure the pattern is present)

Using the + plus symbol as a quantifier, the match will occur until a different character is encountered. For example, this expression will return all words (alpha characters: "[A-Za-z]+"

When a quantifier of {2} is used, only pairs of consecutive A’s are returned. Two pairs are identified within AAAA.

When a quantifier of {2,4} is used, groups of consecutive A’s that are two to four in length are returned.

With the quantifier +, groups of **one or more** are returned:

**Relative position**

These express requirements for what precedes or follows a pattern. For example, to extract sentences, “two numbers that are followed by a period” (""). (?<=\.)\s(?=[A-Z])

| **Position statement** | **Matches to** |
| --- | --- |
| "(?<=b)a" | “a” that **is preceded** by a “b” |
| "(?<!b)a" | “a” that **is NOT preceded** by a “b” |
| "a(?=b)" | “a” that **is followed** by a “b” |
| "a(?!b)" | “a” that **is NOT followed** by a “b” |

**Groups**

Capturing groups in your regular expression is a way to have a more organized output upon extraction.

**Regex examples**

Below is a free text for the examples. We will try to extract useful information from it using a regular expression search term.

This expression matches to all words (any character until hitting non-character such as a space):

The expression "[0-9]{1,2}" matches to consecutive numbers that are 1 or 2 digits in length. It could also be written "\\d{1,2}", or "[:digit:]{1,2}".

You can view a useful list of regex expressions and tips on page 2 of [this cheatsheet](https://evoldyn.gitlab.io/evomics-2018/ref-sheets/R_strings.pdf)

Also see this [tutorial](https://towardsdatascience.com/a-gentle-introduction-to-regular-expressions-with-r-df5e897ca432).

## Resources

A reference sheet for **stringr** functions can be found [here](https://evoldyn.gitlab.io/evomics-2018/ref-sheets/R_strings.pdf)

A vignette on **stringr** can be found [here](https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html)

# Factors

In R, factors are a class of data that allow for ordered categories with a fixed set of acceptable values.

Typically, you would convert a column from character or numeric class to a factor if you want to set an intrinsic order to the values (“levels”) so they can be displayed non-alphabetically in plots and tables. Another common use of factors is to standardise the legends of plots so they do not fluctuate if certain values are temporarily absent from the data.

This page demonstrates use of functions from the package **forcats** (a short name for “**For** **cat**egorical variables”) and some **base** R functions. We also touch upon the use of **lubridate** and **aweek** for special factor cases related to epidemiological weeks.

A complete list of **forcats** functions can be found online [here](https://forcats.tidyverse.org/reference/index.html). Below we demonstrate some of the most common ones.

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

### New categorical variable

For demonstration in this page we will use a common scenario - the creation of a new categorical variable.

Note that if you convert a numeric column to class factor, you will not be able to calculate numeric statistics on it.

#### Create column

We use the existing column days\_onset\_hosp (days from symptom onset to hospital admission) and create a new column delay\_cat by classifying each row into one of several categories. We do this with the **dplyr** function case\_when(), which sequentially applies logical criteria (right-side) to each row and returns the corresponding left-side value for the new column delay\_cat. Read more about case\_when() in [Cleaning data and core functions](#cleaning-data-and-core-functions).

#### Default value order

As created with case\_when(), the new column delay\_cat is a categorical column of class Character - not yet a factor. Thus, in a frequency table, we see that the unique values appear in a default alpha-numeric order - an order that does not make much intuitive sense:

Likewise, if we make a bar plot, the values also appear in this order on the x-axis (see the [ggplot basics](#ggplot-basics) page for more on **ggplot2** - the most common visualization package in R).

## Convert to factor

To convert a character or numeric column to class factor, you can use any function from the **forcats** package (many are detailed [below](#fct_adjust)). They will convert to class factor and then also perform or allow certain ordering of the levels - for example using fct\_relevel() lets you manually specify the level order. The function as\_factor() simply converts the class without any further capabilities.

The **base** R function factor() converts a column to factor and allows you to manually specify the order of the levels, as a character vector to its levels = argument.

Below we use mutate() and fct\_relevel() to convert the column delay\_cat from class character to class factor. The column delay\_cat is created in the [Preparation](#fct_newcat) section above.

The unique “values” in this column are now considered “levels” of the factor. The levels have an order, which can be printed with the **base** R function levels(), or alternatively viewed in a count table via table() from **base** R or tabyl() from **janitor**. By default, the order of the levels will be alpha-numeric, as before. Note that NA is not a factor level.

The function fct\_relevel() has the additional utility of allowing you to manually specify the level order. Simply write the level values in order, in quotation marks, separated by commas, as shown below. Note that the spelling must exactly match the values. If you want to create levels that do not exist in the data, use [fct\_expand() instead](#fct_add)).

We can now see that the levels are ordered, as specified in the previous command, in a sensible order.

Now the plot order makes more intuitive sense as well.

## Add or drop levels

### Add

If you need to add levels to a factor, you can do this with fct\_expand(). Just write the column name followed by the new levels (separated by commas). By tabulating the values, we can see the new levels and the zero counts. You can use table() from **base** R, or tabyl() from **janitor**:

Note: there is a special **forcats** function to easily add missing values (NA) as a level. See the section on [Missing values](#fct_missing) below.

### Drop

If you use fct\_drop(), the “unused” levels with zero counts will be dropped from the set of levels. The levels we added above (“Not admitted to a hospital”) exists as a level but no rows actually have those values. So they will be dropped by applying fct\_drop() to our factor column:

## Adjust level order

The package **forcats** offers useful functions to easily adjust the order of a factor’s levels (after a column been defined as class factor):

These functions can be applied to a factor column in two contexts:

1. To the column in the data frame, as usual, so the transformation is available for any subsequent use of the data
2. Inside of a plot, so that the change is applied only within the plot

### Manually

This function is used to manually order the factor levels. If used on a non-factor column, the column will first be converted to class factor.

Within the parentheses first provide the factor column name, then provide either:

* All the levels in the desired order (as a character vector c()), or
* One level and it’s corrected placement using the after = argument

Here is an example of redefining the column delay\_cat (which is already class Factor) and specifying all the desired order of levels.

If you only want to move one level, you can specify it to fct\_relevel() alone and give a number to the after = argument to indicate where in the order it should be. For example, the command below shifts “<2 days” to the second position:

### Within a plot

The **forcats** commands can be used to set the level order in the data frame, or only within a plot. By using the command to “wrap around” the column name within the ggplot() plotting command, you can reverse/relevel/etc. the transformation will only apply within that plot.

Below, two plots are created with ggplot() (see the [ggplot basics](#ggplot-basics) page). In the first, the delay\_cat column is mapped to the x-axis of the plot, with it’s default level order as in the data linelist. In the second example it is wrapped within fct\_relevel() and the order is changed in the plot.

Note that default x-axis title is now quite complicated - you can overwrite this title with the **ggplot2** labs() argument.

### Reverse

It is rather common that you want to reverse the level order. Simply wrap the factor with fct\_rev().

Note that if you want to reverse only a plot legend but not the actual factor levels, you can do that with guides() (see [ggplot tips](#ggplot-tips)).

### By frequency

To order by frequency that the value appears in the data, use fct\_infreq(). Any missing values (NA) will automatically be included at the end, unless they are converted to an explicit level (see [this section](#fct_missing)). You can reverse the order by further wrapping with fct\_rev().

This function can be used within a ggplot(), as shown below.

### By appearance

Use fct\_inorder() to set the level order to match the order of appearance in the data, starting from the first row. This can be useful if you first carefully arrange() the data in the data frame, and then use this to set the factor order.

### By summary statistic of another column

You can use fct\_reorder() to order the levels of one column by a summary statistic of another column. Visually, this can result in pleasing plots where the bars/points ascend or descend steadily across the plot.

In the examples below, the x-axis is delay\_cat, and the y-axis is numeric column ct\_blood (cycle-threshold value). Box plots show the CT value distribution by delay\_cat group. We want to order the box plots in ascending order by the group median CT value.

In the first example below, the default order alpha-numeric level order is used. You can see the box plot heights are jumbled and not in any particular order. In the second example, the delay\_cat column (mapped to the x-axis) has been wrapped in fct\_reorder(), the column ct\_blood is given as the second argument, and “median” is given as the third argument (you could also use “max”, “mean”, “min”, etc). Thus, the order of the levels of delay\_cat will now reflect ascending median CT values of each delay\_cat group’s median CT value. This is reflected in the second plot - the box plots have been re-arranged to ascend. Note how NA (missing) will appear at the end, unless converted to an explicit level.

Note in this example above there are no steps required prior to the ggplot() call - the grouping and calculations are all done internally to the ggplot command.

### By “end” value

Use fct\_reorder2() for grouped line plots. It orders the levels (and therefore the legend) to align with the vertical ordering of the lines at the “end” of the plot. Technically speaking, it “orders by the y-values associated with the largest x values.”

For example, if you have lines showing case counts by hospital over time, you can apply fct\_reorder2() to the color = argument within aes(), such that the vertical order of hospitals appearing in the legend aligns with the order of lines at the terminal end of the plot. Read more in the [online documentation](https://forcats.tidyverse.org/reference/fct_reorder.html).

## Missing values

If you have NA values in your factor column, you can easily convert them to a named level such as “Missing” with fct\_explicit\_na(). The NA values are converted to “(Missing)” at the end of the level order by default. You can adjust the level name with the argument na\_level =.

Below, this opertation is performed on the column delay\_cat and a table is printed with tabyl() with NA converted to “Missing delay”.

## Combine levels

### Manually

You can adjust the level displays manually manually with fct\_recode(). This is like the **dplyr** function recode() (see [Cleaning data and core functions](#cleaning-data-and-core-functions)), but it allows the creation of new factor levels. If you use the simple recode() on a factor, new re-coded values will be rejected unless they have already been set as permissible levels.

This tool can also be used to “combine” levels, by assigning multiple levels the same re-coded value. Just be careful to not lose information! Consider doing these combining steps in a new column (not over-writing the existing column).

fct\_recode() has a different syntax than recode(). recode() uses OLD = NEW, whereas fct\_recode() uses NEW = OLD.

The current levels of delay\_cat are:

The new levels are created using syntax fct\_recode(column, "new" = "old", "new" = "old", "new" = "old") and printed:

Here they are manually combined with fct\_recode(). Note there is no error raised at the creation of a new level “Less than 5 days”.

### Reduce into “Other”

You can use fct\_other() to manually assign factor levels to an “Other” level. Below, all levels in the column hospital, aside from “Port Hospital” and “Central Hospital”, are combined into “Other”. You can provide a vector to either keep =, or drop =. You can change the display of the “Other” level with other\_level =.

### Reduce by frequency

You can combine the least-frequent factor levels automatically using fct\_lump().

To “lump” together many low-frequency levels into an “Other” group, do one of the following:

* Set n = as the number of groups you want to keep. The n most-frequent levels will be kept, and all others will combine into “Other”.
* Set prop = as the threshold frequency proportion for levels above which you want to keep. All other values will combine into “Other”.

You can change the display of the “Other” level with other\_level =. Below, all but the two most-frequent hospitals are combined into “Other Hospital”.

, warn ## Show all levels

One benefit of using factors is to standardise the appearance of plot legends and tables, regardless of which values are actually present in a dataset.

If you are preparing many figures (e.g. for multiple jurisdictions) you will want the legends and tables to appear identically even with varying levels of data completion or data composition.

### In plots

In a ggplot() figure, simply add the argument drop = FALSE in the relevant scale\_xxxx() function. All factor levels will be displayed, regardless of whether they are present in the data. If your factor column levels are displayed using fill =, then in scale\_fill\_discrete() you include drop = FALSE, as shown below. If your levels are displayed with x = (to the x-axis) color = or size = you would provide this to scale\_color\_discrete() or scale\_size\_discrete() accordingly.

This example is a stacked bar plot of age category, by hospital. Adding scale\_fill\_discrete(drop = FALSE) ensures that all age groups appear in the legend, even if not present in the data.

### In tables

Both the **base** R table() and tabyl() from **janitor** will show all factor levels (even unused levels).

If you use count() or summarise() from **dplyr** to make a table, add the argument .drop = FALSE to include counts for all factor levels even those unused.

Read more in the [Descriptive tables](#descriptive-tables) page, or at the [scale\_discrete documentation](https://ggplot2.tidyverse.org/reference/scale_discrete.html), or the [count() documentation](https://dplyr.tidyverse.org/reference/count.html). You can see another example in the [Contact tracing](#contact-tracing-1) page.

## Epiweeks

Please see the extensive discussion of how to create epidemiological weeks in the [Grouping data](#grouping-data) page.  
Please also see the [Working with dates](#working-with-dates-1) page for tips on how to create and format epidemiological weeks.

### Epiweeks in a plot

If your goal is to create epiweeks to display in a plot, you can do this simply with **lubridate**’s floor\_date(), as explained in the [Grouping data](#grouping-data) page. The values returned will be of class Date with format YYYY-MM-DD. If you use this column in a plot, the dates will naturally order correctly, and you do not need to worry about levels or converting to class Factor. See the ggplot() histogram of onset dates below.

In this approach, you can adjust the display of the dates on an axis with scale\_x\_date(). See the page on [Epidemic curves](#epidemic-curves) for more information. You can specify a “strptime” display format to the date\_labels = argument of scale\_x\_date(). These formats use “%” placeholders and are covered in the [Working with dates](#working-with-dates-1) page. Use “%Y” to represent a 4-digit year, and either “%W” or “%U” to represent the week number (Monday or Sunday weeks respectively).

### Epiweeks in the data

However, if your purpose in factoring is not to plot, you can approach this one of two ways:

1. For fine control over the display, convert the **lubridate** epiweek column (YYYY-MM-DD) to the desired display format (YYYY-WWw) within the data frame itself, and then convert it to class Factor.

First, use format() from **base** R to convert the date display from YYYY-MM-DD to YYYY-Www display (see the [Working with dates](#working-with-dates-1) page). In this process the class will be converted to character. Then, convert from character to class Factor with factor().

**DANGER:** If you place the weeks ahead of the years (“Www-YYYY”) (“%W-%Y”), the default alpha-numeric level ordering will be incorrect (e.g. 01-2015 will be before 35-2014). You could need to manually adjust the order, which would be a long painful process.

1. For fast default display, use the **aweek** package and it’s function date2week(). You can set the week\_start = day, and if you set factor = TRUE then the output column is an ordered factor. As a bonus, the factor includes levels for all possible weeks in the span - even if there are no cases that week.

See the [Working with dates](#working-with-dates-1) page for more information about **aweek**. It also offers the reverse function week2date().

## Resources

R for Data Science page on [factors](https://r4ds.had.co.nz/factors.html)  
[aweek package vignette](https://cran.r-project.org/web/packages/aweek/vignettes/introduction.html)

# Pivoting data

When managing data, pivoting can be understood to refer to one of two processes:

1. The creation of pivot tables, which are tables of statistics that summarise the data of a more extensive table
2. The conversion of a table from **long** to **wide** format, or vice versa.

**In this page, we will focus on the latter definition.** The former is a crucial step in data analysis, and is covered elsewhere in the [Grouping data](#grouping-data) and [Descriptive tables](#descriptive-tables) pages.

This page discusses the formats of data. It is useful to be aware of the idea of “tidy data”, in which each variable has it’s own column, each observation has it’s own row, and each value has it’s own cell. More about this topic can be found [at this online chapter in R for Data Science](https://r4ds.had.co.nz/tidy-data.html).

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

### Malaria count data

In this page, we will use a fictional dataset of daily malaria cases, by facility and age group. If you want to follow along, [click here to download (as .rds file)](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/malaria_facility_count_data.rds). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows are displayed below.

### Linelist case data

In the later part of this page, we will also use the dataset of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

## Wide-to-long

### “Wide” format

Data are often entered and stored in a “wide” format - where a subject’s characteristics or responses are stored in a single row. While this may be useful for presentation, it is not ideal for some types of analysis.

Let us take the count\_data dataset imported in the Preparation section above as an example. You can see that each row represents a “facility-day”. The actual case counts (the right-most columns) are stored in a “wide” format such that the information for every age group on a given facility-day is stored in a single row.

Each observation in this dataset refers to the malaria counts at one of 65 facilities on a given date, ranging from count\_data$data\_date %>% min() to count\_data$data\_date %>% max(). These facilities are located in one Province (North) and four Districts (Spring, Bolo, Dingo, and Barnard). The dataset provides the overall counts of malaria, as well as age-specific counts in each of three age groups - <4 years, 5-14 years, and 15 years and older.

“Wide” data like this are not adhering to “tidy data” standards, because the column headers do not actually represent “variables” - they represent values of a hypothetical “age group” variable.

This format can be useful for presenting the information in a table, or for entering data (e.g. in Excel) from case report forms. However, in the analysis stage, these data typically should be transformed to a “longer” format more aligned with “tidy data” standards. The plotting R package **ggplot2** in particular works best when data are in a “long” format.

Visualising the total malaria counts over time poses no difficulty with the data in it’s current format:

However, what if we wanted to display the relative contributions of each age group to this total count? In this case, we need to ensure that the variable of interest (age group), appears in the dataset in a single column that can be passed to {ggplot2}’s “mapping aesthetics” aes() argument.

### pivot\_longer()

The **tidyr** function pivot\_longer() makes data “longer”. **tidyr** is part of the **tidyverse** of R packages.

It accepts a range of columns to transform (specified to cols =). Therefore, it can operate on only a part of a dataset. This is useful for the malaria data, as we only want to pivot the case count columns.

In this process, you will end up with two “new” columns - one with the categories (the former column names), and one with the corresponding values (e.g. case counts). You can accept the default names for these new columns, or you can specify your own to names\_to = and values\_to = respectively.

Let’s see pivot\_longer() in action…

### Standard pivoting

We want to use **tidyr**’s pivot\_longer() function to convert the “wide” data to a “long” format. Specifically, to convert the four numeric columns with data on malaria counts to two new columns: one which holds the age groups and one which holds the corresponding values.

Notice that the newly created data frame (df\_long) has more rows (12,152 vs 3,038); it has become longer. In fact, it is precisely four times as long, because each row in the original dataset now represents four rows in df\_long, one for each of the malaria count observations (<4y, 5-14y, 15y+, and total).

In addition to becoming longer, the new dataset has fewer columns (8 vs 10), as the data previously stored in four columns (those beginning with the prefix malaria\_) is now stored in two.

Since the names of these four columns all begin with the prefix malaria\_, we could have made use of the handy “tidyselect” function starts\_with() to achieve the same result (see the page [Cleaning data and core functions](#cleaning-data-and-core-functions) for more of these helper functions).

or by position:

or by named range:

These two new columns are given the default names of name and value, but we can override these defaults to provide more meaningful names, which can help remember what is stored within, using the names\_to and values\_to arguments. Let’s use the names age\_group and counts:

We can now pass this new dataset to {ggplot2}, and map the new column count to the y-axis and new column age\_group to the fill = argument (the column internal color). This will display the malaria counts in a stacked bar chart, by age group:

Examine this new plot, and compare it with the plot we created earlier - what has gone wrong?

We have encountered a common problem when wrangling surveillance data - we have also included the total counts from the malaria\_tot column, so the magnitude of each bar in the plot is twice as high as it should be.

We can handle this in a number of ways. We could simply filter these totals from the dataset before we pass it to ggplot():

Alternatively, we could have excluded this variable when we ran pivot\_longer(), thereby maintaining it in the dataset as a separate variable. See how its values “expand” to fill the new rows.

### Pivoting data of multiple classes

The above example works well in situations in which all the columns you want to “pivot longer” are of the same class (character, numeric, logical…).

However, there will be many cases when, as a field epidemiologist, you will be working with data that was prepared by non-specialists and which follow their own non-standard logic - as Hadley Wickham noted (referencing Tolstoy) in his [seminal article](https://vita.had.co.nz/papers/tidy-data.pdf) on **Tidy Data** principles: “Like families, tidy datasets are all alike but every messy dataset is messy in its own way.”

One particularly common problem you will encounter will be the need to pivot columns that contain different classes of data. This pivot will result in storing these different data types in a single column, which is not a good situation. There are various approaches one can take to separate out the mess this creates, but there is an important step you can take using pivot\_longer() to avoid creating such a situation yourself.

Take a situation in which there have been a series of observations at different time steps for each of three items A, B and C. Examples of such items could be individuals (e.g. contacts of an Ebola case being traced each day for 21 days) or remote village health posts being monitored once per year to ensure they are still functional. Let’s use the contact tracing example. Imagine that the data are stored as follows:

As can be seen, the data are a bit complicated. Each row stores information about one item, but with the time series running further and further away to the right as time progresses. Moreover, the column classes alternate between date and character values.

One particularly bad example of this encountered by this author involved cholera surveillance data, in which 8 new columns of observations were added each day over the course of **4 years**. Simply opening the Excel file in which these data were stored took >10 minuntes on my laptop!

In order to work with these data, we need to transform the data frame to long format, but keeping the separation between a date column and a character (status) column, for each observation for each item. If we don’t, we might end up with a mixture of variable types in a single column (a very big “no-no” when it comes to data management and tidy data):

Above, our pivot has merged dates and characters into a single value column. R will react by converting the entire column to class character, and the utility of the dates is lost.

To prevent this situation, we can take advantage of the syntax structure of the original column names. There is a common naming structure, with the observation number, an underscore, and then either “status” or “date”. We can leverage this syntax to keep these two data types in separate columns after the pivot.

We do this by:

* Providing a character vector to the names\_to = argument, with the second item being (".value" ). This special term indicates that the pivoted columns will be split based on a character in their name…
* You must also provide the “splitting” character to the names\_sep = argument. In this case, it is the underscore "\_".

Thus, the naming and split of new columns is based around the underscore in the existing variable names.

**Finishing touches**:

Note that the date column is currently in character class - we can easily convert this into it’s proper date class using the mutate() and as\_date() functions described in the [Working with dates](#working-with-dates-1) page.

We may also want to convert the observation column to a numeric format by dropping the “obs” prefix and converting to numeric. We cando this with str\_remove\_all() from the **stringr** package (see the [Characters and strings](#characters-and-strings) page).

And now, we can start to work with the data in this format, e.g. by plotting a descriptive heat tile:

## Long-to-wide

In some instances, we may wish to convert a dataset to a wider format. For this, we can use the pivot\_wider() function.

A typical use-case is when we want to transform the results of an analysis into a format which is more digestible for the reader (such as a [Table for presentation](#tables-for-presentation)). Usually, this involves transforming a dataset in which information for one subject is are spread over multiple rows into a format in which that information is stored in a single row.

### Data

For this section of the page, we will use the case linelist (see the [Preparation](#pivot_prep) section), which contains one row per case.

Here are the first 50 rows:

Suppose that we want to know the counts of individuals in the different age groups, by gender:

This gives us a long dataset that is great for producing visualisations in **ggplot2**, but not ideal for presentation in a table:

### Pivot wider

Therefore, we can use pivot\_wider() to transform the data into a better format for inclusion as tables in our reports.

The argument names\_from specifies the column from which to generate the new column names, while the argument values\_from specifies the column from which to take the values to populate the cells. The argument id\_cols = is optional, but can be provided a vector of column names that should not be pivoted, and will thus identify each row.

This table is much more reader-friendly, and therefore better for inclusion in our reports. You can convert into a pretty table with several packages including **flextable** and **knitr**. This process is elaborated in the page [Tables for presentation](#tables-for-presentation).

## Fill

In some situations after a pivot, and more commonly after a bind, we are left with gaps in some cells that we would like to fill.

### Data

For example, take two datasets, each with observations for the measurement number, the name of the facility, and the case count at that time. However, the second dataset also has a variable Year.

When we perform a bind\_rows() to join the two datasets together, the Year variable is filled with NA for those rows where there was no prior information (i.e. the first dataset):

### fill()

In this case, Year is a useful variable to include, particularly if we want to explore trends over time. Therefore, we use fill() to fill in those empty cells, by specifying the column to fill and the direction (in this case **up**):

Alternatively, we can rearrange the data so that we would need to fill in a downward direction:

We now have a useful dataset for plotting:

But less useful for presenting in a table, so let’s practice converting this long, untidy dataframe into a wider, tidy dataframe:

N.B. In this case, we had to specify to only include the three variables Facility, Year, and Cases as the additional variable Measurement would interfere with the creation of the table:

## Resources

Here is a helpful [tutorial](https://datacarpentry.org/r-socialsci/03-dplyr-tidyr/index.html)

# Grouping data

This page covers how to group and aggregate data for descriptive analysis. It makes use of the **tidyverse** family of packages for common and easy-to-use functions.

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.

This page will address the following topics:

* Group data with the group\_by() function
* Un-group 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

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). The dataset is imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data.

The first 50 rows of linelist:

## 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().

**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](https://tibble.tidyverse.org/) which, when printed, displays which groupings have been applied and how many groups there are - written just above the header row.

### 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.

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.

### 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().

### 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.

\*\* 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.

## 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:

You can also remove grouping for only specific columns, by placing the column name inside ungroup().

**NOTE:** The verb count() automatically ungroups the data after counting.

## Summarise

See the **dplyr** section of the [Descriptive tables](#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.

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.

**TIP:** The summarise function works with both UK and US spelling - summarise() and summarize() call the same function.

## Counts and tallies

count() and tally() provide similar functionality but are different. Read more about the distinction between tally() and count() [here](https://dplyr.tidyverse.org/reference/tally.html)

### 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.

### 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()

Just like with group\_by() you can create a new column within the count() command:

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 =).

### 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](#group_filter_grp_size) for another example.

### Add totals

To easily add total sum rows or columns after using tally() or count(), see the **janitor** section of the [Descriptive tables](#tbl_janitor) page. This package offers functions like adorn\_totals() and adorn\_percentages() to add totals and convert to show percentages. Below is a brief example:

To add more complex totals rows that involve summary statistics other than sums, see [this section of the Descriptive Tables page](#tbl_dplyr_totals).

## Grouping by date

When grouping data by date, you must have (or create) a column for the date unit of interest - for example “day”, “epiweek”, “month”, etc. You can make this column using floor\_date() from **lubridate**, as explained in the [Epidemiological weeks section](#dates_epi_wks) of the [Working with dates](#working-with-dates-1) page. Once you have this column, you can use count() from **dplyr** to group the rows by those unique date values and achieve aggregate counts.

One additional step common for date situations, is to “fill-in” any dates in the sequence that are not present in the data. Use complete() from **tidyr** so that the aggregated date series is complete including all possible date units within the range. Without this step, a week with no cases reported might not appear in your data!

Within complete() you re-define your date column as a sequence of dates seq.Date() from the minimum to the maximum - thus the dates are expanded. By default, the case count values in any new “expanded” rows will be NA. You can set them to 0 using the fill = argument of complete(), which expects a named list (if your counts column is named n, provide fill = list(n = 0). See ?complete for details and the [Working with dates](#dates_epi_wks) page for an example.

### Linelist cases into days

Here is an example of grouping cases into days without using complete(). Note the first rows skip over dates with no cases.

Below we add the complete() command to ensure every day in the range is represented.

### Linelist cases into weeks

The same principle can be applied for weeks. First create a new column that is the week of the case using floor\_date() with unit = "week". Then, use count() as above to achieve weekly case counts. Finish with complete() to ensure that all weeks are represented, even if they contain no cases.

Here are the first 50 rows of the resulting data frame:

### Linelist cases into months

To aggregate cases into months, again use floor\_date() from the **lubridate** package, but with the argument unit = "months". This rounds each date down to the 1st of its month. The output will be class Date. Note that in the complete() step we also use by = "months".

### Daily counts into weeks

To aggregate daily counts into weekly counts, use floor\_date() as above. However, use group\_by() and summarize() instead of count() because you need to sum() daily case counts instead of just counting the number of rows per week.

#### Daily counts into months

To aggregate daily counts into months counts, use floor\_date() with unit = "month" as above. However, use group\_by() and summarize() instead of count() because you need to sum() daily case counts instead of just counting the number of rows per month.

## 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().

## 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](https://dplyr.tidyverse.org/reference/slice.html) 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

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](#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().

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:

## 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.

## 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.

## Resources

Here are some useful resources for more information:

You can perform any summary function on grouped data; see the [RStudio data transformation cheat sheet](https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf)

The Data Carpentry page on [**dplyr**](https://datacarpentry.org/R-genomics/04-dplyr.html)  
The **tidyverse** reference pages on [group\_by()](https://dplyr.tidyverse.org/reference/group_by.html) and [grouping](https://dplyr.tidyverse.org/articles/grouping.html)

This page on [Data manipulation](https://itsalocke.com/files/DataManipulationinR.pdf)

[Summarize with conditions in dplyr](https://stackoverflow.com/questions/23528862/summarize-with-conditions-in-dplyr)

# Joining data

Above: an animated example of a left join ([*image source*](https://github.com/gadenbuie/tidyexplain/tree/master/images))

This page describes ways to “join”, “match”, “link” “bind”, and otherwise combine data frames.

It is uncommon that your epidemiological analysis or workflow does not involve multiple sources of data, and the linkage of multiple datasets. Perhaps you need to connect laboratory data to patient clinical outcomes, or Google mobility data to infectious disease trends, or even a dataset at one stage of analysis to a transformed version of itself.

In this page we demonstrate code to:

* Conduct joins of two data frames such that rows are matched based on common values in identifier columns
* Join two data frames based on probabilistic (likely) matches between values
* Expand a data frame by directly binding or (“appending”) rows or columns from another data frame

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

To begin, we import the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

### Example datasets

In the joining section below, we will use the following datasets:

1. A “miniature” version of the case linelist, containing only the columns case\_id, date\_onset, and hospital, and only the first 10 rows
2. A separate data frame named hosp\_info, which contains more details about each hospital

In the section on probabilistic matching, we will use two different small datasets. The code to create those datasets is given in that section.

#### “Miniature” case linelist

Below is the the miniature case linelist, which contains only 10 rows and only columns case\_id, date\_onset, and hospital.

#### Hospital information data frame

Below is the code to create a separate data frame with additional information about seven hospitals (the catchment population, and the level of care available). Note that the name “Military Hospital” belongs to two different hospitals - one a primary level serving 10000 residents and the other a secondary level serving 50280 residents.

Here is this data frame:

### Pre-cleaning

Traditional joins (non-probabilistic) are case-sensitive and require exact character matches between values in the two data frames. To demonstrate some of the cleaning steps you might need to do before initiating a join, we will clean and align the linelist\_mini and hosp\_info datasets now.

**Identify differences**

We need the values of the hosp\_name column in the hosp\_info data frame to match the values of the hospital column in the linelist\_mini data frame.

Here are the values in the linelist\_mini data frame, printed with the **base** R function unique():

and here are the values in the hosp\_info data frame:

You can see that while some of the hospitals exist in both data frames, there are many differences in spelling.

**Align values**

We begin by cleaning the values in the hosp\_info data frame. As explained in the [Cleaning data and core functions](#cleaning-data-and-core-functions) page, we can re-code values with logical criteria using **dplyr**’s case\_when() function. For the four hospitals that exist in both data frames we change the values to align with the values in linelist\_mini. The other hospitals we leave the values as they are (TRUE ~ hosp\_name).

**CAUTION:** Typically when cleaning one should create a new column (e.g. hosp\_name\_clean), but for ease of demonstration we show modification of the old column

The hospital names that appear in both data frames are aligned. There are two hospitals in hosp\_info that are not present in linelist\_mini - we will deal with these later, in the join.

Prior to a join, it is often easiest to convert a column to all lowercase or all uppercase. If you need to convert all values in a column to UPPER or lower case, use mutate() and wrap the column with one of these functions from **stringr**, as shown in the page on [Characters and strings](#characters-and-strings).

str\_to\_upper()  
str\_to\_upper()  
str\_to\_title()

## ****dplyr**** joins

The **dplyr** package offers several different join functions. **dplyr** is included in the **tidyverse** package. These join functions are described below, with simple use cases.

Many thanks to [https://github.com/gadenbuie](https://github.com/gadenbuie/tidyexplain/tree/master/images) for the informative gifs!

### General syntax

The join commands can be run as standalone commands to join two data frames into a new object, or they can be used within a pipe chain (%>%) to merge one data frame into another as it is being cleaned or otherwise modified.

In the example below, the function left\_join() is used as a standalone command to create the a new joined\_data data frame. The inputs are data frames 1 and 2 (df1 and df2). The first data frame listed is the baseline data frame, and the second one listed is joined to it.

The third argument by = is where you specify the columns in each data frame that will be used to aligns the rows in the two data frames. If the names of these columns are different, provide them within a c() vector as shown below, where the rows are matched on the basis of common values between the column ID in df1 and the column identifier in df2.

If the by columns in both data frames have the exact same name, you can just provide this one name, within quotes.

If you are joining the data frames based on common values across multiple fields, list these fields within the c() vector. This example joins rows if the values in three columns in each dataset align exactly.

The join commands can also be run within a pipe chain. This will modify the data frame being piped.

In the example below, df1 is is passed through the pipes, df2 is joined to it, and df is thus modified and re-defined.

**CAUTION:** Joins are case-specific! Therefore it is useful to convert all values to lowercase or uppercase prior to joining. See the page on characters/strings.

### Left and right joins

**A left or right join is commonly used to add information to a data frame** - new information is added only to rows that already existed in the baseline data frame. These are common joins in epidemiological work as they are used to add information from one dataset into another.

In using these joins, the written order of the data frames in the command is important\*.

* In a left join, the first data frame written is the baseline
* In a right join, the second data frame written is the baseline

**All rows of the baseline data frame are kept.** Information in the other (secondary) data frame is joined to the baseline data frame only if there is a match via the identifier column(s). In addition:

* Rows in the secondary data frame that do not match are dropped.
* If there are many baseline rows that match to one row in the secondary data frame (many-to-one), the secondary information is added to each matching baseline row.
* If a baseline row matches to multiple rows in the secondary data frame (one-to-many), all combinations are given, meaning new rows may be added to your returned data frame!

Animated examples of left and right joins ([image source](https://github.com/gadenbuie/tidyexplain/tree/master/images))

**Example**

Below is the output of a left\_join() of hosp\_info (secondary data frame, [view here](#joins_hosp_info)) into linelist\_mini (baseline data frame, [view here](#joins_llmini)). The original linelist\_mini has nrow(linelist\_mini) rows. The modified linelist\_mini is displayed. Note the following:

* Two new columns, catchment\_pop and level have been added on the left side of linelist\_mini
* All original rows of the baseline data frame linelist\_mini are kept
* Any original rows of linelist\_mini for “Military Hospital” are duplicated because it matched to two rows in the secondary data frame, so both combinations are returned
* The join identifier column of the secondary dataset (hosp\_name) has disappeared because it is redundant with the identifier column in the primary dataset (hospital)
* When a baseline row did not match to any secondary row (e.g. when hospital is “Other” or “Missing”), NA (blank) fills in the columns from the secondary data frame
* Rows in the secondary data frame with no match to the baseline data frame (“sisters” and “ignace” hospitals) were dropped

#### “Should I use a right join, or a left join?”

To answer the above question, ask yourself “which data frame should retain all of its rows?” - use this one as the baseline. A left join keep all the rows in the first data frame written in the command, whereas a right join keeps all the rows in the second data frame.

The two commands below achieve the same output - 10 rows of hosp\_info joined into a linelist\_mini baseline, but they use different joins. The result is that the column order will differ based on whether hosp\_info arrives from the right (in the left join) or arrives from the left (in the right join). The order of the rows may also shift accordingly. But both of these consequences can be subsequently addressed, using select() to re-order columns or arrange() to sort rows.

Here is the result of hosp\_info into linelist\_mini via a left join (new columns incoming from the right)

Here is the result of hosp\_info into linelist\_mini via a right join (new columns incoming from the left)

Also consider whether your use-case is within a pipe chain (%>%). If the dataset in the pipes is the baseline, you will likely use a left join to add data to it.

### Full join

**A full join is the most inclusive of the joins** - it returns all rows from both data frames.

If there are any rows present in one and not the other (where no match was found), the data frame will include them and become longer. NA missing values are used to fill-in any gaps created. As you join, watch the number of columns and rows carefully to troubleshoot case-sensitivity and exact character matches.

The “baseline” data frame is the one written first in the command. Adjustment of this will not impact which records are returned by the join, but it can impact the resulting column order, row order, and which identifier columns are retained.

Animated example of a full join ([image source](https://github.com/gadenbuie/tidyexplain/tree/master/images))

**Example**

Below is the output of a full\_join() of hosp\_info (originally nrow(hosp\_info), [view here](#joins_hosp_info)) into linelist\_mini (originally nrow(linelist\_mini), [view here](#joins_llmini)). Note the following:

* All baseline rows are kept (linelist\_mini)
* Rows in the secondary that do not match to the baseline are kept (“ignace” and “sisters”), with values in the corresponding baseline columns case\_id and onset filled in with missing values
* Likewise, rows in the baseline data frame that do not match to the secondary (“Other” and “Missing”) are kept, with secondary columns catchment\_pop and level filled-in with missing values
* In the case of one-to-many or many-to-one matches (e.g. rows for “Military Hospital”), all possible combinations are returned (lengthening the final data frame)
* Only the identifier column from the baseline is kept (hospital)

### Inner join

**An inner join is the most restrictive of the joins** - it returns only rows with matches across both data frames.  
This means that the number of rows in the baseline data frame may actually reduce. Adjustment of which data frame is the “baseline” (written first in the function) will not impact which rows are returned, but it will impact the column order, row order, and which identifier columns are retained.

Animated example of an inner join ([image source](https://github.com/gadenbuie/tidyexplain/tree/master/images))

**Example**

Below is the output of an inner\_join() of linelist\_mini (baseline) with hosp\_info (secondary). Note the following:

* Baseline rows with no match to the secondary data are removed (rows where hospital is “Missing” or “Other”)
* Likewise, rows from the secondary data frame that had no match in the baseline are removed (rows where hosp\_name is “sisters” or “ignace”)
* Only the identifier column from the baseline is kept (hospital)

### Semi join

A semi join is a “filtering join” which uses another dataset not to add rows or columns, but to perform filtering.

A **semi-join keeps all observations in the baseline data frame that have a match in the secondary data frame** (but does not add new columns nor duplicate any rows for multiple matches). Read more about these “filtering” joins [here](https://towardsdatascience.com/level-up-with-semi-joins-in-r-a068426096e0).

Animated example of a semi join ([image source](https://github.com/gadenbuie/tidyexplain/tree/master/images))

As an example, the below code returns rows from the hosp\_info data frame that have matches in linelist\_mini based on hospital name.

### Anti join

**The anti join is another “filtering join” that returns rows in the baseline data frame that do not have a match in the secondary data frame.**

Read more about filtering joins [here](https://towardsdatascience.com/level-up-with-semi-joins-in-r-a068426096e0).

Common scenarios for an anti-join include identifying records not present in another data frame, troubleshooting spelling in a join (reviewing records that should have matched), and examining records that were excluded after another join.

**As with right\_join() and left\_join(), the baseline data frame (listed first) is important**. The returned rows are from the baseline data frame only. Notice in the gif below that row in the secondary data frame (purple row 4) is not returned even though it does not match with the baseline.

Animated example of an anti join ([image source](https://github.com/gadenbuie/tidyexplain/tree/master/images))

#### Simple anti\_join() example

For a simple example, let’s find the hosp\_info hospitals that do not have any cases present in linelist\_mini. We list hosp\_info first, as the baseline data frame. The hospitals which are not present in linelist\_mini are returned.

#### Complex anti\_join() example

For another example, let us say we ran an inner\_join() between linelist\_mini and hosp\_info. This returns only a subset of the original linelist\_mini records, as some are not present in hosp\_info.

To review the linelist\_mini records that were excluded during the inner join, we can run an anti-join with the same settings (linelist\_mini as the baseline).

To see the hosp\_info records that were excluded in the inner join, we could also run an anti-join with hosp\_info as the baseline data frame.

## Probabalistic matching

If you do not have a unique identifier common across datasets to join on, consider using a probabilistic matching algorithm. This would find matches between records based on similarity (e.g. Jaro–Winkler string distance, or numeric distance). Below is a simple example using the package **fastLink** .

**Load packages**

Here are two small example datasets that we will use to demonstrate the probabilistic matching (cases and test\_results):

Here is the code used to make the datasets:

**The cases dataset has 9 records** of patients who are awaiting test results.

**The test\_results dataset** has 14 records and contains the column result, which we want to add to the records in cases based on probabilistic matching of records.

### Probabilistic matching

The fastLink() function from the **fastLink** package can be used to apply a matching algorithm. Here is the basic information. You can read more detail by entering ?fastLink in your console.

* Define the two data frames for comparison to arguments dfA = and dfB =
* In varnames = give all column names to be used for matching. They must all exist in both dfA and dfB.
* In stringdist.match = give columns from those in varnames to be evaluated on string “distance”.
* In numeric.match = give columns from those in varnames to be evaluated on numeric distance.
* Missing values are ignored
* By default, each row in either data frame is matched to at most one row in the other data frame. If you want to see all the evaluated matches, set dedupe.matches = FALSE. The deduplication is done using Winkler’s linear assignment solution.

Tip: split one date column into three separate numeric columns using *day()*, *month()*, and *year()* from ***lubridate*** package

The default threshold for matches is 0.94 (threshold.match =) but you can adjust it higher or lower. If you define the threshold, consider that higher thresholds could yield more false-negatives (rows that do not match which actually should match) and likewise a lower threshold could yield more false-positive matches.

Below, the data are matched on string distance across the name and district columns, and on numeric distance for year, month, and day of birth. A match threshold of 95% probability is set.

**Review matches**

We defined the object returned from fastLink() as fl\_output. It is of class list, and it actually contains several data frames within it, detailing the results of the matching. One of these data frames is matches, which contains the most likely matches across cases and results. You can access this “matches” data frame with fl\_output$matches. Below, it is saved as my\_matches for ease of accessing later.

When my\_matches is printed, you see two column vectors: the pairs of row numbers/indices (also called “rownames”) in cases (“inds.a”) and in results (“inds.b”) representing the best matches. If a row number from a datafrane is missing, then no match was found in the other data frame at the specified match threshold.

Things to note:

* Matches occurred despite slight differences in name spelling and dates of birth:
  + “Tony B. Smith” matched to “Anthony B Smith”
  + “Maria Rodriguez” matched to “Marialisa Rodrigues”
  + “Betty Chase” matched to “Elizabeth Chase”
  + “Olivier Laurent De Bordeaux” matched to “Oliver Laurent De Bordow” (missing date of birth ignored)
* One row from cases (for “Blessing Adebayo”, row 9) had no good match in results, so it is not present in my\_matches.

**Join based on the probabilistic matches**

To use these matches to join results to cases, one strategy is:

1. Use left\_join() to join my\_matches to cases (matching rownames in cases to “inds.a” in my\_matches)
2. Then use another left\_join() to join results to cases (matching the newly-acquired “inds.b” in cases to rownames in results)

Before the joins, we should clean the three data frames:

* Both dfA and dfB should have their row numbers (“rowname”) converted to a proper column.
* Both the columns in my\_matches are converted to class character, so they can be joined to the character rownames

As performed using the code above, the resulting data frame complete will contain all columns from both cases and results. Many will be appended with suffixes “.x” and “.y”, because the column names would otherwise be duplicated.

Alternatively, to achieve only the “original” 9 records in cases with the new column(s) from results, use select() on results before the joins, so that it contains only rownames and the columns that you want to add to cases (e.g. the column result).

If you want to subset either dataset to only the rows that matched, you can use the codes below:

Or, to see only the rows that did **not** match:

### Probabilistic deduplication

Probabilistic matching can be used to deduplicate a dataset as well. See the page on deduplication for other methods of deduplication.

Here we began with the cases dataset, but are now calling it cases\_dup, as it has 2 additional rows that could be duplicates of previous rows: See “Tony” with “Anthony”, and “Marialisa Rodrigues” with “Maria Rodriguez”.

Run fastLink() like before, but compare the cases\_dup data frame to itself. When the two data frames provided are identical, the function assumes you want to de-duplicate. Note we do not specify stringdist.match = or numeric.match = as we did previously.

Now, you can review the potential duplicates with getMatches(). Provide the data frame as both dfA = and dfB =, and provide the output of the fastLink() function as fl.out =. fl.out must be of class fastLink.dedupe, or in other words, the result of fastLink().

See the right-most column, which indicates the duplicate IDs - the final two rows are identified as being likely duplicates of rows 2 and 3.

To return the row numbers of rows which are likely duplicates, you can count the number of rows per unique value in the dedupe.ids column, and then filter to keep only those with more than one row. In this case this leaves rows 2 and 3.

To inspect the whole rows of the likely duplicates, put the row number in this command:

## Binding and aligning

Another method of combining two data frames is “binding” them together. You can also think of this as “appending” or “adding” rows or columns.

This section will also discuss how to “align” the order of rows of one data frame to the order in another data frame. This topic is discussed below in the section on Binding columns.

### 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](#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](#descriptive-tables) and [Tables for presentation](#tables-for-presentation) pages.

Here is the hosp\_summary data frame:

Create a data frame with the “total” statistics (not grouped by hospital). This will return just one row.

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).

Now we can bind the rows together with bind\_rows().

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. As explained in the [Tables for presentation](#tables-for-presentation) page, you could “fill-in” this cell with “Total” using replace\_na().

### Bind columns

There is a similar **dplyr** function bind\_cols() which you can use to combine two data frames sideways. Note that rows are matched to each other by position (not like a join above) - for example the 12th row in each data frame will be aligned.

For an example, we bind several summary tables together. In order to do this, we also demonstrate how to re-arrange the order of rows in one data frame to match the order in another data frame, with match().

Here we define case\_info as a summary data frame of linelist cases, by hospital, with the number of cases and the number of deaths.

And let’s say that here is a different data frame contact\_fu containing information on the percent of exposed contacts investigated and “followed-up”, again by hospital.

Note that the hospitals are the same, but are in different orders in each data frame. The easiest solution would be to use a left\_join() on the hospital column, but you could also use bind\_cols() with one extra step.

#### Use match() to align ordering

Because the row orders are different, a simple bind\_cols() command would result in a mis-match of data. To fix this we can use match() from **base** R to align the rows of a data frame in the same order as in another. We assume for this approach that there are no duplicate values in either data frame.

When we use match(), the syntax is match(TARGET ORDER VECTOR, DATA FRAME COLUMN TO CHANGE), where the first argument is the desired order (either a stand-alone vector, or in this case a column in a data frame), and the second argument is the data frame column in the data frame that will be re-ordered. The output of match() is a vector of numbers representing the correct position ordering. You can read more with ?match.

You can use this numeric vector to re-order the data frame - place it within subset brackets [ ] before the comma. Read more about **base** R bracket subset syntax in the [R basics](#r-basics) page. The command below creates a new data frame, defined as the old one in which the rows are ordered in the numeric vector above.

Now we can bind the data frame columns together, with the correct row order. Note that some columns are duplicated and will require cleaning with rename(). Read more aboout bind\_rows() [here](https://dplyr.tidyverse.org/reference/bind.html).

A **base** R alternative to bind\_cols is cbind(), which performs the same operation.

## Resources

The [tidyverse page on joins](https://dplyr.tidyverse.org/reference/join.html)

The [R for Data Science page on relational data](https://r4ds.had.co.nz/relational-data.html)

Th [tidyverse page on dplyr](https://dplyr.tidyverse.org/reference/bind.html) on binding

A vignette on [fastLink](https://github.com/kosukeimai/fastLink) at the package’s Github page

Publication describing methodology of [fastLink](https://imai.fas.harvard.edu/research/files/linkage.pdf)

Publication describing [RecordLinkage package](https://journal.r-project.org/archive/2010/RJ-2010-017/RJ-2010-017.pdf)

# De-duplication

This page covers the following de-duplication techniques:

1. Identifying and removing duplicate rows
2. “Slicing” rows to keep only certain rows (e.g. min or max) from each group of rows
3. “Rolling-up”, or combining values from multiple rows into one row

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

For demonstration, we will use an example dataset that is created with the R code below.

The data are records of COVID-19 phone encounters, including encounters with contacts and with cases. The columns include recordID (computer-generated), personID, name, date of encounter, time of encounter, the purpose of the encounter (either to interview as a case or as a contact), and symptoms\_ever (whether the person in that encounter reported ever having symptoms).

Here is the code to create the obs dataset:

#### Here is the data frame

Use the filter boxes along the top to review the encounters for each person.

A few things to note as you review the data:

* The first two records are 100% complete duplicates including duplicate recordID (must be a computer glitch!)
* The second two rows are duplicates, in all columns except for *recordID*
* Several people had multiple phone encounters, at various dates and times, and as contacts and/or cases
* At each encounter, the person was asked if they had **ever** had symptoms, and some of this information is missing.

And here is a quick summary of the people and the purposes of their encounters, using tabyl() from **janitor**:

## Deduplication

This section describes how to review and remove duplicate rows in a data frame. It also show how to handle duplicate elements in a vector.

### Examine duplicate rows

To quickly review rows that have duplicates, you can use get\_dupes() from the **janitor** package. By default, all columns are considered when duplicates are evaluated - rows returned by the function are 100% duplicates considering the values in all columns.

In the obs data frame, the first two rows are 100% duplicates - they have the same value in every column (including the recordID column, which is supposed to be unique - it must be some computer glitch). The returned data frame automatically includes a new column dupe\_count on the right side, showing the number of rows with that combination of duplicate values.

See the [original data](#dedup_data)

However, if we choose to ignore recordID, the 3rd and 4th rows rows are also duplicates of each other. That is, they have the same values in all columns except for recordID. You can specify specific columns to be ignored in the function using a - minus symbol.

You can also positively specify the columns to consider. Below, only rows that have the same values in the name and purpose columns are returned. Notice how “amrish” now has dupe\_count equal to 3 to reflect his three “contact” encounters.

\*Scroll left for more rows\*\*

See the [original data](#dedup_data).

See ?get\_dupes for more details, or see this [online reference](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html#explore-records-with-duplicated-values-for-specific-combinations-of-variables-with-get_dupes)

### Keep only unique rows

To keep only unique rows of a data frame, use distinct() from **dplyr** (as demonstrated in the [Cleaning data and core functions](#cleaning-data-and-core-functions) page). Rows that are duplicates are removed such that only the first of such rows is kept. By default, “first” means the highest rownumber (order of rows top-to-bottom). Only unique rows remain.

In the example below, we run distinct() such that the column recordID is excluded from consideration - thus **two duplicate rows are removed**. The first row (for “adam”) was 100% duplicated and has been removed. Also row 3 (for “amrish”) was a duplicate in every column except recordID (which is not being considered) and so is also removed. The obs dataset n is now nrow(obs)-2, not nrow(obs) rows).

Scroll to the left to see the entire data frame

**CAUTION:** If using distinct() on grouped data, the function will apply to each group.

**Deduplicate based on specific columns**

You can also specify columns to be the basis for de-duplication. In this way, the de-duplication only applies to rows that are duplicates within the specified columns. Unless you set .keep\_all = TRUE, all columns not mentioned will be dropped.

In the example below, the de-duplication only applies to rows that have identical values for name and purpose columns. Thus, “brian” has only 2 rows instead of 3 - his first “contact” encounter and his only “case” encounter. To adjust so that brian’s latest encounter of each purpose is kept, see the tab on Slicing within groups.

Scroll to the left to see the entire data frame

See the [original data](#dedup_data).

### Deduplicate elements in a vector

The function duplicated() from **base** R will evaluate a vector (column) and return a logical vector of the same length (TRUE/FALSE). The first time a value appears, it will return FALSE (not a duplicate), and subsequent times that value appears it will return TRUE. Note how NA is treated the same as any other value.

To return only the duplicated elements, you can use brackets to subset the original vector:

To return only the unique elements, use unique() from **base** R. To remove NAs from the output, nest na.omit() within unique().

### Using ****base**** R

**To return duplicate rows**

In **base** R, you can also see which rows are 100% duplicates in a data frame df with the command duplicated(df) (returns a logical vector of the rows).

Thus, you can also use the base subset [ ] on the data frame to see the duplicated rows with df[duplicated(df),] (don’t forget the comma, meaning that you want to see all columns!).

**To return unique rows**

See the notes above. To see the unique rows you add the logical negator ! in front of the duplicated() function:  
df[!duplicated(df),]

**To return rows that are duplicates of only certain columns**

Subset the df that is within the *duplicated()* parentheses, so this function will operate on only certain columns of the df.

To specify the columns, provide column numbers or names after a comma (remember, all this is within the duplicated() function).

Be sure to keep the comma , outside after the duplicated() function as well!

For example, to evaluate only columns 2 through 5 for duplicates: df[!duplicated(df[, 2:5]),]  
To evaluate only columns name and purpose for duplicates: df[!duplicated(df[, c("name", "purpose)]),]

## Slicing

To “slice” a data frame to apply a filter on the rows by row number/position. This becomes particularly useful if you have multiple rows per functional group (e.g. per “person”) and you only want to keep one or some of them.

The basic slice() function accepts numbers and returns rows in those positions. If the numbers provided are positive, only they are returned. If negative, those rows are not returned. Numbers must be either all positive or all negative.

See the [original data](#dedup_data).

There are several variations: These should be provided with a column and a number of rows to return (to n =).

* slice\_min() and slice\_max() keep only the row(s) with the minimium or maximum value(s) of the specified column. This also works to return the “min” and “max” of ordered factors.
* slice\_head() and slice\_tail() - keep only the first or last row(s).
* slice\_sample() - keep only a random sample of the rows.

Use arguments n = or prop = to specify the number or proportion of rows to keep. If not using the function in a pipe chain, provide the data argument first (e.g. slice(data, n = 2)). See ?slice for more information.

Other arguments:

.order\_by = used in slice\_min() and slice\_max() this is a column to order by before slicing.  
with\_ties = TRUE by default, meaning ties are kept.  
.preserve = FALSE by default. If TRUE then the grouping structure is re-calculated after slicing.  
weight\_by = Optional, numeric column to weight by (bigger number more likely to get sampled). Also replace = for whether sampling is done with/without replacement.

**TIP:** When using slice\_max() and slice\_min(), be sure to specify/write the n = (e.g. n = 2, not just 2). Otherwise you may get an error Error:…is not empty.

**NOTE:** You may encounter the function [top\_n()](https://dplyr.tidyverse.org/reference/top_n.html), which has been superseded by the slice functions.

### Slice with groups

The slice\_\*() functions can be very useful if applied to a grouped data frame because the slice operation is performed on each group separately. Use the **function** group\_by() in conjunction with slice() to group the data to take a slice from each group.

This is helpful for de-duplication if you have multiple rows per person but only want to keep one of them. You first use group\_by() with key columns that are the same per person, and then use a slice function on a column that will differ among the grouped rows.

In the example below, to keep only the latest encounter per person, we group the rows by name and then use slice\_max() with n = 1 on the date column. Be aware! To apply a function like slice\_max() on dates, the date column must be class Date.

By default, “ties” (e.g. same date in this scenario) are kept, and we would still get multiple rows for some people (e.g. adam). To avoid this we set with\_ties = FALSE. We get back only one row per person.

**CAUTION:** If using arrange(), specify .by\_group = TRUE to have the data arranged within each group.

**DANGER:** If with\_ties = FALSE, the first row of a tie is kept. This may be deceptive. See how for Mariah, she has two encounters on her latest date (6 Jan) and the first (earliest) one was kept. Likely, we want to keep her later encounter on that day. See how to “break” these ties in the next example.

Above, for example we can see that only Amrish’s row on 5 Jan was kept, and only Brian’s row on 7 Jan was kept. See the [original data](#dedup_data).

**Breaking “ties”**

Multiple slice statements can be run to “break ties”. In this case, if a person has multiple encounters on their latest date, the encounter with the latest time is kept (lubridate::hm() is used to convert the character times to a sortable time class).  
Note how now, the one row kept for “Mariah” on 6 Jan is encounter 3 from 08:32, not encounter 2 at 07:25.

In the example above, it would also have been possible to slice by *encounter* number, but we showed the slice on *date* and *time* for example purposes.

**TIP:** To use slice\_max() or slice\_min() on a “character” column, mutate it to an ordered factor class!

See the [original data](#dedup_data).

### Keep all but mark them

If you want to keep all records but mark only some for analysis, consider a two-step approach utilizing a unique recordID/encounter number:

1. Reduce/slice the orginal data frame to only the rows for analysis. Save/retain this reduced data frame.
2. In the original data frame, mark rows as appropriate with case\_when(), based on whether their record unique identifier (recordID in this example) is present in the reduced data frame.

See the [original data](#dedup_data).

### Calculate row completeness

Create a column that contains a metric for the row’s completeness (non-missingness). This could be helpful when deciding which rows to prioritize over others when de-duplicating/slicing.

In this example, “key” columns over which you want to measure completeness are saved in a vector of column names.

Then the new column key\_completeness is created with mutate(). The new value in each row is defined as a calculated fraction: the number of non-missing values in that row among the key columns, divided by the number of key columns.

This involves the function rowSums() from **base** R. Also used is ., which within piping refers to the data frame at that point in the pipe (in this case, it is being subset with brackets []).

\*Scroll to the right to see more rows\*\*

See the [original data](#dedup_data).

## Roll-up values

This section describes:

1. How to “roll-up” values from multiple rows into just one row, with some variations
2. Once you have “rolled-up” values, how to overwrite/prioritize the values in each cell

This tab uses the example dataset from the Preparation tab.

### Roll-up values into one row

The code example below uses group\_by() and summarise() to group rows by person, and then paste together all unique values within the grouped rows. Thus, you get one summary row per person. A few notes:

* A suffix is appended to all new columns ("\_roll" in this example)
* If you want to show only unique values per cell, then wrap the na.omit() with unique()
* na.omit() removes NA values, but if this is not desired it can be removed paste0(.x)…

The result is one row per group (ID), with entries arranged by date and pasted together. Scroll to the left to see more rows

See the [original data](#dedup_data).

**This variation shows unique values only:**

**This variation appends a suffix to each column.**  
In this case "\_roll" to signify that it has been rolled:

### Overwrite values/hierarchy

If you then want to evaluate all of the rolled values, and keep only a specific value (e.g. “best” or “maximum” value), you can use mutate() across the desired columns, to implement case\_when(), which uses str\_detect() from the **stringr** package to sequentially look for string patterns and overwrite the cell content.

Now you can see in the column symptoms\_ever that if the person EVER said “Yes” to symptoms, then only “Yes” is displayed.

See the [original data](#dedup_data).

## Probabilistic de-duplication

Sometimes, you may want to identify “likely” duplicates based on similarity (e.g. string “distance”) across several columns such as name, age, sex, date of birth, etc. You can apply a probabilistic matching algorithm to identify likely duplicates.

See the page on [Joining data](#joining-data) for an explanation on this method. The section on Probabilistic Matching contains an example of applying these algorithms to compare a data frame to itself, thus performing probabilistic de-duplication.

## Resources

Much of the information in this page is adapted from these resources and vignettes online:

[datanovia](https://www.datanovia.com/en/lessons/identify-and-remove-duplicate-data-in-r/)

[dplyr tidyverse reference](https://dplyr.tidyverse.org/reference/slice.html)

[cran janitor vignette](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html#explore-records-with-duplicated-values-for-specific-combinations-of-variables-with-get_dupes)

# Iteration, loops, and lists

Epidemiologists often are faced with repeating analyses on subgroups such as countries, districts, or age groups. These are but a few of the many situations involving iteration. Coding your iterative operations using the approaches below will help you perform such repetitive tasks faster, reduce the chance of error, and reduce code length.

This page will introduce two approaches to iterative operations - using for loops and using the package **purrr**.

1. for loops iterate code across a series of inputs, but are less common in R than in other programming languages. Nevertheless, we introduce them here as a learning tool and reference
2. The **purrr** package is the **tidyverse** approach to iterative operations - it works by “mapping” a function across many inputs (values, columns, datasets, etc.)

Along the way, we’ll show examples like:

* Importing and exporting multiple files
* Creating epicurves for multiple jurisdictions
* Running T-tests for several columns in a data frame

In the **purrr** [section](#iter_purrr) we will also provide several examples of creating and handling lists.

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## for loops

### for loops in R

for loops are not emphasized in R, but are common in other programming languages. As a beginner, they can be helpful to learn and practice with because they are easier to “explore”, “de-bug”, and otherwise grasp exactly what is happening for each iteration, especially when you are not yet comfortable writing your own functions.

You may move quickly through for loops to iterating with mapped functions with **purrr** (see [section below](#iter_purrr)).

### Core components

A for loop has three core parts:

1. The **sequence** of items to iterate through
2. The **operations** to conduct per item in the sequence
3. The **container** for the results (optional)

The basic syntax is: for (item in sequence) {do operations using item}. Note the parentheses and the curly brackets. The results could be printed to console, or stored in a container R object.

A simple for loop example is below.

### Sequence

This is the “for” part of a for loop - the operations will run “for” each item in the sequence. The sequence can be a series of values (e.g. names of jurisdictions, diseases, column names, list elements, etc), or it can be a series of consecutive numbers (e.g. 1,2,3,4,5). Each approach has their own utilities, described below.

The basic structure of a sequence statement is item in vector.

* You can write any character or word in place of “item” (e.g. “i”, “num”, “hosp”, “district”, etc.). The value of this “item” changes with each iteration of the loop, proceeding through each value in the vector.
* The vector could be of character values, column names, or perhaps a sequence of numbers - these are the values that will change with each iteration. You can use them within the for loop operations using the “item” term.

**Example: sequence of character values**

In this example, a loop is performed for each value in a pre-defined character vector of hospital names.

We have chosen the term hosp to represent values from the vector hospital\_names. For the first iteration of the loop, the value of hosp will be hospital\_names[[1]]. For the second loop it will be hospital\_names[[2]]. And so on…

**Example: sequence of column names**

This is a variation on the character sequence above, in which the names of an existing R object are extracted and become the vector. For example, the column names of a data frame. Conveniently, in the operations code of the for loop, the column names can be used to index (subset) their original data frame

Below, the sequence is the names() (column names) of the linelist data frame. Our “item” name is col, which will represent each column name as the loops proceeds.

For purposes of example, we include operations code inside the for loop, which is run for every value in the sequence. In this code, the sequence values (column names) are used to index (subset) linelist, one-at-a-time. As taught in the [R basics](#r-basics) page, double branckets [[ ]] are used to subset. The resulting column is passed to is.na(), then to sum() to produce the number of values in the column that are missing. The result is printed to the console - one number for each column.

A note on indexing with column names - whenever referencing the column itself do not just write “col”! col represents just the character column name! To refer to the entire column you must use the column name as an index on linelist via linelist[[col]].

**Sequence of numbers**

In this approach, the sequence is a series of consecutive numbers. Thus, the value of the “item” is not a character value (e.g. “Central Hospital” or “date\_onset”) but is a number. This is useful for looping through data frames, as you can use the “item” number inside the for loop to index the data frame by row number.

For example, let’s say that you want to loop through every row in your data frame and extract certain information. Your “items” would be numeric row numbers. Often, “items” in this case are written as i.

The for loop process could be explained in words as “for every item in a sequence of numbers from 1 to the total number of rows in my data frame, do X”. For the first iteration of the loop, the value of “item” i would be 1. For the second iteration, i would be 2, etc.

Here is what the sequence looks like in code: for (i in 1:nrow(linelist)) {OPERATIONS CODE} where i represents the “item” and 1:nrow(linelist) produces a sequence of consecutive numbers from 1 through the number of rows in linelist.

If you want the sequence to be numbers, but you are starting from a vector (not a data frame), use the shortcut seq\_along() to return a sequence of numbers for each element in the vector. For example, for (i in seq\_along(hospital\_names) {OPERATIONS CODE}.

The below code actually returns numbers, which would become the value of i in their respective loop.

One advantage of using numbers in the sequence is that is easy to also use the i number to index a container that stores the loop outputs. There is an example of this in the Operations section below.

### Operations

This is code within the curly brackets { } of the for loop. You want this code to run for each “item” in the sequence. Therefore, be careful that every part of your code that changes by the “item” is correctly coded such that it actually changes! E.g. remember to use [[ ]] for indexing.

In the example below, we iterate through each row in the linelist. The gender and age values of each row are pasted together and stored in the container character vector cases\_demographics. Note how we also use indexing [[i]] to save the loop output to the correct position in the “container” vector.

### Container

Sometimes the results of your for loop will be printed to the console or RStudio Plots pane. Other times, you will want to store the outputs in a “container” for later use. Such a container could be a vector, a data frame, or even a list.

It is most efficient to create the container for the results before even beginning the for loop. In practice, this means creating an empty vector, data frame, or list. These can be created with the functions vector() for vectors or lists, or with matrix() and data.frame() for a data frame.

**Empty vector**

Use vector() and specify the mode = based on the expected class of the objects you will insert - either “double” (to hold numbers), “character”, or “logical”. You should also set the length = in advance. This should be the length of your for loop sequence.

Say you want to store the median delay-to-admission for each hospital. You would use “double” and set the length to be the number of expected outputs (the number of unique hospitals in the data set).

**Empty data frame**

You can make an empty data frame by specifying the number of rows and columns like this:

**Empty list**

You may want store some plots created by a for loop in a list. A list is like vector, but holds other R objects within it that can be of different classes. Items in a list could be a single number, a dataframe, a vector, and even another list.

You actually initialize an empty list using the same vector() command as above, but with mode = "list". Specify the length however you wish.

### Printing

Note that to print from within a for loop you will likely need to explicitly wrap with the function print().

In this example below, the sequence is an explicit character vector, which is used to subset the linelist by hospital. The results are not stored in a container, but rather are printed to console with the print() function.

### Testing your for loop

To test your loop, you can run a command to make a temporary assignment of the “item”, such as i <- 10 or hosp <- "Central Hospital". Do this outside the loop and then run your operations code only (the code within the curly brackets) to see if the expected results are produced.

### Looping plots

To put all three components together (container, sequence, and operations) let’s try to plot an epicurve for each hospital (see page on [Epidemic curves](#epidemic-curves)).

We can make a nice epicurve of all the cases by gender using the **incidence2** package as below:

To produce a separate plot for each hospital’s cases, we can put this epicurve code within a for loop.

First, we save a named vector of the unique hospital names, hospital\_names. The for loop will run once for each of these names: for (hosp in hospital\_names). Each iteration of the for loop, the current hospital name from the vector will be represented as hosp for use within the loop.

Within the loop operations, you can write R code as normal, but use the “item” (hosp in this case) knowing that its value will be changing. Within this loop:

* A filter() is applied to linelist, such that column hospital must equal the current value of hosp
* The incidence object is created on the filtered linelist
* The plot for the current hospital is created, with an auto-adjusting title that uses hosp
* The plot for the current hospital is temporarily saved and then printed
* The loop then moves onward to repeat with the next hospital in hospital\_names

### Tracking progress of a loop

A loop with many iterations can run for many minutes or even hours. Thus, it can be helpful to print the progress to the R console. The if statement below can be placed within the loop operations to print every 100th number. Just adjust it so that i is the “item” in your loop.

## ****purrr**** and lists

Another approach to iterative operations is the **purrr** package - it is the **tidyverse** approach to iteration.

If you are faced with performing the same task several times, it is probably worth creating a generalised solution that you can use across many inputs. For example, producing plots for multiple jurisdictions, or importing and combining many files.

There are also a few other advantages to **purrr** - you can use it with pipes %>%, it handles errors better than normal for loops, and the syntax is quite clean and simple! If you are using a for loop, you can probably do it more clearly and succinctly with **purrr**!

Keep in mind that **purrr** is a functional programming tool. That is, the operations that are to be iteratively applied are wrapped up into functions. See the [Writing functions](#writing-functions-1) page to learn how to write your own functions.

**purrr** is also almost entirely based around lists and vectors - so think about it as applying a function to each element of that list/vector!

### Load packages

**purrr** is part of the **tidyverse**, so there is no need to install/load a separate package.

### map()

One core **purrr** function is map(), which “maps” (applies) a function to each input element of a list/vector you provide.

The basic syntax is map(.x = SEQUENCE, .f = FUNCTION, OTHER ARGUMENTS). In a bit more detail:

* .x = are the inputs upon which the .f function will be iteratively applied - e.g. a vector of jurisdiction names, columns in a data frame, or a list of data frames
* .f = is the function to apply to each element of the .x input - it could be a function like print() that already exists, or a custom function that you define. The function is often written after a tilde ~ (details below).

A few more notes on syntax:

* If the function needs no further arguments specified, it can be written with no parentheses and no tilde (e.g. .f = mean). To provide arguments that will be the same value for each iteration, provide them within map() but outside the .f = argument, such as the na.rm = T in map(.x = my\_list, .f = mean, na.rm=T).
* You can use .x (or simply .) within the .f = function as a placeholder for the .x value of that iteration
* Use tilde syntax (~) to have greater control over the function - write the function as normal with parentheses, such as: map(.x = my\_list, .f = ~mean(., na.rm = T)). Use this syntax particularly if the value of an argument will change each iteration, or if it is the value .x itself (see examples below)

**The output of using map() is a list** - a list is an object class like a vector but whose elements can be of different classes. So, a list produced by map() could contain many data frames, or many vectors, many single values, or even many lists! There are alternative versions of map() explained below that produce other types of outputs (e.g. map\_dfr() to produce a data frame, map\_chr() to produce character vectors, and map\_dbl() to produce numeric vectors).

#### Example - import and combine Excel sheets

**Let’s demonstrate with a common epidemiologist task:** - You want to import an Excel workbook with case data, but the data are split across different named sheets in the workbook. How do you efficiently import and combine the sheets into one data frame?

Let’s say we are sent the below Excel workbook. Each sheet contains cases from a given hospital.

Here is one approach that uses map():

1. map() the function import() so that it runs for each Excel sheet
2. Combine the imported data frames into one using bind\_rows()
3. Along the way, preserve the original sheet name for each row, storing this information in a new column in the final data frame

First, we need to extract the sheet names and save them. We provide the Excel workbook’s file path to the function excel\_sheets() from the package **readxl**, which extracts the sheet names. We store them in a character vector called sheet\_names.

Here are the names:

Now that we have this vector of names, map() can provide them one-by-one to the function import(). In this example, the sheet\_names are .x and import() is the function .f.

Recall from the [Import and export](#import-and-export) page that when used on Excel workbooks, import() can accept the argument which = specifying the sheet to import. Within the .f function import(), we provide which = .x, whose value will change with each iteration through the vector sheet\_names - first “Central Hospital”, then “Military Hospital”, etc.

Of note - because we have used map(), the data in each Excel sheet will be saved as a separate data frame within a list. We want each of these list elements (data frames) to have a name, so before we pass sheet\_names to map() we pass it through set\_names() from **purrr**, which ensures that each list element gets the appropriate name.

We save the output list as combined.

When we inspect output, we see that the data from each Excel sheet is saved in the list with a name. This is good, but we are not quite finished.

Lastly, we use the function bind\_rows() (from **dplyr**) which accepts the list of similarly-structured data frames and combines them into one data frame. To create a new column from the list element names, we use the argument .id = and provide it with the desired name for the new column.

Below is the whole sequence of commands:

And now we have one data frame with a column containing the sheet of origin!

There are variations of map() that you should be aware of. For example, map\_dfr() returns a data frame, not a list. Thus, we could have used it for the task above and not have had to bind rows. But then we would not have been able to capture which sheet (hospital) each case came from.

Other variations include map\_chr(), map\_dbl(). These are very useful functions for two reasons. Firstly. they automatically convert the output of an iterative function into a vector (not a list). Secondly, they can explicitly control the class that the data comes back in - you ensure that your data comes back as a character vector with map\_chr(), or numeric vector with map\_dbl(). Lets return to these later in the section!

The functions map\_at() and map\_if() are also very useful for iteration - they allow you to specify which elements of a list you should iterate at! These work by simply applying a vector of indexes/names (in the case of map\_at()) or a logical test (in the case of map\_if()).

Lets use an example where we didn’t want to read the first sheet of hospital data. We use map\_at() instead of map(), and specify the .at = argument to c(-1) which means to not use the first element of .x. Alternatively, you can provide a vector of positive numbers, or names, to .at = to specify which elements to use.

Note that the first sheet name will still appear as an element of the output list - but it is only a single character name (not a data frame). You would need to remove this element before binding rows. We will cover how to remove and modify list elements in a later section.

### Split dataset and export

Below, we give an example of how to split a dataset into parts and then use map() iteration to export each part as a separate Excel sheet, or as a separate CSV file.

#### Split dataset

Let’s say we have the complete case linelist as a data frame, and we now want to create a separate linelist for each hospital and export each as a separate CSV file. Below, we do the following steps:

Use group\_split() (from **dplyr**) to split the linelist data frame by unique values in column hospital. The output is a list containing one data frame per hospital subset.

We can run View(linelist\_split) and see that this list contains 6 data frames (“tibbles”), each representing the cases from one hospital.

However, note that the data frames in the list do not have names by default! We want each to have a name, and then to use that name when saving the CSV file.

One approach to extracting the names is to use pull() (from **dplyr**) to extract the hospital column from each data frame in the list. Then, to be safe, we convert the values to character and then use unique() to get the name for that particular data frame. All of these steps are applied to each data frame via map().

We can now see that each of the list elements has a name. These names can be accessed via names(linelist\_split).

##### More than one group\_split() column

If you wanted to split the linelist by more than one grouping column, such as to produce subset linelist by intersection of hospital AND gender, you will need a different approach to naming the list elements. This involves collecting the unique “group keys” using group\_keys() from **dplyr** - they are returned as a data frame. Then you can combine the group keys into values with unite() as shown below, and assign these conglomerate names to linelist\_split.

Now we combine the groupings together, separated by dashes, and assign them as the names of list elements in linelist\_split. This takes some extra lines as we replace NA with “Missing”, use unite() from **dplyr** to combine the column values together (separated by dashes), and then convert into an un-named vector so it can be used as names of linelist\_split.

#### Export as Excel sheets

To export the hospital linelists as an Excel workbook with one linelist per sheet, we can just provide the named list linelist\_split to the write\_xlsx() function from the **writexl** package. This has the ability to save one Excel workbook with multiple sheets. The list element names are automatically applied as the sheet names.

You can now open the Excel file and see that each hospital has its own sheet.

#### Export as CSV files

It is a bit more complex command, but you can also export each hospital-specific linelist as a separate CSV file, with a file name specific to the hospital.

Again we use map(): we take the vector of list element names (shown above) and use map() to iterate through them, applying export() (from the **rio** package, see [Import and export](#import-and-export) page) on the data frame in the list linelist\_split that has that name. We also use the name to create a unique file name. Here is how it works:

* We begin with the vector of character names, passed to map() as .x
* The .f function is export() , which requires a data frame and a file path to write to
* The input .x (the hospital name) is used within .f to extract/index that specific element of linelist\_split list. This results in only one data frame at a time being provided to export().
* For example, when map() iterates for “Military Hospital”, then linelist\_split[[.x]] is actually linelist\_split[["Military Hospital"]], thus returning the second element of linelist\_split - which is all the cases from Military Hospital.
* The file path provided to export() is dynamic via use of str\_glue() (see [Characters and strings](#characters-and-strings) page):
  + here() is used to get the base of the file path and specify the “data” folder (note single quotes to not interrupt the str\_glue() double quotes)
* Then a slash /, and then again the .x which prints the current hospital name to make the file identifiable
* Finally the extension “.csv” which export() uses to create a CSV file

Now you can see that each file is saved in the “data” folder of the R Project “Epi\_R\_handbook”!

### Custom functions

You may want to create your own function to provide to map().

Let’s say we want to create epidemic curves for each hospital’s cases. To do this using **purrr**, our .f function can be ggplot() and extensions with + as usual. As the output of map() is always a list, the plots are stored in a list. Because they are plots, they can be extracted and plotted with the ggarrange() function from the **ggpubr** package ([documentation](https://rpkgs.datanovia.com/ggpubr/reference/ggarrange.html)).

If this map() code looks too messy, you can achieve the same result by saving your specific ggplot() command as a custom user-defined function, for example we can name it make\_epicurve()). This function is then used within the map(). .x will be iteratively replaced by the hospital name, and used as hosp\_name in the make\_epicurve() function. See the page on [Writing functions](#writing-functions-1).

### Mapping a function across columns

Another common use-case is to map a function across many columns. Below, we map() the function t.test() across numeric columns in the data frame linelist, comparing the numeric values by gender.

Recall from the page on [Simple statistical tests](#simple-statistical-tests) that t.test() can take inputs in a formula format, such as t.test(numeric column ~ binary column). In this example, we do the following:

* The numeric columns of interest are selected from linelist - these become the .x inputs to map()
* The function t.test() is supplied as the .f function, which is applied to each numeric column
* Within the parentheses of t.test():
  + the first ~ precedes the .f that map() will iterate over .x
  + the .x represents the current column being supplied to the function t.test()
  + the second ~ is part of the t-test equation described above
  + the t.test() function expects a binary column on the right-hand side of the equation. We supply the vector linelist$gender independently and statically (note that it is not included in select()).

map() returns a list, so the output is a list of t-test results - one list element for each numeric column analysed.

Here is what the list t.test\_results looks like when opened (Viewed) in RStudio. We have highlighted parts that are important for the examples in this page.

* You can see at the top that the whole list is named t.test\_results and has five elements. Those five elements are named age, wt\_km, ht\_cm, ct\_blood, temp after each variable that was used in a t-test with gender from the linelist.
* Each of those five elements are themselves lists, with elements within them such as p.value and conf.int. Some of these elements like p.value are single numbers, whereas some such as estimate consist of two or more elements (mean in group f and mean in group m).

Note: Remember that if you want to apply a function to only certain columns in a data frame, you can also simply use mutate() and across(), as explained in the [Cleaning data and core functions](#cleaning-data-and-core-functions) page. Below is an example of applying as.character() to only the “age” columns. Note the placement of the parentheses and commas.

### Extract from lists

As map() produces an output of class List, we will spend some time discussing how to extract data from lists using accompanying **purrr** functions. To demonstrate this, we will use the list t.test\_results from the previous section. This is a list of 5 lists - each of the 5 lists contains the results of a t-test between a column from linelist data frame and its binary column gender. See the image in the section above for a visual of the list structure.

#### Names of elements

To extract the names of the elements themselves, simply use names() from **base** R. In this case, we use names() on t.test\_results to return the names of each sub-list, which are the names of the 5 variables that had t-tests performed.

#### Elements by name or position

To extract list elements by name or by position you can use brackets [[ ]] as described in the [R basics](#r-basics) page. Below we use double brackets to index the list t.tests\_results and display the first element which is the results of the t-test on age.

However, below we will demonstrate use of the simple and flexible **purrr** functions map() and pluck() to achieve the same outcomes.

#### pluck()

pluck() pulls out elements by name or by position. For example - to extract the t-test results for age, you can use pluck() like this:

Index deeper levels by specifying the further levels with commas. The below extracts the element named “p.value” from the list age within the list t.test\_results. You can also use numbers instead of character names.

You can extract such inner elements from all first-level elements by using map() to run the pluck() function across each first-level element. For example, the below code extracts the “p.value” elements from all lists within t.test\_results. The list of t-test results is the .x iterated across, pluck() is the .f function being iterated, and the value “p-value” is provided to the function.

As another alternative, map() offers a shorthand where you can write the element name in quotes, and it will pluck it out. If you use map() the output will be a list, whereas if you use map\_chr() it will be a named character vector and if you use map\_dbl() it will be a named numeric vector.

You can read more about pluck() in it’s **purrr** [documentation](https://purrr.tidyverse.org/reference/pluck.html). It has a sibling function chuck() that will return an error instead of NULL if an element does not exist.

### Convert list to data frame

This is a complex topic - see the Resources section for more complete tutorials. Nevertheless, we will demonstrate converting the list of t-test results into a data frame. We will create a data frame with columns for the variable, its p-value, and the means from the two groups (male and female).

Here are some of the new approaches and functions that will be used:

* The function tibble() will be used to create a tibble (like a data frame)
  + We surround the tibble() function with curly brackets { } to prevent the entire t.test\_results from being stored as the first tibble column
* Within tibble(), each column is created explicitly, similar to the syntax of mutate():
  + The . represents t.test\_results
  + To create a column with the t-test variable names (the names of each list element) we use names() as described above
  + To create a column with the p-values we use map\_dbl() as described above to pull the p.value elements and convert them to a numeric vector

But now let’s add columns containing the means for each group (males and females).

We would need to extract the element estimate, but this actually contains two elements within it (mean in group f and mean in group m). So, it cannot be simplified into a vector with map\_chr() or map\_dbl(). Instead, we use map(), which used within tibble() will create a column of class list within the tibble! Yes, this is possible!

Once you have this list column, there are several **tidyr** functions (part of **tidyverse**) that help you “rectangle” or “un-nest” these “nested list” columns. Read more about them here, or by running vignette("rectangle"). In brief:

* unnest\_wider() - gives each element of a list-column its own column
* unnest\_longer() - gives each element of a list-column its own row
* hoist() - acts like unnest\_wider() but you specify which elements to unnest

Below, we pass the tibble to unnest\_wider() specifying the tibble’s means column (which is a nested list). The result is that means is replaced by two new columns, each reflecting the two elements that were previously in each means cell.

### Discard, keep, and compact lists

Because working with **purrr** so often involves lists, we will briefly explore some **purrr** functions to modify lists. See the Resources section for more complete tutorials on **purrr** functions.

* list\_modify() has many uses, one of which can be to remove a list element
* keep() retains the elements specified to .p =, or where a function supplied to .p = evaluates to TRUE
* discard() removes the elements specified to .p, or where a function supplied to .p = evaluates to TRUE
* compact() removes all empty elements

Here are some examples using the combined list created in the section above on [using map() to import and combine multiple files](#iter_combined) (it contains 6 case linelist data frames):

Elements can be removed by name with list\_modify() and setting the name equal to NULL.

You can also remove elements by criteria, by providing a “predicate” equation to .p = (an equation that evaluates to either TRUE or FALSE). Place a tilde ~ before the function and use .x to represent the list element. Using keep() the list elements that evaluate to TRUE will be kept. Inversely, if using discard() the list elements that evaluate to TRUE will be removed.

In the below example, list elements are discarded if their class are not data frames.

Your predicate function can also reference elements/columns within each list item. For example, below, list elements where the mean of column ct\_blood is over 25 are discarded.

This command would remove all empty list elements:

### pmap()

THIS SECTION IS UNDER CONSTRUCTION

## Apply functions

The “apply” family of functions is a **base** R alternative to **purrr** for iterative operations. You can read more about them [here](https://www.datacamp.com/community/tutorials/r-tutorial-apply-family).

## Resources

[for loops with Data Carpentry](https://datacarpentry.org/semester-biology/materials/for-loops-R/)

The [R for Data Science page on iteration](https://r4ds.had.co.nz/iteration.html#iteration)

[Vignette on write/read Excel files](https://martinctc.github.io/blog/vignette-write-and-read-multiple-excel-files-with-purrr/)

A purrr [tutorial](https://jennybc.github.io/purrr-tutorial/index.html) by jennybc

Another purrr [tutorial](http://www.rebeccabarter.com/blog/2019-08-19_purrr/) by Rebecca Barter

A purrr [tutorial](http://zevross.com/blog/2019/06/11/the-power-of-three-purrr-poseful-iteration-in-r-with-map-pmap-and-imap/) on map, pmap, and imap

[purrr cheatsheet](https://raw.githubusercontent.com/rstudio/cheatsheets/master/pngs/thumbnails/purrr-cheatsheet-thumbs.png)

[purrr tips and tricks](https://www.hvitfeldt.me/blog/purrr-tips-and-tricks/)

[keep and discard](https://hookedondata.org/going-off-the-map/#keep-and-discard)

# IV Analysis

# Descriptive tables

This page demonstrates the use of **janitor**, **dplyr**, **gtsummary**, **rstatix**, and **base** R to summarise data and create tables with descriptive statistics.

This page covers how to create\* the underlying tables, whereas the [Tables for presentation](#tables-for-presentation) page covers how to nicely format and print them.\*

Each of these packages has advantages and disadvantages in the areas of code simplicity, accessibility of outputs, quality of printed outputs. Use this page to decide which approach works for your scenario.

You have several choices when producing tabulation and cross-tabulation summary tables. Some of the factors to consider include code simplicity, customizeability, the desired output (printed to R console, as data frame, or as “pretty” .png/.jpeg/.html image), and ease of post-processing. Consider the points below as you choose the tool for your situation.

* Use tabyl() from **janitor** to produce and “adorn” tabulations and cross-tabulations
* Use get\_summary\_stats() from **rstatix** to easily generate data frames of numeric summary statistics for multiple columns and/or groups
* Use summarise() and count() from **dplyr** for more complex statistics, tidy data frame outputs, or preparing data for ggplot()
* Use tbl\_summary() from **gtsummary** to produce detailed publication-ready tables
* Use table() from **base** R if you do not have access to the above packages

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Browse data

### ****skimr**** package

By using the **skimr** package, you can get a detailed and aesthetically pleasing overview of each of the variables in your dataset. Read more about **skimr** at its [github page](https://github.com/ropensci/skimr).

Below, the function skim() is applied to the entire linelist data frame. An overview of the data frame and a summary of every column (by class) is produced.

You can also use the summary() function, from **base** R, to get information about an entire dataset, but this output can be more difficult to read than using **skimr**. Therefore the output is not shown below, to conserve page space.

### Summary statistics

You can use **base** R functions to return summary statistics on a numeric column. You can return most of the useful summary statistics for a numeric column using summary(), as below. Note that the data frame name must also be specified as shown below.

You can access and save one specific part of it with index brackets [ ]:

You can return individual statistics with **base** R functions like max(), min(), median(), mean(), quantile(), sd(), and range(). See the [R basics](#r-basics) page for a complete list.

**CAUTION:** If your data contain missing values, R wants you to know this and so will return NA unless you specify to the above mathematical functions that you want R to ignore missing values, via the argument na.rm = TRUE.

You can use the get\_summary\_stats() function from **rstatix** to return summary statistics in a data frame format. This can be helpful for performing subsequent operations or plotting on the numbers. See the [Simple statistical tests](#simple-statistical-tests) page for more details on the **rstatix** package and its functions.

## ****janitor**** package

The **janitor** packages offers the tabyl() function to produce tabulations and cross-tabulations, which can be “adorned” or modified with helper functions to display percents, proportions, counts, etc.

Below, we pipe the linelist data frame to **janitor** functions and print the result. If desired, you can also save the resulting tables with the assignment operator <-.

### Simple tabyl

The default use of tabyl() on a specific column produces the unique values, counts, and column-wise “percents” (actually proportions). The proportions may have many digits. You can adjust the number of decimals with adorn\_rounding() as described below.

As you can see above, if there are missing values they display in a row labeled <NA>. You can suppress them with show\_na = FALSE. If there are no missing values, this row will not appear. If there are missing values, all proportions are given as both raw (denominator inclusive of NA counts) and “valid” (denominator excludes NA counts).

If the column is class Factor and only certain levels are present in your data, all levels will still appear in the table. You can suppress this feature by specifying show\_missing\_levels = FALSE. Read more on the [Factors](#factors) page.

### Cross-tabulation

Cross-tabulation counts are achieved by adding one or more additional columns within tabyl(). Note that now only counts are returned - proportions and percents can be added with additional steps shown below.

### “Adorning” the tabyl

Use **janitor**’s “adorn” functions to add totals or convert to proportions, percents, or otherwise adjust the display. Often, you will pipe the tabyl through several of these functions.

| **Function** | **Outcome** |
| --- | --- |
| adorn\_totals() | Adds totals (where = “row”, “col”, or “both”). Set name = for “Total”. |
| adorn\_percentages() | Convert counts to proportions, with denominator = “row”, “col”, or “all” |
| adorn\_pct\_formatting() | Converts proportions to percents. Specify digits =. Remove the “%” symbol with affix\_sign = FALSE. |
| adorn\_rounding() | To round proportions to digits = places. To round percents use adorn\_pct\_formatting() with digits =. |
| adorn\_ns() | Add counts to a table of proportions or percents. Indicate position = “rear” to show counts in parentheses, or “front” to put the percents in parentheses. |
| adorn\_title() | Add string via arguments row\_name = and/or col\_name = |

Be conscious of the order you apply the above functions. Below are some examples.

A simple one-way table with percents instead of the default proportions.

A cross-tabulation with a total row and row percents.

A cross-tabulation adjusted so that both counts and percents are displayed.

### Printing the tabyl

By default, the tabyl will print raw to your R console.

Alternatively, you can pass the tabyl to **flextable** or similar package to print as a “pretty” image in the RStudio Viewer, which could be exported as .png, .jpeg, .html, etc. This is discussed in the page [Tables for presentation](#tables-for-presentation). Note that if printing in this manner and using adorn\_titles(), you must specify placement = "combined".

### Use on other tables

You can use **janitor**’s adorn\_\*() functions on other tables, such as those created by summarise() and count() from **dplyr**, or table() from **base** R. Simply pipe the table to the desired **janitor** function. For example:

### Saving the tabyl

If you convert the table to a “pretty” image with a package like **flextable**, you can save it with functions from that package - like save\_as\_html(), save\_as\_word(), save\_as\_ppt(), and save\_as\_image() from **flextable** (as discussed more extensively in the [Tables for presentation](#tables-for-presentation) page). Below, the table is saved as a Word document, in which it can be further hand-edited.

### Statistics

You can apply statistical tests on tabyls, like chisq.test() or fisher.test() from the **stats** package, as shown below. Note missing values are not allowed so they are excluded from the tabyl with show\_na = FALSE.

See the page on [Simple statistical tests](#simple-statistical-tests) for more code and tips about statistics.

### Other tips

* Include the argument na.rm = TRUE to exclude missing values from any of the above calculations.
* If applying any adorn\_\*() helper functions to tables not created by tabyl(), you can specify particular column(s) to apply them to like adorn\_percentage(,,,c(cases,deaths)) (specify them to the 4th unnamed argument). The syntax is not simple. Consider using summarise() instead.
* You can read more detail in the [janitor page](https://cran.r-project.org/web/packages/janitor/vignettes/janitor.html) and this [tabyl vignette](https://cran.r-project.org/web/packages/janitor/vignettes/tabyls.html).

## ****dplyr**** package

**dplyr** is part of the **tidyverse** packages and is an very common data management tool. Creating tables with **dplyr** functions summarise() and count() is a useful approach to calculating summary statistics, summarize by group, or pass tables to ggplot().

summarise() creates a new, summary data frame. If the data are ungrouped, it will return a one-row dataframe with the specified summary statistics of the entire data frame. If the data are grouped, the new data frame will have one row per group (see [Grouping data](#grouping-data) page).

Within the summarise() parentheses, you provide the names of each new summary column followed by an equals sign and a statistical function to apply.

**TIP:** The summarise function works with both UK and US spelling (summarise() and summarize()).

### Get counts

The most simple function to apply within summarise() is n(). Leave the parentheses empty to count the number of rows.

This gets more interesting if we have grouped the data beforehand.

The above command can be shortened by using the count() function instead. count() does the following:

1. Groups the data by the columns provided to it
2. Summarises them with n() (creating column n)
3. Un-groups the data

You can change the name of the counts column from the default n to something else by specifying it to name =.

Tabulating counts of two or more grouping columns are still returned in “long” format, with the counts in the n column. See the page on [Pivoting data](#pivoting-data) to learn about “long” and “wide” data formats.

### Show all levels

If you are tabling a column of class factor you can ensure that all levels are shown (not just the levels with values in the data) by adding .drop = FALSE into the summarise() or count() command.

This technique is useful to standardise your tables/plots. For example if you are creating figures for multiple sub-groups, or repeatedly creating the figure for routine reports. In each of these circumstances, the presence of values in the data may fluctuate, but you can define levels that remain constant.

See the page on [Factors](#factors) for more information.

### Proportions

Proportions can be added by piping the table to mutate() to create a new column. Define the new column as the counts column (n by default) divided by the sum() of the counts column (this will return a proportion).

Note that in this case, sum() in the mutate() command will return the sum of the whole column n for use as the proportion denominator. As explained [in the Grouping data page](#group_summarise), if sum() is used in grouped data (e.g. if the mutate() immediately followed a group\_by() command), it will return sums by group. As stated just above, count() finishes its actions by ungrouping. Thus, in this scenario we get full column proportions.

To easily display percents, you can wrap the proportion in the function percent() from the package **scales** (note this convert to class character).

Below is a method to calculate proportions within groups. It relies on different levels of data grouping being selectively applied and removed. First, the data are grouped on outcome via group\_by(). Then, count() is applied. This function further groups the data by age\_cat and returns counts for each outcome-age-cat combination. Importantly - as it finishes its process, count() also ungroups the age\_cat grouping, so the only remaining data grouping is the original grouping by outcome. Thus, the final step of calculating proportions (denominator sum(n)) is still grouped by outcome.

### Plotting

To display a “long” table output like the above with ggplot() is relatively straight-forward. The data are naturally in “long” format, which is naturally accepted by ggplot(). See further examples in the pages [ggplot basics](#ggplot-basics) and [ggplot tips](#ggplot-tips).

### Summary statistics

One major advantage of **dplyr** and summarise() is the ability to return more advanced statistical summaries like median(), mean(), max(), min(), sd() (standard deviation), and percentiles. You can also use sum() to return the number of rows that meet certain logical criteria. As above, these outputs can be produced for the whole data frame set, or by group.

The syntax is the same - within the summarise() parentheses you provide the names of each new summary column followed by an equals sign and a statistical function to apply. Within the statistical function, give the column(s) to be operated on and any relevant arguments (e.g. na.rm = TRUE for most mathematical functions).

You can also use sum() to return the number of rows that meet a logical criteria. The expression within is counted if it evaluates to TRUE. For example:

* sum(age\_years < 18, na.rm=T)
* sum(gender == "male", na.rm=T)
* sum(response %in% c("Likely", "Very Likely"))

Below, linelist data are summarised to describe the days delay from symptom onset to hospital admission (column days\_onset\_hosp), by hospital.

Some tips:

* Use sum() with a logic statement to “count” rows that meet certain criteria (==)
* Note the use of na.rm = TRUE within mathematical functions like sum(), otherwise NA will be returned if there are any missing values
* Use the function percent() from the **scales** package to easily convert to percents
  + Set accuracy = to 0.1 or 0.01 to ensure 1 or 2 decimal places respectively
* Use round() from **base** R to specify decimals
* To calculate these statistics on the entire dataset, use summarise() without group\_by()
* You may create columns for the purposes of later calculations (e.g. denominators) that you eventually drop from your data frame with select().

### Conditional statistics

You may want to return conditional statistics - e.g. the maximum of rows that meet certain criteria. This can be done by subsetting the column with brackets [ ]. The example below returns the maximum temperature for patients classified having or not having fever. Be aware however - it may be more appropriate to add another column to the group\_by() command and pivot\_wider() (as demonstrated [below](#tbls_pivot_wider)).

### Glueing together

The function str\_glue() from **stringr** is useful to combine values from several columns into one new column. In this context this is typically used after the summarise() command.

In the [Characters and strings](#characters-and-strings) page, various options for combining columns are discussed, including unite(), and paste0(). In this use case, we advocate for str\_glue() because it is more flexible than unite() and has more simple syntax than paste0().

Below, the summary\_table data frame (created above) is mutated such that columns delay\_mean and delay\_sd are combined, parentheses formating is added to the new column, and their respective old columns are removed.

Then, to make the table more presentable, a total row is added with adorn\_totals() from **janitor** (which ignores non-numeric columns). Lastly, we use select() from **dplyr** to both re-order and rename to nicer column names.

Now you could pass to **flextable** and print the table to Word, .png, .jpeg, .html, Powerpoint, RMarkdown, etc.! (see the [Tables for presentation](#tables-for-presentation) page).

#### Percentiles

Percentiles and quantiles in **dplyr** deserve a special mention. To return quantiles, use quantile() with the defaults or specify the value(s) you would like with probs =.

If you want to return quantiles by group, you may encounter long and less useful outputs if you simply add another column to group\_by(). So, try this approach instead - create a column for each quantile level desired.

While **dplyr** summarise() certainly offers more fine control, you may find that all the summary statistics you need can be produced with get\_summary\_stat() from the **rstatix** package. If operating on grouped data, if will return 0%, 25%, 50%, 75%, and 100%. If applied to ungrouped data, you can specify the percentiles with probs = c(.05, .5, .75, .98).

### Summarise aggregated data

If you begin with aggregated data, using n() return the number of rows, not the sum of the aggregated counts. To get sums, use sum() on the data’s counts column.

For example, let’s say you are beginning with the data frame of counts below, called linelist\_agg - it shows in “long” format the case counts by outcome and gender.

Below we create this example data frame of linelist case counts by outcome and gender (missing values removed for clarity).

To sum the counts (in column n) by group you can use summarise() but set the new column equal to sum(n, na.rm=T). To add a conditional element to the sum operation, you can use the subset bracket [ ] syntax on the counts column.

### across() multiple columns

You can use summarise() across multiple columns using across(). This makes life easier when you want to calculate the same statistics for many columns. Place across() within summarise() and specify the following:

* .cols = as either a vector of column names c() or “tidyselect” helper functions (explained below)
* .fns = the function to perform (no parentheses) - you can provide multiple within a list()

Below, mean() is applied to several numeric columns. A vector of columns are named explicitly to .cols = and a single function mean is specified (no parentheses) to .fns =. Any additional arguments for the function (e.g. na.rm=TRUE) are provided after .fns =, separated by a comma.

It can be difficult to get the order of parentheses and commas correct when using across(). Remember that within across() you must include the columns, the functions, and any extra arguments needed for the functions.

Multiple functions can be run at once. Below the functions mean and sd are provided to .fns = within a list(). You have the opportunity to provide character names (e.g. “mean” and “sd”) which are appended in the new column names.

Here are those “tidyselect” helper functions you can provide to .cols = to select columns:

* everything() - all other columns not mentioned
* last\_col() - the last column
* where() - applies a function to all columns and selects those which are TRUE
* starts\_with() - matches to a specified prefix. Example: starts\_with("date")
* ends\_with() - matches to a specified suffix. Example: ends\_with("\_end")
* contains() - columns containing a character string. Example: contains("time")
* matches() - to apply a regular expression (regex). Example: contains("[pt]al")
* num\_range() -
* any\_of() - matches if column is named. Useful if the name might not exist. Example: any\_of(date\_onset, date\_death, cardiac\_arrest)

For example, to return the mean of every numeric column use where() and provide the function as.numeric() (without parentheses). All this remains within the across() command.

### Pivot wider

If you prefer your table in “wide” format you can transform it using the **tidyr** pivot\_wider() function. You will likely need to re-name the columns with rename(). For more information see the page on [Pivoting data](#pivoting-data).

The example below begins with the “long” table age\_by\_outcome from the [proportions section](#tbl_dplyr_prop). We create it again and print, for clarity:

To pivot wider, we create the new columns from the values in the existing column age\_cat (by setting names\_from = age\_cat). We also specify that the new table values will come from the existing column n, with values\_from = n. The columns not mentioned in our pivoting command (outcome) will remain unchanged on the far left side.

### Total rows

When summarise() operates on grouped data it does not automatically produce “total” statistics. Below, two approaches to adding a total row are presented:

#### ****janitor****’s adorn\_totals()

If your table consists only of counts or proportions/percents that can be summed into a total, then you can add sum totals using **janitor**’s adorn\_totals() as described in the section above. Note that this function can only sum the numeric columns - if you want to calculate other total summary statistics see the next approach with **dplyr**.

Below, linelist is grouped by gender and summarised into a table that described the number of cases with known outcome, deaths, and recovered. Piping the table to adorn\_totals() adds a total row at the bottom reflecting the sum of each column. The further adorn\_\*() functions adjust the display as noted in the code.

#### summarise() on “total” data and then bind\_rows()

If your table consists of summary statistics such as median(), mean(), etc, the adorn\_totals() approach shown above will not be sufficient. Instead, to get summary statistics for the entire dataset you must calculate them with a separate summarise() command and then bind the results to the original grouped summary table. To do the binding you can use bind\_rows() from **dplyr** s described in the [Joining data](#joining-data) page. Below is an example:

You can make a summary table of outcome by hospital with group\_by() and summarise() like this:

To get the totals, run the same summarise() command but only group the data by outcome (not by hospital), like this:

We can bind these two data frames together. Note that by\_hospital has 4 columns whereas totals has 3 columns. By using bind\_rows(), the columns are combined by name, and any extra space is filled in with NA (e.g the column hospital values for the two new totals rows). After binding the rows, we convert these empty spaces to “Total” using replace\_na() (see [Cleaning data and core functions](#cleaning-data-and-core-functions) page).

Here is the new table with “Total” rows at the bottom.

This table is in a “long” format, which may be what you want. Optionally, you can pivot this table wider to make it more readable. See the section on pivoting wider above, and the [Pivoting data](#pivoting-data) page. You can also add more columns, and arrange it nicely. This code is below.

And then you can print this nicely as an image - below is the output printed with **flextable**. You can read more in depth about this example and how to achieve this “pretty” table in the [Tables for presentation](#tables-for-presentation) page.

## ****gtsummary**** package

If you want to print your summary statistics in a pretty, publication-ready graphic, you can use the **gtsummary** package and its function tbl\_summary(). The code can seem complex at first, but the outputs look very nice and print to your RStudio Viewer panel as an HTML image. Read a [vignette here](http://www.danieldsjoberg.com/gtsummary/articles/tbl_summary.html).

You can also add the results of statistical tests to **gtsummary** tables. This process is described in the **gtsummary** section of the [Simple statistical tests](#stats_gt) page.

To introduce tbl\_summary() we will show the most basic behavior first, which actually produces a large and beautiful table. Then, we will examine in detail how to make adjustments and more tailored tables.

### Summary table

The default behavior of tbl\_summary() is quite incredible - it takes the columns you provide and creates a summary table in one command. The function prints statistics appropriate to the column class: median and inter-quartile range (IQR) for numeric columns, and counts (%) for categorical columns. Missing values are converted to “Unknown”. Footnotes are added to the bottom to explain the statistics, while the total N is shown at the top.

### Adjustments

Now we will explain how the function works and how to make adjustments. The key arguments are detailed below:

**by =**  
You can stratify your table by a column (e.g. by outcome), creating a 2-way table.

**statistic =**  
Use an equations to specify which statistics to show and how to display them. There are two sides to the equation, separated by a tilde ~. On the right side, in quotes, is the statistical display desired, and on the left are the columns to which that display will apply.

* The right side of the equation uses the syntax of str\_glue() from **stringr** (see [Characters and Strings](#characters-and-strings)), with the desired display string in quotes and the statistics themselves within curly brackets. You can include statistics like “n” (for counts), “N” (for denominator), “mean”, “median”, “sd”, “max”, “min”, percentiles as “p##” like “p25”, or percent of total as “p”. See ?tbl\_summary for details.
* For the left side of the equation, you can specify columns by name (e.g. age or c(age, gender)) or using helpers such as all\_continuous(), all\_categorical(), contains(), starts\_with(), etc.

A simple example of a statistic = equation might look like below, to only print the mean of column age\_years:

A slightly more complex equation might look like "({min}, {max})", incorporating the max and min values within parentheses and separated by a comma:

You can also differentiate syntax for separate columns or types of columns. In the more complex example below, the value provided to statistc = is a **list** indicating that for all continuous columns the table should print mean with standard deviation in parentheses, while for all categorical columns it should print the n, denominator, and percent.

**digits =**  
Adjust the digits and rounding. Optionally, this can be specified to be for continuous columns only (as below).

**label =**  
Adjust how the column name should be displayed. Provide the column name and its desired label separated by a tilde. The default is the column name.

**missing\_text =**  
Adjust how missing values are displayed. The default is “Unknown”.

**type =**  
This is used to adjust how many levels of the statistics are shown. The syntax is similar to statistic = in that you provide an equation with columns on the left and a value on the right. Two common scenarios include:

* type = all\_categorical() ~ "categorical" Forces dichotomous columns (e.g. fever yes/no) to show all levels instead of only the “yes” row
* type = all\_continuous() ~ "continuous2" Allows multi-line statistics per variable, as shown in a later section

In the example below, each of these arguments is used to modify the original summary table:

### Multi-line stats for continuous variables

If you want to print multiple lines of statistics for continuous variables, you can indicate this by setting the type = to “continuous2”. You can combine all of the previously shown elements in one table by choosing which statistics you want to show. To do this you need to tell the function that you want to get a table back by entering the type as “continuous2”. The number of missing values is shown as “Unknown”.

There are many other ways to modify these tables, including adding p-values, adjusting color and headings, etc. Many of these are described in the documentation (enter ?tbl\_summary in Console), and some are given in the section on [statistical tests](https://epirhandbook.com/simple-statistical-tests.html).

## ****base**** R

You can use the function table() to tabulate and cross-tabulate columns. Unlike the options above, you must specify the dataframe each time you reference a column name, as shown below.

**CAUTION:** NA (missing) values will **not** be tabulated unless you include the argument useNA = "always" (which could also be set to “no” or “ifany”).

**TIP:** You can use the %$% from **magrittr** to remove the need for repeating data frame calls within **base** functions. For example the below could be written linelist %$% table(outcome, useNA = "always")

Multiple columns can be cross-tabulated by listing them one after the other, separated by commas. Optionally, you can assign each column a “name” like Outcome = linelist$outcome.

### Proportions

To return proportions, passing the above table to the function prop.table(). Use the margins = argument to specify whether you want the proportions to be of rows (1), of columns (2), or of the whole table (3). For clarity, we pipe the table to the round() function from **base** R, specifying 2 digits.

### Totals

To add row and column totals, pass the table to addmargins(). This works for both counts and proportions.

### Convert to data frame

Converting a table() object directly to a data frame is not straight-forward. One approach is demonstrated below:

1. Create the table, without using useNA = "always". Instead convert NA values to “(Missing)” with fct\_explicit\_na() from **forcats**.
2. Add totals (optional) by piping to addmargins()
3. Pipe to the **base** R function as.data.frame.matrix()
4. Pipe the table to the **tibble** function rownames\_to\_column(), specifying the name for the first column
5. Print, View, or export as desired. In this example we use flextable() from package **flextable** as described in the [Tables for presentation](#tables-for-presentation) page. This will print to the RStudio viewer pane as a pretty HTML image.

## Resources

Much of the information in this page is adapted from these resources and vignettes online:

[gtsummary](http://www.danieldsjoberg.com/gtsummary/articles/tbl_summary.html)

[dplyr](https://dplyr.tidyverse.org/articles/grouping.html)

# Simple statistical tests

This page demonstrates how to conduct simple statistical tests using **base** R, **rstatix**, and **gtsummary**.

* T-test
* Shapiro-Wilk test
* Wilcoxon rank sum test
* Kruskal-Wallis test
* Chi-squared test
* Correlations between numeric variables

…many other tests can be performed, but we showcase just these common ones and link to further documentation.

Each of the above packages bring certain advantages and disadvantages:

* Use **base** R functions to print a statistical outputs to the R Console
* Use **rstatix** functions to return results in a data frame, or if you want tests to run by group
* Use **gtsummary** if you want to quickly print publication-ready tables

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## ****base**** R

You can use **base** R functions to conduct statistical tests. The commands are relatively simple and results will print to the R Console for simple viewing. However, the outputs are usually lists and so are harder to manipulate if you want to use the results in subsequent operations.

### T-tests

A [t-test](https://en.wikipedia.org/wiki/Student%27s_t-test), also called “Student’s t-Test”, is typically used to determine if there is a significant difference between the means of some numeric variable between two groups. Here we’ll show the syntax to do this test depending on whether the columns are in the same data frame.

**Syntax 1:** This is the syntax when your numeric and categorical columns are in the same data frame. Provide the numeric column on the left side of the equation and the categorical column on the right side. Specify the dataset to data =. Optionally, set paired = TRUE, and conf.level = (0.95 default), and alternative = (either “two.sided”, “less”, or “greater”). Enter ?t.test for more details.

**Syntax 2:** You can compare two separate numeric vectors using this alternative syntax. For example, if the two columns are in different data sets.

You can also use a t-test to determine whether a sample mean is significantly different from some specific value. Here we conduct a one-sample t-test with the known/hypothesized population mean as mu =:

### Shapiro-Wilk test

The [Shapiro-Wilk test](https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test) can be used to determine whether a sample came from a normally-distributed population (an assumption of many other tests and analysis, such as the t-test). However, this can only be used on a sample between 3 and 5000 observations. For larger samples a [quantile-quantile plot](https://ggplot2.tidyverse.org/reference/geom_qq.html) may be helpful.

### Wilcoxon rank sum test

The Wilcoxon rank sum test, also called the [Mann–Whitney U test](https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test), is often used to help determine if two numeric samples are from the same distribution when their populations are not normally distributed or have unequal variance.

### Kruskal-Wallis test

The [Kruskal-Wallis test](https://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_one-way_analysis_of_variance) is an extension of the Wilcoxon rank sum test that can be used to test for differences in the distribution of more than two samples. When only two samples are used it gives identical results to the Wilcoxon rank sum test.

### Chi-squared test

[Pearson’s Chi-squared test](https://en.wikipedia.org/wiki/Chi-squared_test) is used in testing for significant differences between categorical croups.

## ****rstatix**** package

The **rstatix** package offers the ability to run statistical tests and retrieve results in a “pipe-friendly” framework. The results are automatically in a data frame so that you can perform subsequent operations on the results. It is also easy to group the data being passed into the functions, so that the statistics are run for each group.

### Summary statistics

The function get\_summary\_stats() is a quick way to return summary statistics. Simply pipe your dataset to this function and provide the columns to analyse. If no columns are specified, the statistics are calculated for all columns.

By default, a full range of summary statistics are returned: n, max, min, median, 25%ile, 75%ile, IQR, median absolute deviation (mad), mean, standard deviation, standard error, and a confidence interval of the mean.

You can specify a subset of summary statistics to return by providing one of the following values to type =: “full”, “common”, “robust”, “five\_number”, “mean\_sd”, “mean\_se”, “mean\_ci”, “median\_iqr”, “median\_mad”, “quantile”, “mean”, “median”, “min”, “max”.

It can be used with grouped data as well, such that a row is returned for each grouping-variable:

You can also use **rstatix** to conduct statistical tests:

### T-test

Use a formula syntax to specify the numeric and categorical columns:

Or use ~ 1 and specify mu = for a one-sample T-test. This can also be done by group.

If applicable, the statistical tests can be done by group, as shown below:

### Shapiro-Wilk test

As stated above, sample size must be between 3 and 5000.

### Wilcoxon rank sum test

### Kruskal-Wallis test

Also known as the Mann-Whitney U test.

### Chi-squared test

The chi-square test function accepts a table, so first we create a cross-tabulation. There are many ways to create a cross-tabulation (see [Descriptive tables](#descriptive-tables)) but here we use tabyl() from **janitor** and remove the left-most column of value labels before passing to chisq\_test().

Many many more functions and statistical tests can be run with **rstatix** functions. See the documentation for **rstatix** [online here](https://github.com/kassambara/rstatix) or by entering ?rstatix.

## gtsummary package

Use **gtsummary** if you are looking to add the results of a statistical test to a pretty table that was created with this package (as described in the **gtsummary** section of the [Descriptive tables](#tbl_gt) page).

Performing statistical tests of comparison with tbl\_summary is done by adding the add\_p function to a table and specifying which test to use. It is possible to get p-values corrected for multiple testing by using the add\_q function. Run ?tbl\_summary for details.

### Chi-squared test

Compare the proportions of a categorical variable in two groups. The default statistical test for add\_p() when applied to a categorical variable is to perform a chi-squared test of independence with continuity correction, but if any expected call count is below 5 then a Fisher’s exact test is used.

### T-tests

Compare the difference in means for a continuous variable in two groups. For example, compare the mean age by patient outcome.

### Wilcoxon rank sum test

Compare the distribution of a continuous variable in two groups. The default is to use the Wilcoxon rank sum test and the median (IQR) when comparing two groups. However for non-normally distributed data or comparing multiple groups, the Kruskal-wallis test is more appropriate.

### Kruskal-wallis test

Compare the distribution of a continuous variable in two or more groups, regardless of whether the data is normally distributed.

## Correlations

Correlation between numeric variables can be investigated using the **tidyverse**  
**corrr** package. It allows you to compute correlations using Pearson, Kendall tau or Spearman rho. The package creates a table and also has a function to automatically plot the values.

## Resources

Much of the information in this page is adapted from these resources and vignettes online:

[gtsummary](http://www.danieldsjoberg.com/gtsummary/articles/tbl_summary.html) [dplyr](https://dplyr.tidyverse.org/articles/grouping.html) [corrr](https://corrr.tidymodels.org/articles/using-corrr.html) [sthda correlation](http://www.sthda.com/english/wiki/correlation-test-between-two-variables-in-r)

# Univariate and multivariable regression

This page demonstrates the use of **base** R regression functions such as glm() and the **gtsummary** package to look at associations between variables (e.g. odds ratios, risk ratios and hazard ratios). It also uses functions like tidy() from the **broom** package to clean-up regression outputs.

1. Univariate: two-by-two tables
2. Stratified: mantel-haenszel estimates
3. Multivariable: variable selection, model selection, final table
4. Forest plots

For Cox proportional hazard regression, see the [Survival analysis](#survival-analysis) page.

**NOTE:** We use the term multivariable to refer to a regression with multiple explanatory variables. In this sense a multivariate model would be a regression with several outcomes - see this [editorial](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3518362/) for detail

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

### Clean data

#### Store explanatory variables

We store the names of the explanatory columns as a character vector. This will be referenced later.

#### Convert to 1’s and 0’s

Below we convert the explanatory columns from “yes”/“no”, “m”/“f”, and “dead”/“alive” to **1 / 0**, to cooperate with the expectations of logistic regression models. To do this efficiently, used across() from **dplyr** to transform multiple columns at one time. The function we apply to each column is case\_when() (also **dplyr**) which applies logic to convert specified values to 1’s and 0’s. See sections on across() and case\_when() in the [Cleaning data and core functions page](#clean_across)).

Note: the “.” below represents the column that is being processed by across() at that moment.

#### Drop rows with missing values

To drop rows with missing values, can use the **tidyr** function drop\_na(). However, we only want to do this for rows that are missing values in the columns of interest.

The first thing we must to is make sure our explanatory\_vars vector includes the column age (age would have produced an error in the previous case\_when() operation, which was only for dichotomous variables). Then we pipe the linelist to drop\_na() to remove any rows with missing values in the outcome column or any of the explanatory\_vars columns.

Before running the code, the number of rows in the linelist is nrow(linelist).

The number of rows remaining in linelist is nrow(linelist).

## Univariate

Just like in the page on [Descriptive tables](https://epirhandbook.com/descriptive-tables.html), your use case will determine which R package you use. We present two options for doing univariate analysis:

* Use functions available in **base** R to quickly print results to the console. Use the **broom** package to tidy up the outputs.
* Use the **gtsummary** package to model and get publication-ready outputs

### ****base**** R

#### Linear regression

The **base** R function lm() perform linear regression, assessing the relationship between numeric response and explanatory variables that are assumed to have a linear relationship.

Provide the equation as a formula, with the response and explanatory column names separated by a tilde ~. Also, specify the dataset to data =. Define the model results as an R object, to use later.

You can then run summary() on the model results to see the coefficients (Estimates), P-value, residuals, and other measures.

Alternatively you can use the tidy() function from the **broom** package to pull the results in to a table. What the results tell us is that for each year increase in age the height increases by 3.5 cm and this is statistically significant.

You can then also use this regression to add it to a **ggplot**, to do this we first pull the points for the observed data and the fitted line in to one data frame using the augment() function from **broom**.

It is also possible to add a simple linear regression straight straight in **ggplot** using the geom\_smooth() function.

See the Resource section at the end of this chapter for more detailed tutorials.

#### Logistic regression

The function glm() from the **stats** package (part of **base** R) is used to fit Generalized Linear Models (GLM).

glm() can be used for univariate and multivariable logistic regression (e.g. to get Odds Ratios). Here are the core parts:

* formula = The model is provided to glm() as an equation, with the outcome on the left and explanatory variables on the right of a tilde ~.
* family = This determines the type of model to run. For logistic regression, use family = "binomial", for poisson use family = "poisson". Other examples are in the table below.
* data = Specify your data frame

If necessary, you can also specify the link function via the syntax family = familytype(link = "linkfunction")). You can read more in the documentation about other families and optional arguments such as weights = and subset = (?glm).

| **Family** | **Default link function** |
| --- | --- |
| "binomial" | (link = "logit") |
| "gaussian" | (link = "identity") |
| "Gamma" | (link = "inverse") |
| "inverse.gaussian" | (link = "1/mu^2") |
| "poisson" | (link = "log") |
| "quasi" | (link = "identity", variance = "constant") |
| "quasibinomial" | (link = "logit") |
| "quasipoisson" | (link = "log") |

When running glm() it is most common to save the results as a named R object. Then you can print the results to your console using summary() as shown below, or perform other operations on the results (e.g. exponentiate).

If you need to run a negative binomial regression you can use the **MASS** package; the glm.nb() uses the same syntax as glm(). For a walk-through of different regressions, see the [UCLA stats page](https://stats.idre.ucla.edu/other/dae/).

#### Univariate glm()

In this example we are assessing the association between different age categories and the outcome of death (coded as 1 in the Preparation section). Below is a univariate model of outcome by age\_cat. We save the model output as model and then print it with summary() to the console. Note the estimates provided are the log odds and that the baseline level is the first factor level of age\_cat (“0-4”).

To alter the baseline level of a given variable, ensure the column is class Factor and move the desired level to the first position with fct\_relevel() (see page on [Factors](#factors)). For example, below we take column age\_cat and set “20-29” as the baseline before piping the modified data frame into glm().

#### Printing results

For most uses, several modifications must be made to the above outputs. The function tidy() from the package **broom** is convenient for making the model results presentable.

Here we demonstrate how to combine model outputs with a table of counts.

1. Get the exponentiated log odds ratio estimates and confidence intervals by passing the model to tidy() and setting exponentiate = TRUE and conf.int = TRUE.

Below is the outputted tibble model:

1. Combine these model results with a table of counts. Below, we create the a counts cross-table with the tabyl() function from **janitor**, as covered in the [Descriptive tables](#descriptive-tables) page.

Here is what this counts\_table data frame looks like:

Now we can bind the counts\_table and the model results together horizontally with bind\_cols() (**dplyr**). Remember that with bind\_cols() the rows in the two data frames must be aligned perfectly. In this code, because we are binding within a pipe chain, we use . to represent the piped object counts\_table as we bind it to model. To finish the process, we use select() to pick the desired columns and their order, and finally apply the **base** R round() function across all numeric columns to specify 2 decimal places.

Here is what the combined data frame looks like, printed nicely as an image with a function from **flextable**. The [Tables for presentation](#tables-for-presentation) explains how to customize such tables with **flextable**, or or you can use numerous other packages such as **knitr** or **GT**.

#### Looping multiple univariate models

Below we present a method using glm() and tidy() for a more simple approach, see the section on **gtsummary**.

To run the models on several exposure variables to produce univariate odds ratios (i.e. not controlling for each other), you can use the approach below. It uses str\_c() from **stringr** to create univariate formulas (see [Characters and strings](#characters-and-strings)), runs the glm() regression on each formula, passes each glm() output to tidy() and finally collapses all the model outputs together with bind\_rows() from **tidyr**. This approach uses map() from the package **purrr** to iterate - see the page on [Iteration, loops, and lists](#iteration-loops-and-lists) for more information on this tool.

1. Create a vector of column names of the explanatory variables. We already have this as explanatory\_vars from the Preparation section of this page.
2. Use str\_c() to create multiple string formulas, with outcome on the left, and a column name from explanatory\_vars on the right. The period . substitutes for the column name in explanatory\_vars.
3. Pass these string formulas to map() and set ~glm() as the function to apply to each input. Within glm(), set the regression formula as as.formula(.x) where .x will be replaced by the string formula defined in the step above. map() will loop over each of the string formulas, running regressions for each one.
4. The outputs of this first map() are passed to a second map() command, which applies tidy() to the regression outputs.
5. Finally the output of the second map() (a list of tidied data frames) is condensed with bind\_rows(), resulting in one data frame with all the univariate results.

This time, the end object models is longer because it now represents the combined results of several univariate regressions. Click through to see all the rows of model.

As before, we can create a counts table from the linelist for each explanatory variable, bind it to models, and make a nice table. We begin with the variables, and iterate through them with map(). We iterate through a user-defined function which involves creating a counts table with **dplyr** functions. Then the results are combined and bound with the models model results.

Below is what the data frame looks like. See the page on [Tables for presentation](#tables-for-presentation) for ideas on how to further convert this table to pretty HTML output (e.g. with **flextable**).

### ****gtsummary**** package

Below we present the use of tbl\_uvregression() from the **gtsummary** package. Just like in the page on [Descriptive tables](https://epirhandbook.com/descriptive-tables.html), **gtsummary** functions do a good job of running statistics and producing professional-looking outputs. This function produces a table of univariate regression results.

We select only the necessary columns from the linelist (explanatory variables and the outcome variable) and pipe them into tbl\_uvregression(). We are going to run univariate regression on each of the columns we defined as explanatory\_vars in the data Preparation section (gender, fever, chills, cough, aches, vomit, and age\_cat).

Within the function itself, we provide the method = as glm (no quotes), the y = outcome column (outcome), specify to method.args = that we want to run logistic regression via family = binomial, and we tell it to exponentiate the results.

The output is HTML and contains the counts

There are many modifications you can make to this table output, such as adjusting the text labels, bolding rows by their p-value, etc. See tutorials [here](http://www.danieldsjoberg.com/gtsummary/articles/tbl_regression.html) and elsewhere online.

## Stratified

Stratified analysis is currently still being worked on for **gtsummary**, this page will be updated in due course.

## Multivariable

For multivariable analysis, we again present two approaches:

* glm() and tidy()
* **gtsummary** package

The workflow is similar for each and only the last step of pulling together a final table is different.

### Conduct multivariable

Here we use glm() but add more variables to the right side of the equation, separated by plus symbols (+).

To run the model with all of our explanatory variables we would run:

If you want to include two variables and an interaction between them you can separate them with an asterisk \* instead of a +. Separate them with a colon : if you are only specifying the interaction. For example:

Optionally, you can use this code to leverage the pre-defined vector of column names and re-create the above command using str\_c(). This might be useful if your explanatory variable names are changing, or you don’t want to type them all out again.

#### Building the model

You can build your model step-by-step, saving various models that include certain explanatory variables. You can compare these models with likelihood-ratio tests using lrtest() from the package **lmtest**, as below:

**NOTE:** Using **base** anova(model1, model2, test = "Chisq) produces the same results

Another option is to take the model object and apply the step() function from the **stats** package. Specify which variable selection direction you want use when building the model.

You can also turn off scientific notation in your R session, for clarity:

As described in the section on univariate analysis, pass the model output to tidy() to exponentiate the log odds and CIs. Finally we round all numeric columns to two decimal places. Scroll through to see all the rows.

Here is what the resulting data frame looks like:

### Combine univariate and multivariable

#### Combine with ****gtsummary****

The **gtsummary** package provides the tbl\_regression() function, which will take the outputs from a regression (glm() in this case) and produce an nice summary table.

Let’s see the table:

You can also combine several different output tables produced by **gtsummary** with the tbl\_merge() function. We now combine the multivariable results with the **gtsummary** univariate results that we created [above](#reg_gt_uni):

#### Combine with ****dplyr****

An alternative way of combining the glm()/tidy() univariate and multivariable outputs is with the **dplyr** join functions.

* Join the univariate results from earlier (univ\_tab\_base, which contains counts) with the tidied multivariable results mv\_tab\_base
* Use select() to keep only the columns we want, specify their order, and re-name them
* Use round() with two decimal places on all the column that are class Double

## Forest plot

This section shows how to produce a plot with the outputs of your regression. There are two options, you can build a plot yourself using **ggplot2** or use a meta-package called **easystats** (a package that includes many packages).

See the page on [ggplot basics](#ggplot-basics) if you are unfamiliar with the **ggplot2** plotting package.

### ****ggplot2**** package

You can build a forest plot with ggplot() by plotting elements of the multivariable regression results. Add the layers of the plots using these “geoms”:

* estimates with geom\_point()
* confidence intervals with geom\_errorbar()
* a vertical line at OR = 1 with geom\_vline()

Before plotting, you may want to use fct\_relevel() from the **forcats** package to set the order of the variables/levels on the y-axis. ggplot() may display them in alpha-numeric order which would not work well for these age category values (“30” would appear before “5”). See the page on [Factors](#factors) for more details.

### ****easystats**** packages

An alternative, if you do not want to the fine level of control that **ggplot2** provides, is to use a combination of **easystats** packages.

The function model\_parameters() from the **parameters** package does the equivalent of the **broom** package function tidy(). The **see** package then accepts those outputs and creates a default forest plot as a ggplot() object.

## Resources

The content of this page was informed by these resources and vignettes online:

[Linear regression in R](https://www.datacamp.com/community/tutorials/linear-regression-R)

[gtsummary](http://www.danieldsjoberg.com/gtsummary/articles/tbl_regression.html)

[UCLA stats page](https://stats.idre.ucla.edu/other/dae/)

[sthda stepwise regression](http://www.sthda.com/english/articles/36-classification-methods-essentials/150-stepwise-logistic-regression-essentials-in-r/)

# Missing data

This page will cover how to:

1. Assess missingness
2. Filter out rows by missingness
3. Plot missingness over time
4. Handle how NA is displayed in plots
5. Perform missing value imputation: MCAR, MAR, MNAR

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

### Convert missing on import

When importing your data, be aware of values that should be classified as missing. For example, 99, 999, “Missing”, blank cells ("“), or cells with an empty space (” "). You can convert these to NA (R’s version of missing data) during the data import command.  
See the page on importing page section on [Missing data](#import_missing) for details, as the exact syntax varies by file type.

## Missing values in R

Below we explore ways that missingness is presented and assessed in R, along with some adjacent values and functions.

### NA

In R, missing values are represented by a reserved (special) value - NA. Note that this is typed without quotes. “NA” is different and is just a normal character value (also a Beatles lyric from the song Hey Jude).

Your data may have other ways of representing missingness, such as “99”, or “Missing”, or “Unknown” - you may even have empty character value "" which looks “blank”, or a single space " ". Be aware of these and consider whether to [convert them to NA during import](#import_missing) or during data cleaning with na\_if().

In your data cleaning, you may also want to convert the other way - changing all NA to “Missing” or similar with replace\_na() or with fct\_explicit\_na() for factors.

### Versions of NA

Most of the time, NA represents a missing value and everything works fine. However, in some circumstances you may encounter the need for variations of NA specific to an object class (character, numeric, etc). This will be rare, but you should be aware.  
The typical scenario for this is when creating a new column with the **dplyr** function case\_when(). As described in the [Cleaning data and core functions](#clean_case_when) page, this function evaluates every row in the data frame, assess whether the rows meets specified logical criteria (right side of the code), and assigns the correct new value (left side of the code). Importantly: all values on the right side must be the same class.

If you want NA on the right side, you may need to specify one of the special NA options listed below. If the other right side values are character, consider using “Missing” instead or otherwise use NA\_character\_. If they are all numeric, use NA\_real\_. If they are all dates or logical, you can use NA.

* NA - use for dates or logical TRUE/FALSE
* NA\_character\_ - use for characters
* NA\_real\_ - use for numeric

Again, it is not likely you will encounter these variations unless you are using case\_when() to create a new column. See the [R documentation on NA](https://stat.ethz.ch/R-manual/R-devel/library/base/html/NA.html) for more information.

### NULL

NULL is another reserved value in R. It is the logical representation of a statement that is neither true nor false. It is returned by expressions or functions whose values are undefined. Generally do not assign NULL as a value, unless writing functions or perhaps writing a [**shiny** app](#dashboards-with-shiny) to return NULL in specific scenarios.

Null-ness can be assessed using is.null() and conversion can made with as.null().

See this [blog post](https://www.r-bloggers.com/2010/04/r-na-vs-null/) on the difference between NULL and NA.

### NaN

Impossible values are represented by the special value NaN. An example of this is when you force R to divide 0 by 0. You can assess this with is.nan(). You may also encounter complementary functions including is.infinite() and is.finite().

### Inf

Inf represents an infinite value, such as when you divide a number by 0.

As an example of how this might impact your work: let’s say you have a vector/column z that contains these values: z <- c(1, 22, NA, Inf, NaN, 5)

If you want to use max() on the column to find the highest value, you can use the na.rm = TRUE to remove the NA from the calculation, but the Inf and NaN remain and Inf will be returned. To resolve this, you can use brackets [ ] and is.finite() to subset such that only finite values are used for the calculation: max(z[is.finite(z)]).

### Examples

| **R command** | **Outcome** |
| --- | --- |
| 5 / 0 | Inf |
| 0 / 0 | NaN |
| 5 / NA | NA |
| 5 / Inf |0NA - 5|NAInf / 5|Infclass(NA)| "logical"class(NaN)| "numeric"class(Inf)| "numeric"class(NULL)` | “NULL” |

“NAs introduced by coercion” is a common warning message. This can happen if you attempt to make an illegal conversion like inserting a character value into a vector that is otherwise numeric.

NULL is ignored in a vector.

Variance of one number results in NA.

## Useful functions

The following are useful **base** R functions when assessing or handling missing values:

### is.na() and !is.na()

Use is.na()to identify missing values, or use its opposite (with ! in front) to identify non-missing values. These both return a logical value (TRUE or FALSE). Remember that you can sum() the resulting vector to count the number TRUE, e.g. sum(is.na(linelist$date\_outcome)).

### na.omit()

This function, if applied to a data frame, will remove rows with any missing values. It is also from **base** R.  
If applied to a vector, it will remove NA values from the vector it is applied to. For example:

### drop\_na()

This is a **tidyr** function that is useful in a [data cleaning pipeline](#cleaning-data-and-core-functions). If run with the parentheses empty, it removes rows with any missing values. If column names are specified in the parentheses, rows with missing values in those columns will be dropped. You can also use “tidyselect” syntax to specify the columns.

### na.rm = TRUE

When you run a mathematical function such as max(), min(), sum() or mean(), if there are any NA values present the returned value will be NA. This default behavior is intentional, so that you are alerted if any of your data are missing.

You can avoid this by removing missing values from the calculation. To do this, include the argument na.rm = TRUE (“na.rm” stands for “remove NA”).

## Assess missingness in a data frame

You can use the package **naniar** to assess and visualize missingness in the data frame linelist.

### Quantifying missingness

To find the percent of all values that are missing use pct\_miss(). Use n\_miss() to get the number of missing values.

The two functions below return the percent of rows with any missing value, or that are entirely complete, respectively. Remember that NA means missing, and that `"" or " " will not be counted as missing.

### Visualizing missingness

The gg\_miss\_var() function will show you the number (or %) of missing values in each column. A few nuances:

* You can add a column name (not in quote) to the argument facet = to see the plot by groups
* By default, counts are shown instead of percents, change this with show\_pct = TRUE
* You can add axis and title labels as for a normal ggplot() with + labs(...)

Here the data are piped %>% into the function. The facet = argument is also used to split the data.

You can use vis\_miss() to visualize the data frame as a heatmap, showing whether each value is missing or not. You can also select() certain columns from the data frame and provide only those columns to the function.

### Explore and visualize missingness relationships

How do you visualize something that is not there??? By default, ggplot() removes points with missing values from plots.

**naniar** offers a solution via geom\_miss\_point(). When creating a scatterplot of two columns, records with one of the values missing and the other value present are shown by setting the missing values to 10% lower than the lowest value in the column, and coloring them distinctly.

In the scatterplot below, the red dots are records where the value for one column is present but the value for the other column is missing. This allows you to see the distribution of missing values in relation to the non-missing values.

To assess missingness in the data frame stratified by another column, consider gg\_miss\_fct(), which returns a heatmap of percent missingness in the data frame by a factor/categorical (or date) column:

This function can also be used with a date column to see how missingness has changed over time:

### “Shadow” columns

Another way to visualize missingness in one column by values in a second column is using the “shadow” that **naniar** can create. bind\_shadow() creates a binary NA/not NA column for every existing column, and binds all these new columns to the original dataset with the appendix "\_NA". This doubles the number of columns - see below:

These “shadow” columns can be used to plot the proportion of values that are missing, by any another column.

For example, the plot below shows the proportion of records missing days\_onset\_hosp (number of days from symptom onset to hospitalisation), by that record’s value in date\_hospitalisation. Essentially, you are plotting the density of the x-axis column, but stratifying the results (color =) by a shadow column of interest. This analysis works best if the x-axis is a numeric or date column.

You can also use these “shadow” columns to stratify a statistical summary, as shown below:

An alternative way to plot the proportion of a column’s values that are missing over time is shown below. It does not involve **naniar**. This example shows percent of weekly observations that are missing).

1. Aggregate the data into a useful time unit (days, weeks, etc.), summarizing the proportion of observations with NA (and any other values of interest)
2. Plot the proportion missing as a line using ggplot()

Below, we take the linelist, add a new column for week, group the data by week, and then calculate the percent of that week’s records where the value is missing. (note: if you want % of 7 days the calculation would be slightly different).

Then we plot the proportion missing as a line, by week. The [ggplot basics](#ggplot-basics) page if you are unfamiliar with the **ggplot2** plotting package.

## Using data with missing values

### Filter out rows with missing values

To quickly remove rows with missing values, use the **dplyr** function drop\_na().

The original linelist has nrow(linelist) rows. The adjusted number of rows is shown below:

You can specify to drop rows with missingness in certain columns:

You can list columns one after the other, or use [“tidyselect” helper functions](#clean_tidyselect):

### Handling NA in ggplot()

It is often wise to report the number of values excluded from a plot in a caption. Below is an example:

In ggplot(), you can add labs() and within it a caption =. In the caption, you can use str\_glue() from **stringr** package to paste values together into a sentence dynamically so they will adjust to the data. An example is below:

* Note the use of \n for a new line.
* Note that if multiple column would contribute to values not being plotted (e.g. age or sex if those are reflected in the plot), then you must filter on those columns as well to correctly calculate the number not shown.

Sometimes, it can be easier to save the string as an object in commands prior to the ggplot() command, and simply reference the named string object within the str\_glue().

### NA in factors

If your column of interest is a factor, use fct\_explicit\_na() from the **forcats** package to convert NA values to a character value. See more detail in the [Factors](#factors) page. By default, the new value is “(Missing)” but this can be adjusted via the na\_level = argument.

## Imputation

Sometimes, when analyzing your data, it will be important to “fill in the gaps” and impute missing data While you can always simply analyze a dataset after removing all missing values, this can cause problems in many ways. Here are two examples:

1. By removing all observations with missing values or variables with a large amount of missing data, you might reduce your power or ability to do some types of analysis. For example, as we discovered earlier, only a small fraction of the observations in our linelist dataset have no missing data across all of our variables. If we removed the majority of our dataset we’d be losing a lot of information! And, most of our variables have some amount of missing data–for most analysis it’s probably not reasonable to drop every variable that has a lot of missing data either.
2. Depending on why your data is missing, analysis of only non-missing data might lead to biased or misleading results. For example, as we learned earlier we are missing data for some patients about whether they’ve had some important symptoms like fever or cough. But, as one possibility, maybe that information wasn’t recorded for people that just obviously weren’t very sick. In that case, if we just removed these observations we’d be excluding some of the healthiest people in our dataset and that might really bias any results.

It’s important to think about why your data might be missing in addition to seeing how much is missing. Doing this can help you decide how important it might be to impute missing data, and also which method of imputing missing data might be best in your situation.

### Types of missing data

Here are three general types of missing data:

1. **Missing Completely at Random** (MCAR). This means that there is no relationship between the probability of data being missing and any of the other variables in your data. The probability of being missing is the same for all cases This is a rare situation. But, if you have strong reason to believe your data is MCAR analyzing only non-missing data without imputing won’t bias your results (although you may lose some power). [TODO: consider discussing statistical tests for MCAR]
2. **Missing at Random** (MAR). This name is actually a bit misleading as MAR means that your data is missing in a systematic, predictable way based on the other information you have. For example, maybe every observation in our dataset with a missing value for fever was actually not recorded because every patient with chills and and aches was just assumed to have a fever so their temperature was never taken. If true, we could easily predict that every missing observation with chills and aches has a fever as well and use this information to impute our missing data. In practice, this is more of a spectrum. Maybe if a patient had both chills and aches they were more likely to have a fever as well if they didn’t have their temperature taken, but not always. This is still predictable even if it isn’t perfectly predictable. This is a common type of missing data
3. **Missing not at Random** (MNAR). Sometimes, this is also called **Not Missing at Random** (NMAR). This assumes that the probability of a value being missing is NOT systematic or predictable using the other information we have but also isn’t missing randomly. In this situation data is missing for unknown reasons or for reasons you don’t have any information about. For example, in our dataset maybe information on age is missing because some very elderly patients either don’t know or refuse to say how old they are. In this situation, missing data on age is related to the value itself (and thus isn’t random) and isn’t predictable based on the other information we have. MNAR is complex and often the best way of dealing with this is to try to collect more data or information about why the data is missing rather than attempt to impute it.

In general, imputing MCAR data is often fairly simple, while MNAR is very challenging if not impossible. Many of the common data imputation methods assume MAR.

### Useful packages

Some useful packages for imputing missing data are Mmisc, missForest (which uses random forests to impute missing data), and mice (Multivariate Imputation by Chained Equations). For this section we’ll just use the mice package, which implements a variety of techniques. The maintainer of the mice package has published an online book about imputing missing data that goes into more detail here (<https://stefvanbuuren.name/fimd/>).

Here is the code to load the mice package:

### Mean Imputation

Sometimes if you are doing a simple analysis or you have strong reason to think you can assume MCAR, you can simply set missing numerical values to the mean of that variable. Perhaps we can assume that missing temperature measurements in our dataset were either MCAR or were just normal values. Here is the code to create a new variable that replaces missing temperature values with the mean temperature value in our dataset. However, in many situations replacing data with the mean can lead to bias, so be careful.

You could also do a similar process for replacing categorical data with a specific value. For our dataset, imagine you knew that all observations with a missing value for their outcome (which can be “Death” or “Recover”) were actually people that died (note: this is not actually true for this dataset):

### Regression imputation

A somewhat more advanced method is to use some sort of statistical model to predict what a missing value is likely to be and replace it with the predicted value. Here is an example of creating predicted values for all the observations where temperature is missing, but age and fever are not, using simple linear regression using fever status and age in years as predictors. In practice you’d want to use a better model than this sort of simple approach.

Or, using the same modeling approach through the mice package to create imputed values for the missing temperature observations:

This is the same type of approach by some more advanced methods like using the missForest package to replace missing data with predicted values. In that case, the prediction model is a random forest instead of a linear regression. You can use other types of models to do this as well. However, while this approach works well under MCAR you should be a bit careful if you believe MAR or MNAR more accurately describes your situation. The quality of your imputation will depend on how good your prediction model is and even with a very good model the variability of your imputed data may be underestimated.

### LOCF and BOCF

Last observation carried forward (LOCF) and baseline observation carried forward (BOCF) are imputation methods for time series/longitudinal data. The idea is to take the previous observed value as a replacement for the missing data. When multiple values are missing in succession, the method searches for the last observed value.

The fill() function from the **tidyr** package can be used for both LOCF and BOCF imputation (however, other packages such as **HMISC**, **zoo**, and **data.table** also include methods for doing this). To show the fill() syntax we’ll make up a simple time series dataset containing the number of cases of a disease for each quarter of the years 2000 and 2001. However, the year value for subsequent quarters after Q1 are missing so we’ll need to impute them. The fill() junction is also demonstrated in the [Pivoting data](#pivoting-data) page.

Note: make sure your data are sorted correctly before using the fill() function. fill() defaults to filling “down” but you can also impute values in different directions by changing the .direction parameter. We can make a similar dataset where the year value is recorded only at the end of the year and missing for earlier quarters:

In this example, LOCF and BOCF are clearly the right things to do, but in more complicated situations it may be harder to decide if these methods are appropriate. For example, you may have missing laboratory values for a hospital patient after the first day. Sometimes, this can mean the lab values didn’t change…but it could also mean the patient recovered and their values would be very different after the first day! Use these methods with caution.

### Multiple Imputation

The online book we mentioned earlier by the author of the mice package (<https://stefvanbuuren.name/fimd/>) contains a detailed explanation of multiple imputation and why you’d want to use it. But, here is a basic explanation of the method:

When you do multiple imputation, you create multiple datasets with the missing values imputed to plausible data values (depending on your research data you might want to create more or less of these imputed datasets, but the mice package sets the default number to 5). The difference is that rather than a single, specific value each imputed value is drawn from an estimated distribution (so it includes some randomness). As a result, each of these datasets will have slightly different different imputed values (however, the non-missing data will be the same in each of these imputed datasets). You still use some sort of predictive model to do the imputation in each of these new datasets (mice has many options for prediction methods including Predictive Mean Matching, logistic regression, and random forest) but the mice package can take care of many of the modeling details.

Then, once you have created these new imputed datasets, you can apply then apply whatever statistical model or analysis you were planning to do for each of these new imputed datasets and pool the results of these models together. This works very well to reduce bias in both MCAR and many MAR settings and often results in more accurate standard error estimates.

Here is an example of applying the Multiple Imputation process to predict temperature in our linelist dataset using a age and fever status (our simplified model\_dataset from above):

Here we used the mice default method of imputation, which is Predictive Mean Matching. We then used these imputed datasets to separately estimate and then pool results from simple linear regressions on each of these datasets. There are many details we’ve glossed over and many settings you can adjust during the Multiple Imputation process while using the mice package. For example, you won’t always have numerical data and might need to use other imputation methods (you can still use the mice package for many other types of data and methods). But, for a more robust analysis when missing data is a significant concern, Multiple Imputation is good solution that isn’t always much more work than doing a complete case analysis.

## Resources

Vignette on the [naniar package](https://cran.r-project.org/web/packages/naniar/vignettes/getting-started-w-naniar.html)

Gallery of [missing value visualizations](https://cran.r-project.org/web/packages/naniar/vignettes/naniar-visualisation.html)

[Online book](https://stefvanbuuren.name/fimd/) about multiple imputation in R by the maintainer of the **mice** package

# Standardised rates

This page will show you two ways to standardize an outcome, such as hospitalizations or mortality, by characteristics such as age and sex.

* Using **dsr** package
* Using **PHEindicatormethods** package

We begin by extensively demonstrating the processes of data preparation/cleaning/joining, as this is common when combining population data from multiple countries, standard population data, deaths, etc.

## Overview

There are two main ways to standardize: direct and indirect standardization. Let’s say we would like to the standardize mortality rate by age and sex for country A and country B, and compare the standardized rates between these countries.

* For direct standardization, you will have to know the number of the at-risk population and the number of deaths for each stratum of age and sex, for country A and country B. One stratum in our example could be females between ages 15-44.
* For indirect standardization, you only need to know the total number of deaths and the age- and sex structure of each country. This option is therefore feasible if age- and sex-specific mortality rates or population numbers are not available. Indirect standardization is furthermore preferable in case of small numbers per stratum, as estimates in direct standardization would be influenced by substantial sampling variation.

## Preparation

To show how standardization is done, we will use fictitious population counts and death counts from country A and country B, by age (in 5 year categories) and sex (female, male). To make the datasets ready for use, we will perform the following preparation steps:

1. Load packages
2. Load datasets
3. Join the population and death data from the two countries
4. Pivot longer so there is one row per age-sex stratum
5. Clean the reference population (world standard population) and join it to the country data

In your scenario, your data may come in a different format. Perhaps your data are by province, city, or other catchment area. You may have one row for each death and information on age and sex for each (or a significant proportion) of these deaths. In this case, see the pages on [Grouping data](#grouping-data), [Pivoting data](#pivoting-data), and [Descriptive tables](#descriptive-tables) to create a dataset with event and population counts per age-sex stratum.

We also need a reference population, the standard population. For the purposes of this exercise we will use the world\_standard\_population\_by\_sex. The World standard population is based on the populations of 46 countries and was developed in 1960. There are many “standard” populations - as one example, the website of [NHS Scotland](https://www.opendata.nhs.scot/dataset/standard-populations) is quite informative on the European Standard Population, World Standard Population and Scotland Standard Population.

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

**CAUTION:** If you have a newer version of R, the **dsr** package cannot be directly downloaded from CRAN. However, it is still available from the CRAN archive. You can install and use this one.

For non-Mac users:

For Mac users:

### Load population data

See the [Download handbook and data](#download-handbook-and-data) page for instructions on how to download all the example data in the handbook. You can import the Standardisation page data directly into R from our Github repository by running the following import() commands:

First we load the demographic data (counts of males and females by 5-year age category) for the two countries that we will be comparing, “Country A” and “Country B”.

### Load death counts

Conveniently, we also have the counts of deaths during the time period of interest, by age and sex. Each country’s counts are in a separate file, shown below.

Deaths in Country A

Deaths in Country B

### Clean populations and deaths

We need to join and transform these data in the following ways:

* Combine country populations into one dataset and pivot “long” so that each age-sex stratum is one row
* Combine country death counts into one dataset and pivot “long” so each age-sex stratum is one row
* Join the deaths to the populations

First, we combine the country populations datasets, pivot longer, and do minor cleaning. See the page on [Pivoting data](#pivoting-data) for more detail.

The combined population data now look like this (click through to see countries A and B):

And now we perform similar operations on the two deaths datasets.

The deaths data now look like this, and contain data from both countries:

We now join the deaths and population data based on common columns Country, age\_cat5, and Sex. This adds the column Deaths.

We can now classify Sex, age\_cat5, and Country as factors and set the level order using fct\_relevel() function from the **forcats** package, as described in the page on [Factors](#factors). Note, classifying the factor levels doesn’t visibly change the data, but the arrange() command does sort it by Country, age category, and sex.

**CAUTION:** If you have few deaths per stratum, consider using 10-, or 15-year categories, instead of 5-year categories for age.

### Load reference population

Lastly, for the direct standardisation, we import the reference population (world “standard population” by sex)

### Clean reference population

The age category values in the country\_data and standard\_pop\_data data frames will need to be aligned.

Currently, the values of the column age\_cat5 from the standard\_pop\_data data frame contain the word “years” and “plus”, while those of the country\_data data frame do not. We will have to make the age category values match. We use str\_replace\_all() from the **stringr** package, as described in the page on [Characters and strings](#characters-and-strings), to replace these patterns with no space "".

Furthermore, the package **dsr** expects that in the standard population, the column containing counts will be called "pop". So we rename that column accordingly.

**CAUTION:** If you try to use str\_replace\_all() to remove a plus symbol, it won’t work because it is a special symbol. “Escape” the specialnes by putting two back slashes in front, as in str\_replace\_call(column, "\\+", "").

### Create dataset with standard population

Finally, the package **PHEindicatormethods**, detailed [below](#standard_phe), expects the standard populations joined to the country event and population counts. So, we will create a dataset all\_data for that purpose.

This complete dataset looks like this:

## ****dsr**** package

Below we demonstrate calculating and comparing directly standardized rates using the **dsr** package. The **dsr** package allows you to calculate and compare directly standardized rates (no indirectly standardized rates!).

In the data Preparation section, we made separate datasets for country counts and standard population:

1. the country\_data object, which is a population table with the number of population and number of deaths per stratum per country
2. the standard\_pop\_clean object, containing the number of population per stratum for our reference population, the World Standard Population

We will use these separate datasets for the **dsr** approach.

### Standardized rates

Below, we calculate rates per country directly standardized for age and sex. We use the dsr() function.

Of note - dsr() expects one data frame for the country populations and event counts (deaths), and a ***separate*** data frame with the reference population. It also expects that in this reference population dataset the unit-time column name is “pop” (we assured this in the data Preparation section).

There are many arguments, as annotated in the code below. Notably, event = is set to the column Deaths, and the fu = (“follow-up”) is set to the Population column. We set the subgroups of comparison as the column Country and we standardize based on age\_cat5 and Sex. These last two columns are not assigned a particular named argument. See ?dsr for details.

Above, we see that while country A had a lower crude mortality rate than country B, it has a higher standardized rate after direct age and sex standardization.

### Standardized rate ratios

The standardized mortality rate is 1.22 times higher in country A compared to country B (95% CI 1.17-1.27).

### Standardized rate difference

Country A has 4.24 additional deaths per 100.000 population (95% CI 3.24-5.24) compared to country A.

## ****PHEindicatormethods**** package

Another way of calculating standardized rates is with the **PHEindicatormethods** package. This package allows you to calculate directly as well as indirectly standardized rates. We will show both.

This section will use the all\_data data frame created at the end of the Preparation section. This data frame includes the country populations, death events, and the world standard reference population. You can view it [here](#standard_all).

### Directly standardized rates

Below, we first group the data by Country and then pass it to the function phe\_dsr() to get directly standardized rates per country.

Of note - the reference (standard) population can be provided as a **column within the country-specific data frame** or as a **separate vector**. If provided within the country-specific data frame, you have to set stdpoptype = "field". If provided as a vector, set stdpoptype = "vector". In the latter case, you have to make sure the ordering of rows by strata is similar in both the country-specific data frame and the reference population, as records will be matched by position. In our example below, we provided the reference population as a column within the country-specific data frame.

See the help with ?phr\_dsr or the links in the References section for more information.

### Indirectly standardized rates

For indirect standardization, you need a reference population with the number of deaths and number of population per stratum. In this example, we will be calculating rates for country A using country B as the reference population, as the standard\_pop\_clean reference population does not include number of deaths per stratum.

Below, we first create the reference population from country B. Then, we pass mortality and population data for country A, combine it with the reference population, and pass it to the function phe\_isr(), to get indirectly standardized rates. Of course, you can do it also vice versa.

Of note - in our example below, the reference population is provided as a separate data frame. In this case, we make sure that x =, n =, x\_ref = and n\_ref = vectors are all ordered by the same standardization category (stratum) values as that in our country-specific data frame, as records will be matched by position.

See the help with ?phr\_isr or the links in the References section for more information.

## Resources

If you would like to see another reproducible example using **dsr** please see [this vignette](https://mran.microsoft.com/snapshot/2020-02-12/web/packages/dsr/vignettes/dsr.html)

For another example using **PHEindicatormethods**, please go to [this website](https://mran.microsoft.com/snapshot/2018-10-22/web/packages/PHEindicatormethods/vignettes/IntroductiontoPHEindicatormethods.html)

See the **PHEindicatormethods** [reference pdf file](https://cran.r-project.org/web/packages/PHEindicatormethods/PHEindicatormethods.pdf)

# Moving averages

This page will cover two methods to calculate and visualize moving averages:

1. Calculate with the **slider** package
2. Calculate within a ggplot() command with the **tidyquant** package

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Calculate with ****slider****

**Use this approach to calculate a moving average in a data frame prior to plotting.**

The **slider** package provides several “sliding window” functions to compute rolling averages, cumulative sums, rolling regressions, etc. It treats a data frame as a vector of rows, allowing iteration row-wise over a data frame.

Here are some of the common functions:

* slide\_dbl() - iterates through a numeric (hence "\_dbl") column performing an operation using a sliding window
  + slide\_sum() - rolling sum shortcut function for slide\_dbl()
  + slide\_mean() - rolling average shortcut function for slide\_dbl()
* slide\_index\_dbl() - applies the rolling window on a numeric column using a separate column to index the window progression (useful if rolling by date with some dates absent)
  + slide\_index\_sum() - rolling sum shortcut function with indexing
  + slide\_index\_mean() - rolling mean shortcut function with indexing

The **slider** package has many other functions that are covered in the Resources section of this page. We briefly touch upon the most common.

**Core arguments**

* .x, the first argument by default, is the vector to iterate over and to apply the function to
* .i = for the “index” versions of the **slider** functions - provide a column to “index” the roll on (see section [below](#roll_index))
* .f =, the second argument by default, either:
  + A function, written without parentheses, like mean, or
  + A formula, which will be converted into a function. For example ~ .x - mean(.x) will return the result of the current value minus the mean of the window’s value
* For more details see this [reference material](https://davisvaughan.github.io/slider/reference/slide.html)

**Window size**

Specify the size of the window by using either .before, .after, or both arguments:

* .before = - Provide an integer
* .after = - Provide an integer
* .complete = - Set this to TRUE if you only want calculation performed on complete windows

For example, to achieve a 7-day window including the current value and the six previous, use .before = 6. To achieve a “centered” window provide the same number to both .before = and .after =.

By default, .complete = will be FALSE so if the full window of rows does not exist, the functions will use available rows to perform the calculation. Setting to TRUE restricts so calculations are only performed on complete windows.

**Expanding window**

To achieve cumulative operations, set the .before = argument to Inf. This will conduct the operation on the current value and all coming before.

### Rolling by date

The most likely use-case of a rolling calculation in applied epidemiology is to examine a metric over time. For example, a rolling measurement of case incidence, based on daily case counts.

If you have clean time series data with values for every date, you may be OK to use slide\_dbl(), as demonstrated here in the [Time series and outbreak detection](#timeseries_moving) page.

However, in many applied epidemiology circumstances you may have dates absent from your data, where there are no events recorded. In these cases, it is best to use the “index” versions of the **slider** functions.

### Indexed data

Below, we show an example using slide\_index\_dbl() on the case linelist. Let us say that our objective is to calculate a rolling 7-day incidence - the sum of cases using a rolling 7-day window. If you are looking for an example of rolling average, see the section below on [grouped rolling](#roll_slider_group).

To begin, the dataset daily\_counts is created to reflect the daily case counts from the linelist, as calculated with count() from **dplyr**.

Here is the daily\_counts data frame - there are nrow(daily\_counts) rows, each day is represented by one row, but especially early in the epidemic some days are not present (there were no cases admitted on those days).

It is crucial to recognize that a standard rolling function (like slide\_dbl() would use a window of 7 rows, not 7 days. So, if there are any absent dates, some windows will actually extend more than 7 calendar days!

A “smart” rolling window can be achieved with slide\_index\_dbl(). The “index” means that the function uses a separate column as an “index” for the rolling window. The window is not simply based on the rows of the data frame.

If the index column is a date, you have the added ability to specify the window extent to .before = and/or .after = in units of **lubridate** days() or months(). If you do these things, the function will include absent days in the windows as if they were there (as NA values).

Let’s show a comparison. Below, we calculate rolling 7-day case incidence with regular and indexed windows.

Observe how in the regular column for the first 7 rows the count steadily increases despite the rows not being within 7 days of each other! The adjacent “indexed” column accounts for these absent calendar days, so its 7-day sums are much lower, at least in this period of the epidemic when the cases a farther between.

Now you can plot these data using ggplot():

### Rolling by group

If you group your data prior to using a **slider** function, the sliding windows will be applied by group. Be careful to arrange your rows in the desired order by group.

Each time a new group begins, the sliding window will re-start. Therefore, one nuance to be aware of is that if your data are grouped and you have set .complete = TRUE, you will have empty values at each transition between groups. As the function moved downward through the rows, every transition in the grouping column will re-start the accrual of the minimum window size to allow a calculation.

See handbook page on [Grouping data](#grouping-data) for details on grouping data.

Below, we count linelist cases by date and by hospital. Then we arrange the rows in ascending order, first ordering by hospital and then within that by date. Next we set group\_by(). Then we can create our new rolling average.

Here is the new dataset:

We can now plot the moving averages, displaying the data by group by specifying ~ hospital to facet\_wrap() in ggplot(). For fun, we plot two geometries - a geom\_col() showing the daily case counts and a geom\_line() showing the 7-day moving average.

**DANGER:** If you get an error saying “slide() was deprecated in tsibble 0.9.0 and is now defunct. Please use slider::slide() instead.”, it means that the slide() function from the **tsibble** package is masking the slide() function from **slider** package. Fix this by specifying the package in the command, such as slider::slide\_dbl().

## Calculate with ****tidyquant**** within ggplot()

The package **tidyquant** offers another approach to calculating moving averages - this time from within a ggplot() command itself.

Below the linelist data are counted by date of onset, and this is plotted as a faded line (alpha < 1). Overlaid on top is a line created with geom\_ma() from the package **tidyquant**, with a set window of 7 days (n = 7) with specified color and thickness.

By default geom\_ma() uses a simple moving average (ma\_fun = "SMA"), but other types can be specified, such as:

* “EMA” - exponential moving average (more weight to recent observations)
* “WMA” - weighted moving average (wts are used to weight observations in the moving average)
* Others can be found in the function documentation

See this [vignette](https://cran.r-project.org/web/packages/tidyquant/vignettes/TQ04-charting-with-tidyquant.html) for more details on the options available within **tidyquant**.

## Resources

See the helpful online [vignette for the **slider** package](https://cran.r-project.org/web/packages/slider/vignettes/slider.html)

The **slider** [github page](https://github.com/DavisVaughan/slider)

A **slider** [vignette](https://davisvaughan.github.io/slider/articles/slider.html)

[tidyquant vignette](https://cran.r-project.org/web/packages/tidyquant/vignettes/TQ04-charting-with-tidyquant.html)

If your use case requires that you “skip over” weekends and even holidays, you might like **almanac** package.

# Time series and outbreak detection

## Overview

This tab demonstrates the use of several packages for time series analysis. It primarily relies on packages from the [**tidyverts**](https://tidyverts.org/) family, but will also use the RECON [**trending**](https://github.com/reconhub/trending) package to fit models that are more appropriate for infectious disease epidemiology.

Note in the below example we use a dataset from the **surveillance** package on Campylobacter in Germany (see the [data chapter](https://epirhandbook.com/download-handbook-and-data.html), of the handbook for details). However, if you wanted to run the same code on a dataset with multiple countries or other strata, then there is an example code template for this in the [r4epis github repo](https://github.com/R4EPI/epitsa).

Topics covered include:

1. Time series data
2. Descriptive analysis
3. Fitting regressions
4. Relation of two time series
5. Outbreak detection
6. Interrupted time series

## Preparation

### Packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load packages with library() from **base** R. See the page on [R basics](https://epirhandbook.com/r-basics.html) for more information on R packages.

### Load data

You can download all the data used in this handbook via the instructions in the [Download handbook and data](#download-handbook-and-data) page.

The example dataset used in this section is weekly counts of campylobacter cases reported in Germany between 2001 and 2011. [You can click here to download this data file (.xlsx).](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/time_series/campylobacter_germany.xlsx)

This dataset is a reduced version of the dataset available in the [**surveillance**](https://cran.r-project.org/web/packages/surveillance/) package. (for details load the surveillance package and see ?campyDE)

Import these data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 10 rows of the counts are displayed below.

### Clean data

The code below makes sure that the date column is in the appropriate format. For this tab we will be using the **tsibble** package and so the yearweek function will be used to create a calendar week variable. There are several other ways of doing this (see the [Working with dates](https://epirhandbook.com/working-with-dates.html) page for details), however for time series its best to keep within one framework (**tsibble**).

### Download climate data

In the relation of two time series section of this page, we will be comparing campylobacter case counts to climate data.

Climate data for anywhere in the world can be downloaded from the EU’s Copernicus Satellite. These are not exact measurements, but based on a model (similar to interpolation), however the benefit is global hourly coverage as well as forecasts.

You can download each of these climate data files from the [Download handbook and data](#download-handbook-and-data) page.

For purposes of demonstration here, we will show R code to use the **ecmwfr** package to pull these data from the Copernicus climate data store. You will need to create a free account in order for this to work. The package website has a useful [walkthrough](https://github.com/bluegreen-labs/ecmwfr#use-copernicus-climate-data-store-cds) of how to do this. Below is example code of how to go about doing this, once you have the appropriate API keys. You have to replace the X’s below with your account IDs. You will need to download one year of data at a time otherwise the server times-out.

If you are not sure of the coordinates for a location you want to download data for, you can use the **tmaptools** package to pull the coordinates off open street maps. An alternative option is the [**photon**](https://github.com/rCarto/photon) package, however this has not been released on to CRAN yet; the nice thing about **photon** is that it provides more contextual data for when there are several matches for your search.

### Load climate data

Whether you downloaded the climate data via our handbook, or used the code above, you now should have 10 years of “.nc” climate data files stored in the same folder on your computer.

Use the code below to import these files into R with the **stars** package.

Once these files have been imported as the object data, we will convert them to a data frame.

## Time series data

There are a number of different packages for structuring and handling time series data. As said, we will focus on the **tidyverts** family of packages and so will use the **tsibble** package to define our time series object. Having a data set defined as a time series object means it is much easier to structure our analysis.

To do this we use the tsibble() function and specify the “index”, i.e. the variable specifying the time unit of interest. In our case this is the epiweek variable.

If we had a data set with weekly counts by province, for example, we would also be able to specify the grouping variable using the key = argument. This would allow us to do analysis for each group.

Looking at class(counts) tells you that on top of being a tidy data frame (“tbl\_df”, “tbl”, “data.frame”), it has the additional properties of a time series data frame (“tbl\_ts”).

You can take a quick look at your data by using **ggplot2**. We see from the plot that there is a clear seasonal pattern, and that there are no missings. However, there seems to be an issue with reporting at the beginning of each year; cases drop in the last week of the year and then increase for the first week of the next year.

**DANGER:** Most datasets aren’t as clean as this example. You will need to check for duplicates and missings as below.

### Duplicates

**tsibble** does not allow duplicate observations. So each row will need to be unique, or unique within the group (key variable). The package has a few functions that help to identify duplicates. These include are\_duplicated() which gives you a TRUE/FALSE vector of whether the row is a duplicate, and duplicates() which gives you a data frame of the duplicated rows.

See the page on [De-duplication](https://epirhandbook.com/de-duplication.html) for more details on how to select rows you want.

### Missings

We saw from our brief inspection above that there are no missings, but we also saw there seems to be a problem with reporting delay around new year. One way to address this problem could be to set these values to missing and then to impute values. The simplest form of time series imputation is to draw a straight line between the last non-missing and the next non-missing value. To do this we will use the **imputeTS** package function na\_interpolation().

See the [Missing data](https://epirhandbook.com/missing-data.html) page for other options for imputation.

Another alternative would be to calculate a moving average, to try and smooth over these apparent reporting issues (see next section, and the page on [Moving averages](https://epirhandbook.com/moving-averages.html)).

## Descriptive analysis

### Moving averages

If data is very noisy (counts jumping up and down) then it can be helpful to calculate a moving average. In the example below, for each week we calculate the average number of cases from the four previous weeks. This smooths the data, to make it more interpretable. In our case this does not really add much, so we will stick to the interpolated data for further analysis. See the [Moving averages](https://epirhandbook.com/moving-averages.html) page for more detail.

### Periodicity

Below we define a custom function to create a periodogram. See the [Writing functions](#writing-functions-1) page for information about how to write functions in R.

First, the function is defined. Its arguments include a dataset with a column counts, start\_week = which is the first week of the dataset, a number to indicate how many periods per year (e.g. 52, 12), and lastly the output style (see details in the code below).

**NOTE:** It is possible to use the above weeks to add them to sin and cosine terms, however we will use a function to generate these terms (see regression section below)

### Decomposition

Classical decomposition is used to break a time series down several parts, which when taken together make up for the pattern you see. These different parts are:

* The trend-cycle (the long-term direction of the data)
* The seasonality (repeating patterns)
* The random (what is left after removing trend and season)

### Autocorrelation

Autocorrelation tells you about the relation between the counts of each week and the weeks before it (called lags).

Using the ACF() function, we can produce a plot which shows us a number of lines for the relation at different lags. Where the lag is 0 (x = 0), this line would always be 1 as it shows the relation between an observation and itself (not shown here). The first line shown here (x = 1) shows the relation between each observation and the observation before it (lag of 1), the second shows the relation between each observation and the observation before last (lag of 2) and so on until lag of 52 which shows the relation between each observation and the observation from 1 year (52 weeks before).

Using the PACF() function (for partial autocorrelation) shows the same type of relation but adjusted for all other weeks between. This is less informative for determining periodicity.

You can formally test the null hypothesis of independence in a time series (i.e.  that it is not autocorrelated) using the Ljung-Box test (in the **stats** package). A significant p-value suggests that there is autocorrelation in the data.

## Fitting regressions

It is possible to fit a large number of different regressions to a time series, however, here we will demonstrate how to fit a negative binomial regression - as this is often the most appropriate for counts data in infectious diseases.

### Fourier terms

Fourier terms are the equivalent of sin and cosin curves. The difference is that these are fit based on finding the most appropriate combination of curves to explain your data.

If only fitting one fourier term, this would be the equivalent of fitting a sin and a cosin for your most frequently occurring lag seen in your periodogram (in our case 52 weeks). We use the fourier() function from the **forecast** package.

In the below code we assign using the $, as fourier() returns two columns (one for sin one for cosin) and so these are added to the dataset as a list, called “fourier” - but this list can then be used as a normal variable in regression.

### Negative binomial

It is possible to fit regressions using base **stats** or **MASS** functions (e.g. lm(), glm() and glm.nb()). However we will be using those from the **trending** package, as this allows for calculating appropriate confidence and prediction intervals (which are otherwise not available). The syntax is the same, and you specify an outcome variable then a tilde (~) and then add your various exposure variables of interest separated by a plus (+).

The other difference is that we first define the model and then fit() it to the data. This is useful because it allows for comparing multiple different models with the same syntax.

**TIP:** If you wanted to use rates, rather than counts you could include the population variable as a logarithmic offset term, by adding offset(log(population). You would then need to set population to be 1, before using predict() in order to produce a rate.

**TIP:** For fitting more complex models such as ARIMA or prophet, see the [**fable**](https://fable.tidyverts.org/index.html) package.

### Residuals

To see how well our model fits the observed data we need to look at the residuals. The residuals are the difference between the observed counts and the counts estimated from the model. We could calculate this simply by using case\_int - estimate, but the residuals() function extracts this directly from the regression for us.

What we see from the below, is that we are not explaining all of the variation that we could with the model. It might be that we should fit more fourier terms, and address the amplitude. However for this example we will leave it as is. The plots show that our model does worse in the peaks and troughs (when counts are at their highest and lowest) and that it might be more likely to underestimate the observed counts.

## Relation of two time series

Here we look at using weather data (specifically the temperature) to explain campylobacter case counts.

### Merging datasets

We can join our datasets using the week variable. For more on merging see the handbook section on [joining](https://epirhandbook.com/joining-data.html).

### Descriptive analysis

First plot your data to see if there is any obvious relation. The plot below shows that there is a clear relation in the seasonality of the two variables, and that temperature might peak a few weeks before the case number. For more on pivoting data, see the handbook section on [pivoting data](https://epirhandbook.com/pivoting-data.html).

### Lags and cross-correlation

To formally test which weeks are most highly related between cases and temperature. We can use the cross-correlation function (CCF()) from the **feasts** package. You could also visualise (rather than using arrange) using the autoplot() function.

We see from this that a lag of 4 weeks is most highly correlated, so we make a lagged temperature variable to include in our regression.

**DANGER:** Note that the first four weeks of our data in the lagged temperature variable are missing (NA) - as there are not four weeks prior to get data from. In order to use this dataset with the **trending** predict() function, we need to use the the simulate\_pi = FALSE argument within predict() further down. If we did want to use the simulate option, then we have to drop these missings and store as a new data set by adding drop\_na(t2m\_lag4) to the code chunk below.

### Negative binomial with two variables

We fit a negative binomial regression as done previously. This time we add the temperature variable lagged by four weeks.

**CAUTION:** Note the use of simulate\_pi = FALSE within the predict() argument. This is because the default behaviour of **trending** is to use the **ciTools** package to estimate a prediction interval. This does not work if there are NA counts, and also produces more granular intervals. See ?trending::predict.trending\_model\_fit for details.

To investigate the individual terms, we can pull the original negative binomial regression out of the **trending** format using get\_model() and pass this to the **broom** package tidy() function to retrieve exponentiated estimates and associated confidence intervals.

What this shows us is that lagged temperature, after controlling for trend and seasonality, is similar to the case counts (estimate ~ 1) and significantly associated. This suggests that it might be a good variable for use in predicting future case numbers (as climate forecasts are readily available).

A quick visual inspection of the model shows that it might do a better job of estimating the observed case counts.

#### Residuals

We investigate the residuals again to see how well our model fits the observed data. The results and interpretation here are similar to those of the previous regression, so it may be more feasible to stick with the simpler model without temperature.

## Outbreak detection

We will demonstrate two (similar) methods of detecting outbreaks here. The first builds on the sections above. We use the **trending** package to fit regressions to previous years, and then predict what we expect to see in the following year. If observed counts are above what we expect, then it could suggest there is an outbreak. The second method is based on similar principles but uses the **surveillance** package, which has a number of different algorithms for aberration detection.

**CAUTION:** Normally, you are interested in the current year (where you only know counts up to the present week). So in this example we are pretending to be in week 39 of 2011.

### ****trending**** package

For this method we define a baseline (which should usually be about 5 years of data). We fit a regression to the baseline data, and then use that to predict the estimates for the next year.

#### Cut-off date

It is easier to define your dates in one place and then use these throughout the rest of your code.

Here we define a start date (when our observations started) and a cut-off date (the end of our baseline period - and when the period we want to predict for starts). ~We also define how many weeks are in our year of interest (the one we are going to be predicting)~. We also define how many weeks are between our baseline cut-off and the end date that we are interested in predicting for.

**NOTE:** In this example we pretend to currently be at the end of September 2011 (“2011 W39”).

#### Add rows

To be able to forecast in a tidyverse format, we need to have the right number of rows in our dataset, i.e. one row for each week up to the end\_datedefined above. The code below allows you to add these rows for by a grouping variable - for example if we had multiple countries in one dataset, we could group by country and then add rows appropriately for each. The group\_by\_key() function from **tsibble** allows us to do this grouping and then pass the grouped data to **dplyr** functions, group\_modify() and add\_row(). Then we specify the sequence of weeks between one after the maximum week currently available in the data and the end week.

#### Fourier terms

We need to redefine our fourier terms - as we want to fit them to the baseline date only and then predict (extrapolate) those terms for the next year. To do this we need to combine two output lists from the fourier() function together; the first one is for the baseline data, and the second one predicts for the year of interest (by defining the h argument).

N.b. to bind rows we have to use rbind() (rather than tidyverse bind\_rows) as the fourier columns are a list (so not named individually).

#### Split data and fit regression

We now have to split our dataset in to the baseline period and the prediction period. This is done using the **dplyr** group\_split() function after group\_by(), and will create a list with two data frames, one for before your cut-off and one for after.

We then use the **purrr** package pluck() function to pull the datasets out of the list (equivalent of using square brackets, e.g. dat[[1]]), and can then fit our model to the baseline data, and then use the predict() function for our data of interest after the cut-off.

See the page on [Iteration, loops, and lists](#iteration-loops-and-lists) to learn more about **purrr**.

**CAUTION:** Note the use of simulate\_pi = FALSE within the predict() argument. This is because the default behaviour of **trending** is to use the **ciTools** package to estimate a prediction interval. This does not work if there are NA counts, and also produces more granular intervals. See ?trending::predict.trending\_model\_fit for details.

As previously, we can visualise our model with **ggplot**. We highlight alerts with red dots for observed counts above the 95% prediction interval. This time we also add a vertical line to label when the forecast starts.

#### Prediction validation

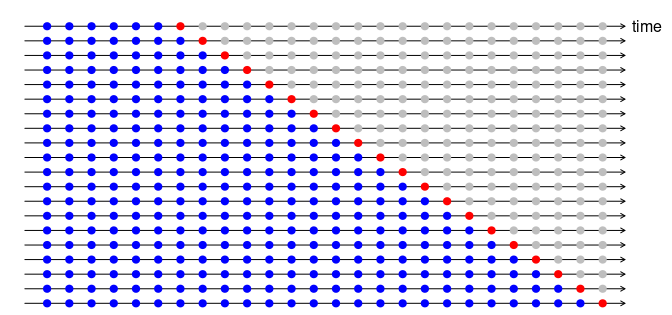
Beyond inspecting residuals, it is important to investigate how good your model is at predicting cases in the future. This gives you an idea of how reliable your threshold alerts are.

The traditional way of validating is to see how well you can predict the latest year before the present one (because you don’t yet know the counts for the “current year”). For example in our data set we would use the data from 2002 to 2009 to predict 2010, and then see how accurate those predictions are. Then refit the model to include 2010 data and use that to predict 2011 counts.

As can be seen in the figure below by Hyndman et al in [“Forecasting principles and practice”](https://otexts.com/fpp3/).

figure reproduced with permission from the authors

The downside of this is that you are not using all the data available to you, and it is not the final model that you are using for prediction.

An alternative is to use a method called cross-validation. In this scenario you roll over all of the data available to fit multiple models to predict one year ahead. You use more and more data in each model, as seen in the figure below from the same [Hyndman et al text]((<https://otexts.com/fpp3/>). For example, the first model uses 2002 to predict 2003, the second uses 2002 and 2003 to predict 2004, and so on. figure reproduced with permission from the authors

In the below we use **purrr** package map() function to loop over each dataset. We then put estimates in one data set and merge with the original case counts, to use the **yardstick** package to compute measures of accuracy. We compute four measures including: Root mean squared error (RMSE), Mean absolute error (MAE), Mean absolute scaled error (MASE), Mean absolute percent error (MAPE).

**CAUTION:** Note the use of simulate\_pi = FALSE within the predict() argument. This is because the default behaviour of **trending** is to use the **ciTools** package to estimate a prediction interval. This does not work if there are NA counts, and also produces more granular intervals. See ?trending::predict.trending\_model\_fit for details.

### ****surveillance**** package

In this section we use the **surveillance** package to create alert thresholds based on outbreak detection algorithms. There are several different methods available in the package, however we will focus on two options here. For details, see these papers on the [application](https://cran.r-project.org/web/packages/surveillance/vignettes/monitoringCounts.pdf) and [theory](https://cran.r-project.org/web/packages/surveillance/vignettes/glrnb.pdf) of the alogirthms used.

The first option uses the improved Farrington method. This fits a negative binomial glm (including trend) and down-weights past outbreaks (outliers) to create a threshold level.

The second option use the glrnb method. This also fits a negative binomial glm but includes trend and fourier terms (so is favoured here). The regression is used to calculate the “control mean” (~fitted values) - it then uses a computed generalized likelihood ratio statistic to assess if there is shift in the mean for each week. Note that the threshold for each week takes in to account previous weeks so if there is a sustained shift an alarm will be triggered. (Also note that after each alarm the algorithm is reset)

In order to work with the **surveillance** package, we first need to define a “surveillance time series” object (using the sts() function) to fit within the framework.

#### Farrington method

We then define each of our parameters for the Farrington method in a list. Then we run the algorithm using farringtonFlexible() and then we can extract the threshold for an alert using farringtonmethod@upperboundto include this in our dataset. It is also possible to extract a TRUE/FALSE for each week if it triggered an alert (was above the threshold) using farringtonmethod@alarm.

We can then visualise the results in ggplot as done previously.

#### GLRNB method

Similarly for the GLRNB method we define each of our parameters for the in a list, then fit the algorithm and extract the upper bounds.

**CAUTION:** This method uses “brute force” (similar to bootstrapping) for calculating thresholds, so can take a long time!

See the [GLRNB vignette](https://cran.r-project.org/web/packages/surveillance/vignettes/glrnb.pdf) for details.

Visualise the outputs as previously.

## Interrupted timeseries

Interrupted timeseries (also called segmented regression or intervention analysis), is often used in assessing the impact of vaccines on the incidence of disease. But it can be used for assessing impact of a wide range of interventions or introductions. For example changes in hospital procedures or the introduction of a new disease strain to a population. In this example we will pretend that a new strain of Campylobacter was introduced to Germany at the end of 2008, and see if that affects the number of cases. We will use negative binomial regression again. The regression this time will be split in to two parts, one before the intervention (or introduction of new strain here) and one after (the pre and post-periods). This allows us to calculate an incidence rate ratio comparing the two time periods. Explaining the equation might make this clearer (if not then just ignore!).

The negative binomial regression can be defined as follows:

\[\log(Y\_t)= β\_0 + β\_1 \times t+ β\_2 \times δ(t-t\_0) + β\_3\times(t-t\_0 )^+ + log(pop\_t) + e\_t\]

Where: \(Y\_t\)is the number of cases observed at time \(t\)  
\(pop\_t\) is the population size in 100,000s at time \(t\) (not used here)  
\(t\_0\) is the last year of the of the pre-period (including transition time if any)  
\(δ(x\) is the indicator function (it is 0 if x≤0 and 1 if x>0)  
\((x)^+\) is the cut off operator (it is x if x>0 and 0 otherwise)  
\(e\_t\) denotes the residual Additional terms trend and season can be added as needed.

\(β\_2 \times δ(t-t\_0) + β\_3\times(t-t\_0 )^+\) is the generalised linear part of the post-period and is zero in the pre-period. This means that the \(β\_2\) and \(β\_3\) estimates are the effects of the intervention.

We need to re-calculate the fourier terms without forecasting here, as we will use all the data available to us (i.e. retrospectively). Additionally we need to calculate the extra terms needed for the regression.

We then use these terms to fit a negative binomial regression, and produce a table with percentage change. What this example shows is that there was no significant change.

**CAUTION:** Note the use of simulate\_pi = FALSE within the predict() argument. This is because the default behaviour of **trending** is to use the **ciTools** package to estimate a prediction interval. This does not work if there are NA counts, and also produces more granular intervals. See ?trending::predict.trending\_model\_fit for details.

As previously we can visualise the outputs of the regression.

## Resources

[forecasting: principles and practice textbook](https://otexts.com/fpp3/)  
[EPIET timeseries analysis case studies](https://github.com/EPIET/TimeSeriesAnalysis)  
[Penn State course](https://online.stat.psu.edu/stat510/lesson/1) [Surveillance package manuscript](https://www.jstatsoft.org/article/view/v070i10)

# Epidemic modeling

## Overview

There exists a growing body of tools for epidemic modelling that lets us conduct fairly complex analyses with minimal effort. This section will provide an overview on how to use these tools to:

* estimate the effective reproduction number Rt and related statistics such as the doubling time
* produce short-term projections of future incidence

It is not intended as an overview of the methodologies and statistical methods underlying these tools, so please refer to the Resources tab for links to some papers covering this. Make sure you have an understanding of the methods before using these tools; this will ensure you can accurately interpret their results.

Below is an example of one of the outputs we’ll be producing in this section.

## Preparation

We will use two different methods and packages for Rt estimation, namely **EpiNow** and **EpiEstim**, as well as the **projections** package for forecasting case incidence.

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

We will use the cleaned case linelist for all analyses in this section. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). See the [Download handbook and data](#download-handbook-and-data) page to download all example data used in this handbook.

## Estimating Rt

### EpiNow2 vs. EpiEstim

The reproduction number R is a measure of the transmissibility of a disease and is defined as the expected number of secondary cases per infected case. In a fully susceptible population, this value represents the basic reproduction number R0. However, as the number of susceptible individuals in a population changes over the course of an outbreak or pandemic, and as various response measures are implemented, the most commonly used measure of transmissibility is the effective reproduction number Rt; this is defined as the expected number of secondary cases per infected case at a given time t.

The **EpiNow2** package provides the most sophisticated framework for estimating Rt. It has two key advantages over the other commonly used package, **EpiEstim**:

* It accounts for delays in reporting and can therefore estimate Rt even when recent data is incomplete.
* It estimates Rt on dates of infection rather than the dates of onset of reporting, which means that the effect of an intervention will be immediately reflected in a change in Rt, rather than with a delay.

However, it also has two key disadvantages:

* It requires knowledge of the generation time distribution (i.e. distribution of delays between infection of a primary and secondary cases), incubation period distribution (i.e. distribution of delays between infection and symptom onset) and any further delay distribution relevant to your data (e.g. if you have dates of reporting, you require the distribution of delays from symptom onset to reporting). While this will allow more accurate estimation of Rt, **EpiEstim** only requires the serial interval distribution (i.e. the distribution of delays between symptom onset of a primary and a secondary case), which may be the only distribution available to you.
* **EpiNow2** is significantly slower than **EpiEstim**, anecdotally by a factor of about 100-1000! For example, estimating Rt for the sample outbreak considered in this section takes about four hours (this was run for a large number of iterations to ensure high accuracy and could probably be reduced if necessary, however the points stands that the algorithm is slow in general). This may be unfeasible if you are regularly updating your Rt estimates.

Which package you choose to use will therefore depend on the data, time and computational resources available to you.

### EpiNow2

#### Estimating delay distributions

The delay distributions required to run **EpiNow2** depend on the data you have. Essentially, you need to be able to describe the delay from the date of infection to the date of the event you want to use to estimate Rt. If you are using dates of onset, this would simply be the incubation period distribution. If you are using dates of reporting, you require the delay from infection to reporting. As this distribution is unlikely to be known directly, **EpiNow2** lets you chain multiple delay distributions together; in this case, the delay from infection to symptom onset (e.g. the incubation period, which is likely known) and from symptom onset to reporting (which you can often estimate from the data).

As we have the dates of onset for all our cases in the example linelist, we will only require the incubation period distribution to link our data (e.g. dates of symptom onset) to the date of infection. We can either estimate this distribution from the data or use values from the literature.

A literature estimate of the incubation period of Ebola (taken from [this paper](https://www.nejm.org/doi/full/10.1056/nejmoa1411100)) with a mean of 9.1, standard deviation of 7.3 and maximum value of 30 would be specified as follows:

Note that **EpiNow2** requires these delay distributions to be provided on a **log** scale, hence the log call around each value (except the max parameter which, confusingly, has to be provided on a natural scale). The mean\_sd and sd\_sd define the standard deviation of the mean and standard deviation estimates. As these are not known in this case, we choose the fairly arbitrary value of 0.1.

In this analysis, we instead estimate the incubation period distribution from the linelist itself using the function bootstrapped\_dist\_fit, which will fit a lognormal distribution to the observed delays between infection and onset in the linelist.

The other distribution we require is the generation time. As we have data on infection times **and** transmission links, we can estimate this distribution from the linelist by calculating the delay between infection times of infector-infectee pairs. To do this, we use the handy get\_pairwise function from the package **epicontacts**, which allows us to calculate pairwise differences of linelist properties between transmission pairs. We first create an epicontacts object (see [Transmission chains](#transmission-chains) page for further details):

We then fit the difference in infection times between transmission pairs, calculated using get\_pairwise, to a gamma distribution:

#### Running ****EpiNow2****

Now we just need to calculate daily incidence from the linelist, which we can do easily with the **dplyr** functions group\_by() and n(). Note that **EpiNow2** requires the column names to be date and confirm.

We can then estimate Rt using the epinow function. Some notes on the inputs:

* We can provide any number of ‘chained’ delay distributions to the delays argument; we would simply insert them alongside the incubation\_period object within the delay\_opts function.
* return\_output ensures the output is returned within R and not just saved to a file.
* verbose specifies that we want a readout of the progress.
* horizon indicates how many days we want to project future incidence for.
* We pass additional options to the stan argument to specify how long we want to run the inference for. Increasing samples and chains will give you a more accurate estimate that better characterises uncertainty, however will take longer to run.

#### Analysing outputs

Once the code has finished running, we can plot a summary very easily as follows. Scroll the image to see the full extent.

We can also look at various summary statistics:

For further analyses and custom plotting, you can access the summarised daily estimates via $estimates$summarised. We will convert this from the default data.table to a tibble for ease of use with **dplyr**.

As an example, let’s make a plot of the doubling time and Rt. We will only look at the first few months of the outbreak when Rt is well above one, to avoid plotting extremely high doublings times.

We use the formula log(2)/growth\_rate to calculate the doubling time from the estimated growth rate.

### EpiEstim

To run **EpiEstim**, we need to provide data on daily incidence and specify the serial interval (i.e. the distribution of delays between symptom onset of primary and secondary cases).

Incidence data can be provided to **EpiEstim** as a vector, a data frame, or an incidence object from the original **incidence** package. You can even distinguish between imports and locally acquired infections; see the documentation at ?estimate\_R for further details.

We will create the input using **incidence2**. See the page on [Epidemic curves](#epidemic-curves) for more examples with the **incidence2** package. Since there have been updates to the **incidence2** package that don’t completely align with estimateR()’s expected input, there are some minor additional steps needed. The incidence object consists of a tibble with dates and their respective case counts. We use complete() from **tidyr** to ensure all dates are included (even those with no cases), and then rename() the columns to align with what is expected by estimate\_R() in a later step.

The package provides several options for specifying the serial interval, the details of which are provided in the documentation at ?estimate\_R. We will cover two of them here.

#### Using serial interval estimates from the literature

Using the option method = "parametric\_si", we can manually specify the mean and standard deviation of the serial interval in a config object created using the function make\_config. We use a mean and standard deviation of 12.0 and 5.2, respectively, defined in [this paper](https://bmcmedicine.biomedcentral.com/articles/10.1186/s12916-014-0196-0):

We can then estimate Rt with the estimate\_R function:

and plot a summary of the outputs:

#### Using serial interval estimates from the data

As we have data on dates of symptom onset and transmission links, we can also estimate the serial interval from the linelist by calculating the delay between onset dates of infector-infectee pairs. As we did in the **EpiNow2** section, we will use the get\_pairwise function from the **epicontacts** package, which allows us to calculate pairwise differences of linelist properties between transmission pairs. We first create an epicontacts object (see [Transmission chains](#transmission-chains) page for further details):

We then fit the difference in onset dates between transmission pairs, calculated using get\_pairwise, to a gamma distribution. We use the handy fit\_disc\_gamma from the **epitrix** package for this fitting procedure, as we require a discretised distribution.

We then pass this information to the config object, run **EpiEstim** again and plot the results:

#### Specifying estimation time windows

These default options will provide a weekly sliding estimate and might act as a warning that you are estimating Rt too early in the outbreak for a precise estimate. You can change this by setting a later start date for the estimation as shown below. Unfortunately, **EpiEstim** only provides a very clunky way of specifying these estimations times, in that you have to provide a vector of **integers** referring to the start and end dates for each time window.

Now we re-run **EpiEstim** and can see that the estimates only start from June:

#### Analysing outputs

The main outputs can be accessed via $R. As an example, we will create a plot of Rt and a measure of “transmission potential” given by the product of Rt and the number of cases reported on that day; this represents the expected number of cases in the next generation of infection.

## Projecting incidence

### EpiNow2

Besides estimating Rt, **EpiNow2** also supports forecasting of Rt and projections of case numbers by integration with the **EpiSoon** package under the hood. All you need to do is specify the horizon argument in your epinow function call, indicating how many days you want to project into the future; see the **EpiNow2** section under the “Estimating Rt” for details on how to get **EpiNow2** up and running. In this section, we will just plot the outputs from that analysis, stored in the epinow\_res object.

### projections

The **projections** package developed by RECON makes it very easy to make short term incidence forecasts, requiring only knowledge of the effective reproduction number Rt and the serial interval. Here we will cover how to use serial interval estimates from the literature and how to use our own estimates from the linelist.

#### Using serial interval estimates from the literature

**projections** requires a discretised serial interval distribution of the class distcrete from the package **distcrete**. We will use a gamma distribution with a mean of 12.0 and and standard deviation of 5.2 defined in [this paper](https://bmcmedicine.biomedcentral.com/articles/10.1186/s12916-014-0196-0). To convert these values into the shape and scale parameters required for a gamma distribution, we will use the function gamma\_mucv2shapescale from the **epitrix** package.

Here is a quick check to make sure the serial interval looks correct. We access the density of the gamma distribution we have just defined by $d, which is equivalent to calling dgamma:

#### Using serial interval estimates from the data

As we have data on dates of symptom onset and transmission links, we can also estimate the serial interval from the linelist by calculating the delay between onset dates of infector-infectee pairs. As we did in the **EpiNow2** section, we will use the get\_pairwise function from the **epicontacts** package, which allows us to calculate pairwise differences of linelist properties between transmission pairs. We first create an epicontacts object (see [Transmission chains](#transmission-chains) page for further details):

We then fit the difference in onset dates between transmission pairs, calculated using get\_pairwise, to a gamma distribution. We use the handy fit\_disc\_gamma from the **epitrix** package for this fitting procedure, as we require a discretised distribution.

#### Projecting incidence

To project future incidence, we still need to provide historical incidence in the form of an incidence object, as well as a sample of plausible Rt values. We will generate these values using the Rt estimates generated by **EpiEstim** in the previous section (under “Estimating Rt”) and stored in the epiestim\_res\_emp object. In the code below, we extract the mean and standard deviation estimates of Rt for the last time window of the outbreak (using the tail function to access the last element in a vector), and simulate 1000 values from a gamma distribution using rgamma. You can also provide your own vector of Rt values that you want to use for forward projections.

We then use the project() function to make the actual forecast. We specify how many days we want to project for via the n\_days arguments, and specify the number of simulations using the n\_sim argument.

We can then handily plot the incidence and projections using the plot() and add\_projections() functions. We can easily subset the incidence object to only show the most recent cases by using the square bracket operator.

You can also easily extract the raw estimates of daily case numbers by converting the output to a dataframe.

## Resources

* [Here is the paper](https://www.sciencedirect.com/science/article/pii/S1755436519300350) describing the methodology implemented in **EpiEstim**.
* [Here is the paper](https://wellcomeopenresearch.org/articles/5-112/v1) describing the methodology implemented in **EpiNow2**.
* [Here is a paper](https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008409) describing various methodological and practical considerations for estimating Rt.

# Contact tracing

This page demonstrates descriptive analysis of contact tracing data, addessing some key considerations and approaches unique to these kinds of data.

This page references many of the core R data management and visualisation competencies covered in other pages (e.g. data cleaning, pivoting, tables, time-series analyses), but we will highlight examples specific to contact tracing that have been useful for operational decision making. For example, this includes visualizing contact tracing follow-up data over time or across geographic areas, or producing clean Key Performance Indicator (KPI) tables for contact tracing supervisors.

For demonstration purposes we will use sample contact tracing data from the [Go.Data](https://www.who.int/tools/godata) platform. The principles covered here will apply for contact tracing data from other platforms - you may just need to undergo different data pre-processing steps depending on the structure of your data.

You can read more about the Go.Data project on the [Github Documentation site](https://worldhealthorganization.github.io/godata/) or [Community of Practice](https://community-godata.who.int/).

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

We will import sample datasets of contacts, and of their “follow-up”. These data have been retrieved and un-nested from the Go.Data API and stored as “.rds” files.

You can download all the example data for this handbook from the [Download handbook and data](#download-handbook-and-data) page.

If you want to download the example contact tracing data specific to this page, use the three download links below:

[Click to download the case investigation data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/cases_clean.rds?raw=true)

[Click to download the contact registration data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/contacts_clean.rds?raw=true)

[Click to download the contact follow-up data (.rds file)](https://github.com/WorldHealthOrganization/godata/blob/master/analytics/r-reporting/data/followups_clean.rds?raw=true)

In their original form in the downloadable files, the data reflect data as provided by the Go.Data API (learn about [APIs here](#import_api)). For example purposes here, we will clean the data to make it easier to read on this page. If you are using a Go.Data instance, you can view complete instructions on how to retrieve your data [here](https://github.com/WorldHealthOrganization/godata/tree/master/analytics/r-reporting).

Below, the datasets are imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data. We use here() to specify the file path - you should provide the file path specific to your computer. We then use select() to select only certain columns of the data, to simplify for purposes of demonstration.

#### Case data

These data are a table of the cases, and information about them.

Here are the nrow(cases) cases:

#### Contacts data

These data are a table of all the contacts and information about them. Again, provide your own file path. After importing we perform a few preliminary data cleaning steps including:

* Set age\_class as a factor and reverse the level order so that younger ages are first
* Select only certain column, while re-naming a one of them
* Artificially assign rows with missing admin level 2 to “Djembe”, to improve clarity of some example visualisations

Here are the nrow(contacts) rows of the contacts dataset:

#### Follow-up data

These data are records of the “follow-up” interactions with the contacts. Each contact is supposed to have an encounter each day for 14 days after their exposure.

We import and perform a few cleaning steps. We select certain columns, and also convert a character column to all lowercase values.

Here are the first 50 rows of the nrow(followups)-row followups dataset (each row is a follow-up interaction, with outcome status in the followup\_status column):

#### Relationships data

Here we import data showing the relationship between cases and contacts. We select certain column to show.

Below are the first 50 rows of the relationships dataset, which records all relationships between cases and contacts.

## Descriptive analyses

You can use the techniques covered in other pages of this handbook to conduct descriptive analyses of your cases, contacts, and their relationships. Below are some examples.

### Demographics

As demonstrated in the page covering [Demographic pyramids](#demographic-pyramids-and-likert-scales), you can visualise the age and gender distribution (here we use the **apyramid** package).

#### Age and Gender of contacts

The pyramid below compares the age distribution of contacts, by gender. Note that contacts missing age are included in their own bar at the top. You can change this default behavior, but then consider listing the number missing in a caption.

With the Go.Data data structure, the relationships dataset contains the ages of both cases and contacts, so you could use that dataset and create an age pyramid showing the differences between these two groups of people. The relationships data frame will be mutated to transform the numberic age columns into categories (see the [Cleaning data and core functions](#cleaning-data-and-core-functions) page). We also pivot the dataframe longer to facilitate easy plotting with **ggplot2** (see [Pivoting data](#pivoting-data)).

Now we can plot this transformed dataset with age\_pyramid() as before, but replacing gender with category (contact, or case).

We can also view other characteristics such as occupational breakdown (e.g. in form of a pie chart).

### Contacts per case

The number of contacts per case can be an important metric to assess quality of contact enumeration and the compliance of the population toward public health response.

Depending on your data structure, this can be assessed with a dataset that contains all cases and contacts. In the Go.Data datasets, the links between cases (“sources”) and contacts (“targets”) is stored in the relationships dataset.

In this dataset, each row is a contact, and the source case is listed in the row. There are no contacts who have relationships with multiple cases, but if this exists you may need to account for those before plotting (and explore them too!).

We begin by counting the number of rows (contacts) per source case. This is saved as a data frame.

We use geom\_histogram() to plot these data as a histogram.

## Contact Follow Up

Contact tracing data often contain “follow-up” data, which record outcomes of daily symptom checks of persons in quarantine. Analysis of this data can inform response strategy, identify contacts at-risk of loss-to-follow-up or at-risk of developing disease.

### Data cleaning

These data can exist in a variety of formats. They may exist as a “wide” format Excel sheet with one row per contact, and one column per follow-up “day”. See [Pivoting data](#pivoting-data) for descriptions of “long” and “wide” data and how to pivot data wider or longer.

In our Go.Data example, these data are stored in the followups data frame, which is in a “long” format with one row per follow-up interaction. The first 50 rows look like this:

**CAUTION:** Beware of duplicates when dealing with followup data; as there could be several erroneous followups on the same day for a given contact. Perhaps it seems to be an error but reflects reality - e.g. a contact tracer could submit a follow-up form early in the day when they could not reach the contact, and submit a second form when they were later reached. It will depend on the operational context for how you want to handle duplicates - just make sure to document your approach clearly.

Let’s see how many instances of “duplicate” rows we have:

In our example data, the only records that this applies to are ones missing an ID! We can remove those. But, for purposes of demonstration we will go show the steps for de-duplication so there is only one follow-up encoutner per person per day. See the page on [De-duplication](#de-duplication) for more detail. We will assume that the most recent encounter record is the correct one. We also take the opportunity to clean the followup\_number column (the “day” of follow-up which should range 1 - 14).

For each follow-up encounter, we have a follow-up status (such as whether the encounter occurred and if so, did the contact have symptoms or not). To see all the values we can run a quick tabyl() (from **janitor**) or table() (from **base** R) (see [Descriptive tables](#descriptive-tables)) by followup\_status to see the frequency of each of the outcomes.

In this dataset, “seen\_not\_ok” means “seen with symptoms”, and “seen\_ok” means “seen without symptoms”.

### Plot over time

As the dates data are continuous, we will use a histogram to plot them with date\_of\_followup assigned to the x-axis. We can achieve a “stacked” histogram by specifying a fill = argument within aes(), which we assign to the column followup\_status. Consequently, you can set the legend title using the fill = argument of labs().

We can see that the contacts were identified in waves (presumably corresponding with epidemic waves of cases), and that follow-up completion did not seemingly improve over the course of the epidemic.

**CAUTION:** If you are preparing many plots (e.g. for multiple jurisdictions) you will want the legends to appear identically even with varying levels of data completion or data composition. There may be plots for which not all follow-up statuses are present in the data, but you still want those categories to appear the legends. In ggplots (like above), you can specify the drop = FALSE argument of the scale\_fill\_discrete(). In tables, use tabyl() which shows counts for all factor levels, or if using count() from **dplyr** add the argument .drop = FALSE to include counts for all factor levels.

### Daily individual tracking

If your outbreak is small enough, you may want to look at each contact individually and see their status over the course of their follow-up. Fortunately, this followups dataset already contains a column with the day “number” of follow-up (1-14). If this does not exist in your data, you could create it by calculating the difference between the encounter date and the date follow-up was intended to begin for the contact.

A convenient visualisation mechanism (if the number of cases is not too large) can be a heat plot, made with geom\_tile(). See more details in the [heat plot] page.

### Analyse by group

Perhaps these follow-up data are being viewed on a daily or weekly basis for operational decision-making. You may want more meaningful disaggregations by geographic area or by contact-tracing team. We can do this by adjusting the columns provided to group\_by().

## KPI Tables

There are a number of different Key Performance Indicators (KPIs) that can be calculated and tracked at varying levels of disaggregations and across different time periods to monitor contact tracing performance. Once you have the calculations down and the basic table format; it is fairly easy to swap in and out different KPIs.

There are numerous sources of contact tracing KPIs, such as this one from [ResolveToSaveLives.org](https://contacttracingplaybook.resolvetosavelives.org/checklists/metrics). The majority of the work will be walking through your data structure and thinking through all of the inclusion/exclusion criteria. We show a few examples below; using Go.Data metadata structure:

| **Category** | **Indicator** | **Go.Data Numerator** | **Go.Data Denominator** |
| --- | --- | --- | --- |
| Process Indicator - Speed of Contact Tracing | % cases interviewed and isolated within 24h of case report | COUNT OF case\_id WHERE (date\_of\_reporting - date\_of\_data\_entry) < 1 day AND (isolation\_startdate - date\_of\_data\_entry) < 1 day | COUNT OF case\_id |
| Process Indicator - Speed of Contact Tracing | % contacts notified and quarantined within 24h of elicitation | COUNT OF contact\_id WHERE followup\_status == “SEEN\_NOT\_OK” OR “SEEN\_OK” AND date\_of\_followup - date\_of\_reporting < 1 day | COUNT OF contact\_id |
| Process Indicator - Completeness of Testing | % new symptomatic cases tested and interviewed within 3 days of onset of symptoms | COUNT OF case\_id WHERE (date\_of\_reporting - date\_of\_onset) < =3 days | COUNT OF case\_id |
| Outcome Indicator - Overall | % new cases among existing contact list | COUNT OF case\_id WHERE was\_contact == “TRUE” | COUNT OF case\_id |

Below we will walk through a sample exercise of creating a nice table visual to show contact follow-up across admin areas. At the end, we will make it fit for presentation with the **formattable** package (but you could use other packages like **flextable** - see [Tables for presentation](#tables-for-presentation)).

How you create a table like this will depend on the structure of your contact tracing data. Use the [Descriptive tables](#descriptive-tables) page to learn how to summarise data using **dplyr** functions.

We will create a table that will be dynamic and change as the data change. To make the results interesting, we will set a report\_date to allow us to simulate running the table on a certain day (we pick 10th June 2020). The data are filtered to that date.

Now, based on our data structure, we will do the following:

1. Begin with the followups data and summarise it to contain, for each unique contact:

* The date of latest record (no matter the status of the encounter)
* The date of latest encounter where the contact was “seen”
* The encounter status at that final “seen” encounter (e.g. with symptoms, without symptoms)

1. Join these data to the contacts data, which contains other information such as the overall contact status, date of last exposure to a case, etc. Also we will calculate metrics of interest for each contact such as days since last exposure
2. We group the enhanced contact data by geographic region (admin\_2\_name) and calculate summary statistics per region
3. Finally, we format the table nicely for presentation

First we summarise the follow-up data to get the information of interest:

Here is how these data look:

Now we will add this information to the contacts dataset, and calculate some additional columns.

Here is how these data look. Note contacts column to the right, and new calculated column at the far right.

Next we summarise the contacts data by region, to achieve a concise data frame of summary statistic columns.

And now we apply styling from the **formattable** and **knitr** packages, including a footnote that shows the “as of” date.

## Transmission Matrices

As discussed in the [Heat plots](#heat-plots) page, you can create a matrix of “who infected whom” using geom\_tile().

When new contacts are created, Go.Data stores this relationship information in the relationships API endpoint; and we can see the first 50 rows of this dataset below. This means that we can create a heat plot with relatively few steps given each contact is already joined to it’s source case.

As done above for the age pyramid comparing cases and contacts, we can select the few variables we need and create columns with categorical age groupings for both sources (cases) and targets (contacts).

As described previously, we create cross-tabulation;

convert into long format with proportions;

and create a heat-map for age.

## Resources

<https://github.com/WorldHealthOrganization/godata/tree/master/analytics/r-reporting>

<https://worldhealthorganization.github.io/godata/>

<https://community-godata.who.int/>

# Survey analysis

## Overview

This page demonstrates the use of several packages for survey analysis.

Most survey R packages rely on the [**survey** package](https://cran.r-project.org/web/packages/survey/index.html) for doing weighted analysis. We will use **survey** as well as [**srvyr**](https://cran.r-project.org/web/packages/srvyr/index.html) (a wrapper for **survey** allowing for tidyverse-style coding) and [**gtsummary**](https://cran.r-project.org/web/packages/gtsummary/index.html) (a wrapper for **survey** allowing for publication ready tables). While the original **survey** package does not allow for tidyverse-style coding, it does have the added benefit of allowing for survey-weighted generalised linear models (which will be added to this page at a later date). We will also demonstrate using a function from the [**sitrep**](https://github.com/R4EPI/sitrep) package to create sampling weights (n.b this package is currently not yet on CRAN, but can be installed from github).

Most of this page is based off work done for the [“R4Epis” project](https://r4epis.netlify.app/); for detailed code and R-markdown templates see the [“R4Epis” github page](https://github.com/R4EPI/sitrep). Some of the **survey** package based code is based off early versions of [EPIET case studies](https://github.com/EPIET/RapidAssessmentSurveys).

At current this page does not address sample size calculations or sampling. For a simple to use sample size calculator see [OpenEpi](https://www.openepi.com/Menu/OE_Menu.htm). The [GIS basics](https://epirhandbook.com/gis-basics.html) page of the handbook will eventually have a section on spatial random sampling, and this page will eventually have a section on sampling frames as well as sample size calculations.

1. Survey data
2. Observation time
3. Weighting
4. Survey design objects
5. Descriptive analysis
6. Weighted proportions
7. Weighted rates

## Preparation

### Packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.  
Here we also demonstrate using the p\_load\_gh() function from **pacman** to install a load a package from github which has not yet been published on CRAN.

### Load data

The example dataset used in this section:

* fictional mortality survey data.
* fictional population counts for the survey area.
* data dictionary for the fictional mortality survey data.

This is based off the MSF OCA ethical review board pre-approved survey. The fictional dataset was produced as part of the [“R4Epis” project](https://r4epis.netlify.app/). This is all based off data collected using [KoboToolbox](https://www.kobotoolbox.org/), which is a data collection software based off [Open Data Kit](https://opendatakit.org/).

Kobo allows you to export both the collected data, as well as the data dictionary for that dataset. We strongly recommend doing this as it simplifies data cleaning and is useful for looking up variables/questions.

**TIP:** The Kobo data dictionary has variable names in the “name” column of the survey sheet. Possible values for each variable are specified in choices sheet. In the choices tab, “name” has the shortened value and the “label::english” and “label::french” columns have the appropriate long versions. Using the **epidict** package msf\_dict\_survey() function to import a Kobo dictionary excel file will re-format this for you so it can be used easily to recode.

**CAUTION:** The example dataset is not the same as an export (as in Kobo you export different questionnaire levels individually) - see the survey data section below to merge the different levels.

The dataset is imported using the import() function from the **rio** package. See the page on [Import and export](https://epirhandbook.com/import-and-export.html) for various ways to import data.

The first 10 rows of the survey are displayed below.

We also want to import the data on sampling population so that we can produce appropriate weights. This data can be in different formats, however we would suggest to have it as seen below (this can just be typed in to an excel).

The first 10 rows of the survey are displayed below.

For cluster surveys you may want to add survey weights at the cluster level. You could read this data in as above. Alternatively if there are only a few counts, these could be entered as below in to a tibble. In any case you will need to have one column with a cluster identifier which matches your survey data, and another column with the number of households in each cluster.

### Clean data

The below makes sure that the date column is in the appropriate format. There are several other ways of doing this (see the [Working with dates](https://epirhandbook.com/working-with-dates.html) page for details), however using the dictionary to define dates is quick and easy.

We also create an age group variable using the age\_categories() function from **epikit** - see [cleaning data](https://epirhandbook.com/cleaning-data-and-core-functions.html#num_cats) handbook section for details. In addition, we create a character variable defining which district the various clusters are in.

Finally, we recode all of the yes/no variables to TRUE/FALSE variables - otherwise these cant be used by the **survey** proportion functions.

## Survey data

There numerous different sampling designs that can be used for surveys. Here we will demonstrate code for: - Stratified - Cluster - Stratified and cluster

As described above (depending on how you design your questionnaire) the data for each level would be exported as a separate dataset from Kobo. In our example there is one level for households and one level for individuals within those households.

These two levels are linked by a unique identifier. For a Kobo dataset this variable is "\_index" at the household level, which matches the "\_parent\_index" at the individual level. This will create new rows for household with each matching individual, see the handbook section on [joining](https://epirhandbook.com/joining-data.html) for details.

## Observation time

For mortality surveys we want to now how long each individual was present for in the location to be able to calculate an appropriate mortality rate for our period of interest. This is not relevant to all surveys, but particularly for mortality surveys this is important as they are conducted frequently among mobile or displaced populations.

To do this we first define our time period of interest, also known as a recall period (i.e. the time that participants are asked to report on when answering questions). We can then use this period to set inappropriate dates to missing, i.e. if deaths are reported from outside the period of interest.

We can then use our date variables to define start and end dates for each individual. We can use the find\_start\_date() function from **sitrep** to fine the causes for the dates and then use that to calculate the difference between days (person-time).

start date: Earliest appropriate arrival event within your recall period Either the beginning of your recall period (which you define in advance), or a date after the start of recall if applicable (e.g. arrivals or births)

end date: Earliest appropriate departure event within your recall period Either the end of your recall period, or a date before the end of recall if applicable (e.g. departures, deaths)

## Weighting

It is important that you drop erroneous observations before adding survey weights. For example if you have observations with negative observation time, you will need to check those (you can do this with the assert\_positive\_timespan() function from **sitrep**. Another thing is if you want to drop empty rows (e.g. with drop\_na(uid)) or remove duplicates (see handbook section on [De-duplication](#de-duplication) for details). Those without consent need to be dropped too.

In this example we filter for the cases we want to drop and store them in a separate data frame - this way we can describe those that were excluded from the survey. We then use the anti\_join() function from **dplyr** to remove these dropped cases from our survey data.

**DANGER:** You cant have missing values in your weight variable, or any of the variables relevant to your survey design (e.g. age, sex, strata or cluster variables).

As mentioned above we demonstrate how to add weights for three different study designs (stratified, cluster and stratified cluster). These require information on the source population and/or the clusters surveyed. We will use the stratified cluster code for this example, but use whichever is most appropriate for your study design.

## Survey design objects

Create survey object according to your study design. Used the same way as data frames to calculate weight proportions etc. Make sure that all necessary variables are created before this.

There are four options, comment out those you do not use: - Simple random - Stratified - Cluster - Stratified cluster

For this template - we will pretend that we cluster surveys in two separate strata (health districts A and B). So to get overall estimates we need have combined cluster and strata weights.

As mentioned previously, there are two packages available for doing this. The classic one is **survey** and then there is a wrapper package called **srvyr** that makes tidyverse-friendly objects and functions. We will demonstrate both, but note that most of the code in this chapter will use **srvyr** based objects. The one exception is that the **gtsummary** package only accepts **survey** objects.

### ****Survey**** package

The **survey** package effectively uses **base** R coding, and so it is not possible to use pipes (%>%) or other **dplyr** syntax. With the **survey** package we use the svydesign() function to define a survey object with appropriate clusters, weights and strata.

**NOTE:** we need to use the tilde (~) in front of variables, this is because the package uses the **base** R syntax of assigning variables based on formulae.

### ****Srvyr**** package

With the **srvyr** package we can use the as\_survey\_design() function, which has all the same arguments as above but allows pipes (%>%), and so we do not need to use the tilde (~).

## Descriptive analysis

Basic descriptive analysis and visualisation is covered extensively in other chapters of the handbook, so we will not dwell on it here. For details see the chapters on [descriptive tables](https://epirhandbook.com/descriptive-tables.html), [statistical tests](https://epirhandbook.com/simple-statistical-tests.html), [tables for presentation](https://epirhandbook.com/tables-for-presentation.html), [ggplot basics](https://epirhandbook.com/ggplot-basics.html) and [R markdown reports](https://epirhandbook.com/r-markdown-reports.html).

In this section we will focus on how to investigate bias in your sample and visualise this. We will also look at visualising population flow in a survey setting using alluvial/sankey diagrams.

In general, you should consider including the following descriptive analyses:

* Final number of clusters, households and individuals included
* Number of excluded individuals and the reasons for exclusion
* Median (range) number of households per cluster and individuals per household

### Sampling bias

Compare the proportions in each age group between your sample and the source population. This is important to be able to highlight potential sampling bias. You could similarly repeat this looking at distributions by sex.

Note that these p-values are just indicative, and a descriptive discussion (or visualisation with age-pyramids below) of the distributions in your study sample compared to the source population is more important than the binomial test itself. This is because increasing sample size will more often than not lead to differences that may be irrelevant after weighting your data.

### Demographic pyramids

Demographic (or age-sex) pyramids are an easy way of visualising the distribution in your survey population. It is also worth considering creating [descriptive tables](https://epirhandbook.com/descriptive-tables.html) of age and sex by survey strata. We will demonstrate using the **apyramid** package as it allows for weighted proportions using our survey design object created above. Other options for creating [demographic pyramids](https://epirhandbook.com/demographic-pyramids-and-likert-scales.html) are covered extensively in that chapter of the handbook. We will also use a wrapper function from **sitrep** called plot\_age\_pyramid() which saves a few lines of coding for producing a plot with proportions.

As with the formal binomial test of difference, seen above in the sampling bias section, we are interested here in visualising whether our sampled population is substantially different from the source population and whether weighting corrects this difference. To do this we will use the **patchwork** package to show our **ggplot** visualisations side-by-side; for details see the section on combining plots in [ggplot tips](https://epirhandbook.com/ggplot-tips.html?q=patch#combine-plots) chapter of the handbook. We will visualise our source population, our un-weighted survey population and our weighted survey population. You may also consider visualising by each strata of your survey - in our example here that would be by using the argument stack\_by = "health\_district" (see ?plot\_age\_pyramid for details).

**NOTE:** The x and y axes are flipped in pyramids

### Alluvial/sankey diagram

Visualising starting points and outcomes for individuals can be very helpful to get an overview. There is quite an obvious application for mobile populations, however there are numerous other applications such as cohorts or any other situation where there are transitions in states for individuals. These diagrams have several different names including alluvial, sankey and parallel sets - the details are in the handbook chapter on [diagrams and charts](https://epirhandbook.com/diagrams-and-charts.html#alluvialsankey-diagrams).

## Weighted proportions

This section will detail how to produce tables for weighted counts and proportions, with associated confidence intervals and design effect. There are four different options using functions from the following packages: **survey**, **srvyr**, **sitrep** and **gtsummary**. For minimal coding to produce a standard epidemiology style table, we would recommend the **sitrep** function - which is a wrapper for **srvyr** code; note however that this is not yet on CRAN and may change in the future. Otherwise, the **survey** code is likely to be the most stable long-term, whereas **srvyr** will fit most nicely within tidyverse work-flows. While **gtsummary** functions hold a lot of potential, they appear to be experimental and incomplete at the time of writing.

### ****Survey**** package

We can use the svyciprop() function from **survey** to get weighted proportions and accompanying 95% confidence intervals. An appropriate design effect can be extracted using the svymean() rather than svyprop() function. It is worth noting that svyprop() only appears to accept variables between 0 and 1 (or TRUE/FALSE), so categorical variables will not work.

**NOTE:** Functions from **survey** also accept **srvyr** design objects, but here we have used the **survey** design object just for consistency

We can combine the functions from **survey** shown above in to a function which we define ourselves below, called svy\_prop; and we can then use that function together with map() from the purrr package to iterate over several variables and create a table. See the handbook [iteration](https://epirhandbook.com/iteration-loops-and-lists.html) chapter for details on **purrr**.

### ****Srvyr**** package

With **srvyr** we can use **dplyr** syntax to create a table. Note that the survey\_mean() function is used and the proportion argument is specified, and also that the same function is used to calculate design effect. This is because **srvyr** wraps around both of the **survey** package functions svyciprop() and svymean(), which are used in the above section.

**NOTE:** It does not seem to be possible to get proportions from categorical variables using **srvyr** either, if you need this then check out the section below using **sitrep**

Here too we could write a function to then iterate over multiple variables using the **purrr** package. See the handbook [iteration](https://epirhandbook.com/iteration-loops-and-lists.html) chapter for details on **purrr**.

### ****Sitrep**** package

The tab\_survey() function from **sitrep** is a wrapper for **srvyr**, allowing you to create weighted tables with minimal coding. It also allows you to calculate weighted proportions for categorical variables.

### ****Gtsummary**** package

With **gtsummary** there does not seem to be inbuilt functions yet to add confidence intervals or design effect. Here we show how to define a function for adding confidence intervals and then add confidence intervals to a **gtsummary** table created using the tbl\_svysummary() function.

## Weighted ratios

Similarly for weighted ratios (such as for mortality ratios) you can use the **survey** or the **srvyr** package. You could similarly write functions (similar to those above) to iterate over several variables. You could also create a function for **gtsummary** as above but currently it does not have inbuilt functionality.

### ****Survey**** package

### ****Srvyr**** package

## Resources

[UCLA stats page](https://stats.idre.ucla.edu/r/seminars/survey-data-analysis-with-r/)

[Analyze survey data free](http://asdfree.com/)

[srvyr packge](http://gdfe.co/srvyr/)

[gtsummary package](http://www.danieldsjoberg.com/gtsummary/reference/index.html)

[EPIET survey case studies](https://github.com/EPIET/RapidAssessmentSurveys)

# Survival analysis

## Overview

Survival analysis focuses on describing for a given individual or group of individuals, a defined point of event called **the failure** (occurrence of a disease, cure from a disease, death, relapse after response to treatment…) that occurs after a period of time called **failure time** (or **follow-up time** in cohort/population-based studies) during which individuals are observed. To determine the failure time, it is then necessary to define a time of origin (that can be the inclusion date, the date of diagnosis…).

The target of inference for survival analysis is then the time between an origin and an event. In current medical research, it is widely used in clinical studies to assess the effect of a treatment for instance, or in cancer epidemiology to assess a large variety of cancer survival measures.

It is usually expressed through the **survival probability** which is the probability that the event of interest has not occurred by a duration t.

**Censoring**: Censoring occurs when at the end of follow-up, some of the individuals have not had the event of interest, and thus their true time to event is unknown. We will mostly focus on right censoring here but for more details on censoring and survival analysis in general, you can see references.

## Preparation

### Load packages

To run survival analyses in R, one the most widely used package is the **survival** package. We first install it and then load it as well as the other packages that will be used in this section:

In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

This page explores survival analyses using the linelist used in most of the previous pages and on which we apply some changes to have a proper survival data.

### Import dataset

We import the dataset of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

### Data management and transformation

In short, survival data can be described as having the following three characteristics:

1. the dependent variable or response is the waiting time until the occurrence of a well-defined event,
2. observations are censored, in the sense that for some units the event of interest has not occurred at the time the data are analyzed, and
3. there are predictors or explanatory variables whose effect on the waiting time we wish to assess or control.

Thus, we will create different variables needed to respect that structure and run the survival analysis.

We define:

* a new data frame linelist\_surv for this analysis
* our event of interest as being “death” (hence our survival probability will be the probability of being alive after a certain time after the time of origin),
* the follow-up time (futime) as the time between the time of onset and the time of outcome in days,
* censored patients as those who recovered or for whom the final outcome is not known ie the event “death” was not observed (event=0).

**CAUTION:** Since in a real cohort study, the information on the time of origin and the end of the follow-up is known given individuals are observed, we will remove observations where the date of onset or the date of outcome is unknown. Also the cases where the date of onset is later than the date of outcome will be removed since they are considered as wrong.

**TIP:** Given that filtering to greater than (>) or less than (<) a date can remove rows with missing values, applying the filter on the wrong dates will also remove the rows with missing dates.

We then use case\_when() to create a column age\_cat\_small in which there are only 3 age categories.

**TIP:** We can verify the new columns we have created by doing a summary on the futime and a cross-tabulation between event and outcome from which it was created. Besides this verification it is a good habit to communicate the median follow-up time when interpreting survival analysis results.

Now we cross-tabulate the new age\_cat\_small var and the old age\_cat col to ensure correct assingments

Now we review the 10 first observations of the linelist\_surv data looking at specific variables (including those newly created).

We can also cross-tabulate the columns age\_cat\_small and gender to have more details on the distribution of this new column by gender. We use tabyl() and the adorn functions from **janitor** as described in the [Descriptive tables](#descriptive-tables) page.

## Basics of survival analysis

### Building a surv-type object

We will first use Surv() from **survival** to build a survival object from the follow-up time and event columns.

The result of such a step is to produce an object of type Surv that condenses the time information and whether the event of interest (death) was observed. This object will ultimately be used in the right-hand side of subsequent model formulae (see [documentation](https://cran.r-project.org/web/packages/survival/vignettes/survival.pdf)).

To review, here are the first 10 rows of the linelist\_surv data, viewing only some important columns.

And here are the first 10 elements of survobj. It prints as essentially a vector of follow-up time, with “+” to represent if an observation was right-censored. See how the numbers align above and below.

### Running initial analyses

We then start our analysis using the survfit() function to produce a survfit object, which fits the default calculations for **Kaplan Meier** (KM) estimates of the overall (marginal) survival curve, which are in fact a step function with jumps at observed event times. The final survfit object contains one or more survival curves and is created using the Surv object as a response variable in the model formula.

**NOTE:** The Kaplan-Meier estimate is a nonparametric maximum likelihood estimate (MLE) of the survival function. . (see resources for more information).

The summary of this survfit object will give what is called a life table. For each time step of the follow-up (time) where an event happened (in ascending order):

* the number of people who were at risk of developing the event (people who did not have the event yet nor were censored: n.risk)
* those who did develop the event (n.event)
* and from the above: the probability of not developing the event (probability of not dying, or of surviving past that specific time)
* finally, the standard error and the confidence interval for that probability are derived and displayed

We fit the KM estimates using the formula where the previously Surv object “survobj” is the response variable. “~ 1” precises we run the model for the overall survival.

While using summary() we can add the option times and specify certain times at which we want to see the survival information

We can also use the print() function. The print.rmean = TRUE argument is used to obtain the mean survival time and its standard error (se).

**NOTE:** The restricted mean survival time (RMST) is a specific survival measure more and more used in cancer survival analysis and which is often defined as the area under the survival curve, given we observe patients up to restricted time T (more details in Resources section).

**TIP:** We can create the surv object directly in the survfit() function and save a line of code. This will then look like: linelistsurv\_quick <- survfit(Surv(futime, event) ~ 1, data=linelist\_surv).

### Cumulative hazard

Besides the summary() function, we can also use the str() function that gives more details on the structure of the survfit() object. It is a list of 16 elements.

Among these elements is an important one: cumhaz, which is a numeric vector. This could be plotted to allow show the **cumulative hazard**, with the **hazard** being the **instantaneous rate of event occurrence** (see references).

### Plotting Kaplan-Meir curves

Once the KM estimates are fitted, we can visualize the probability of being alive through a given time using the basic plot() function that draws the “Kaplan-Meier curve”. In other words, the curve below is a conventional illustration of the survival experience in the whole patient group.

We can quickly verify the follow-up time min and max on the curve.

An easy way to interpret is to say that at time zero, all the participants are still alive and survival probability is then 100%. This probability decreases over time as patients die. The proportion of participants surviving past 60 days of follow-up is around 40%.

The confidence interval of the KM survival estimates are also plotted by default and can be dismissed by adding the option conf.int = FALSE to the plot() command.

Since the event of interest is “death”, drawing a curve describing the complements of the survival proportions will lead to drawing the cumulative mortality proportions. This can be done with lines(), which adds information to an existing plot.

## Comparison of survival curves

To compare the survival within different groups of our observed participants or patients, we might need to first look at their respective survival curves and then run tests to evaluate the difference between independent groups. This comparison can concern groups based on gender, age, treatment, comorbidity…

### Log rank test

The log rank test is a popular test that compares the entire survival experience between two or more independent groups and can be thought of as a test of whether the survival curves are identical (overlapping) or not (null hypothesis of no difference in survival between the groups). The survdiff() function of the **survival package** allows running the log-rank test when we specify rho = 0 (which is the default). The test results gives a chi-square statistic along with a p-value since the log rank statistic is approximately distributed as a chi-square test statistic.

We first try to compare the survival curves by gender group. For this, we first try to visualize it (check whether the two survival curves are overlapping). A new survfit object will be created with a slightly different formula. Then the survdiff object will be created.

By supplying ~ gender as the right side of the formula, we no longer plot the overall survival but instead by gender.

Now we can plot the survival curves by gender. Have a look at the order of the strata levels in the gender column before defining your colors and legend.

And now we can compute the test of the difference between the survival curves using survdiff()

We see that the survival curve for women and the one for men overlap and the log-rank test does not give evidence of a survival difference between women and men.

Some other R packages allow illustrating survival curves for different groups and testing the difference all at once. Using the ggsurvplot() function from the **survminer** package, we can also include in our curve the printed risk tables for each group, as well the p-value from the log-rank test.

**CAUTION:** **survminer** functions require that you specify the survival object and again specify the data used to fit the survival object. Remember to do this to avoid non-specific error messages.

We may also want to test for differences in survival by the source of infection (source of contamination).

In this case, the Log rank test gives enough evidence of a difference in the survival probabilities at alpha= 0.005. The survival probabilities for patients that were infected at funerals are higher than the survival probabilities for patients that got infected in other places, suggesting a survival benefit.

## Cox regression analysis

Cox proportional hazards regression is one of the most popular regression techniques for survival analysis. Other models can also be used since the Cox model requires important assumptions that need to be verified for an appropriate use such as the proportional hazards assumption: see references.

In a Cox proportional hazards regression model, the measure of effect is the **hazard rate** (HR), which is the risk of failure (or the risk of death in our example), given that the participant has survived up to a specific time. Usually, we are interested in comparing independent groups with respect to their hazards, and we use a hazard ratio, which is analogous to an odds ratio in the setting of multiple logistic regression analysis. The cox.ph() function from the **survival** package is used to fit the model. The function cox.zph() from **survival** package may be used to test the proportional hazards assumption for a Cox regression model fit.

**NOTE:** A probability must lie in the range 0 to 1. However, the hazard represents the expected number of events per one unit of time.

* If the hazard ratio for a predictor is close to 1 then that predictor does not affect survival,
* if the HR is less than 1, then the predictor is protective (i.e., associated with improved survival),
* and if the HR is greater than 1, then the predictor is associated with increased risk (or decreased survival).

### Fitting a Cox model

We can first fit a model to assess the effect of age and gender on the survival. By just printing the model, we have the information on:

* the estimated regression coefficients coef which quantifies the association between the predictors and the outcome,
* their exponential (for interpretability, exp(coef)) which produces the hazard ratio,
* their standard error se(coef),
* the z-score: how many standard errors is the estimated coefficient away from 0,
* and the p-value: the probability that the estimated coefficient could be 0.

The summary() function applied to the cox model object gives more information, such as the confidence interval of the estimated HR and the different test scores.

The effect of the first covariate gender is presented in the first row. genderm (male) is printed, implying that the first strata level (“f”), i.e the female group, is the reference group for the gender. Thus the interpretation of the test parameter is that of men compared to women. The p-value indicates there was not enough evidence of an effect of the gender on the expected hazard or of an association between gender and all-cause mortality.

The same lack of evidence is noted regarding age-group.

It was interesting to run the model and look at the results but a first look to verify whether the proportional hazards assumptions is respected could help saving time.

**NOTE:** A second argument called method can be specified when computing the cox model, that determines how ties are handled. The default is “efron”, and the other options are “breslow” and “exact”.

In another model we add more risk factors such as the source of infection and the number of days between date of onset and admission. This time, we first verify the proportional hazards assumption before going forward.

In this model, we have included a continuous predictor (days\_onset\_hosp). In this case we interpret the parameter estimates as the increase in the expected log of the relative hazard for each one unit increase in the predictor, holding other predictors constant. We first verify the proportional hazards assumption.

The graphical verification of this assumption may be performed with the function ggcoxzph() from the **survminer** package.

The model results indicate there is a negative association between onset to admission duration and all-cause mortality. The expected hazard is 0.9 times lower in a person who who is one day later admitted than another, holding gender constant. Or in a more straightforward explanation, a one unit increase in the duration of onset to admission is associated with a 10.7% (coef \*100) decrease in the risk of death.

Results show also a positive association between the source of infection and the all-cause mortality. Which is to say there is an increased risk of death (1.21x) for patients that got a source of infection other than funerals.

We can verify this relationship with a table:

We would need to consider and investigate why this association exists in the data. One possible explanation could be that patients who live long enough to be admitted later had less severe disease to begin with. Another perhaps more likely explanation is that since we used a simulated fake dataset, this pattern does not reflect reality!

### Forest plots

We can then visualize the results of the cox model using the practical forest plots with the ggforest() function of the **survminer package**.

## Time-dependent covariates in survival models

Some of the following sections have been adapted with permission from an excellent [introduction to survival analysis in R](https://www.emilyzabor.com/tutorials/survival_analysis_in_r_tutorial.html) by [Dr. Emily Zabor](https://www.emilyzabor.com/)

In the last section we covered using Cox regression to examine associations between covariates of interest and survival outcomes.But these analyses rely on the covariate being measured at baseline, that is, before follow-up time for the event begins.

What happens if you are interested in a covariate that is measured **after** follow-up time begins? Or, what if you have a covariate that can change over time?

For example, maybe you are working with clinical data where you repeated measures of hospital laboratory values that can change over time. This is an example of a **Time Dependent Covariate**. In order to address this you need a special setup, but fortunately the cox model is very flexible and this type of data can also be modeled with tools from the **survival** package.

### Time-dependent covariate setup

Analysis of time-dependent covariates in R requires setup of a special dataset. If interested, see the more detailed paper on this by the author of the **survival** package [Using Time Dependent Covariates and Time Dependent Coefficients in the Cox Model](https://cran.r-project.org/web/packages/survival/vignettes/timedep.pdf).

For this, we’ll use a new dataset from the SemiCompRisks package named BMT, which includes data on 137 bone marrow transplant patients. The variables we’ll focus on are:

* T1 - time (in days) to death or last follow-up
* delta1 - death indicator; 1-Dead, 0-Alive
* TA - time (in days) to acute graft-versus-host disease
* deltaA - acute graft-versus-host disease indicator;
  + 1 - Developed acute graft-versus-host disease
  + 0 - Never developed acute graft-versus-host disease

We’ll load this dataset from the **survival** package using the **base** R command data(), which can be used for loading data that is already included in a R package that is loaded. The data frame BMT will appear in your R environment.

#### Add unique patient identifier

There is no unique ID column in the BMT data, which is needed to create the type of dataset we want. So we use the function rowid\_to\_column() from the **tidyverse** package **tibble** to create a new id column called my\_id (adds column at start of data frame with sequential row ids, starting at 1). We name the data frame bmt.

The dataset now looks like this:

#### Expand patient rows

Next, we’ll use the tmerge() function with the event() and tdc() helper functions to create the restructured dataset. Our goal is to restructure the dataset to create a separate row for each patient for each time interval where they have a different value for deltaA. In this case, each patient can have at most two rows depending on whether they developed acute graft-versus-host disease during the data collection period. We’ll call our new indicator for the development of acute graft-versus-host disease agvhd.

* tmerge() creates a long dataset with multiple time intervals for the different covariate values for each patient
* event() creates the new event indicator to go with the newly-created time intervals
* tdc() creates the time-dependent covariate column, agvhd, to go with the newly created time intervals

To see what this does, let’s look at the data for the first 5 individual patients.

The variables of interest in the original data looked like this:

The new dataset for these same patients looks like this:

Now some of our patients have two rows in the dataset corresponding to intervals where they have a different value of our new variable, agvhd. For example, Patient 1 now has two rows with a agvhd value of zero from time 0 to time 67, and a value of 1 from time 67 to time 2081.

### Cox regression with time-dependent covariates

Now that we’ve reshaped our data and added the new time-dependent aghvd variable, let’s fit a simple single variable cox regression model. We can use the same coxph() function as before, we just need to change our Surv() function to specify both the start and stop time for each interval using the time1 = and time2 = arguments.

Again, we’ll visualize our cox model results using the ggforest() function from the **survminer package**.:

As you can see from the forest plot, confidence interval, and p-value, there does not appear to be a strong association between death and acute graft-versus-host disease in the context of our simple model.

## Resources

[Survival Analysis Part I: Basic concepts and first analyses](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2394262/)

[Survival Analysis in R](https://www.emilyzabor.com/tutorials/survival_analysis_in_r_tutorial.html)

[Survival analysis in infectious disease research: Describing events in time](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2954271/)

[Chapter on advanced survival models Princeton](https://data.princeton.edu/wws509/notes/c7.pdf)

[Using Time Dependent Covariates and Time Dependent Coefficients in the Cox Model](https://cran.r-project.org/web/packages/survival/vignettes/timedep.pdf)

[Survival analysis cheatsheet R](https://publicifsv.sund.ku.dk/~ts/survival/survival-cheat.pdf)

[Survminer cheatsheet](https://paulvanderlaken.files.wordpress.com/2017/08/survminer_cheatsheet.pdf)

[Paper on different survival measures for cancer registry data with Rcode provided as supplementary materials](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6322561/)

# GIS basics

## Overview

Spatial aspects of your data can provide a lot of insights into the situation of the outbreak, and to answer questions such as:

* Where are the current disease hotspots?
* How have the hotspots have changed over time?
* How is the access to health facilities? Are any improvements needed?

The current focus of this GIS page to address the needs of applied epidemiologists in outbreak response. We will explore basic spatial data visualization methods using **tmap** and **ggplot2** packages. We will also walk through some of the basic spatial data management and querying methods with the **sf** package. Lastly, we will briefly touch upon concepts of spatial statistics such as spatial relationships, spatial autocorrelation, and spatial regression using the **spdep** package.

## Key terms

Below we introduce some key terminology. For a thorough introduction to GIS and spatial analysis, we suggest that you review one of the longer tutorials or courses listed in the References section.

**Geographic Information System (GIS)** - A GIS is a framework or environment for gathering, managing, analyzing, and visualizing spatial data.

### GIS software

Some popular GIS software allow point-and-click interaction for map development and spatial analysis. These tools comes with advantages such as not needing to learn code and the ease of manually selecting and placing icons and features on a map. Here are two popular ones:

**ArcGIS** - A commercial GIS software developed by the company ESRI, which is very popular but quite expensive

**QGIS** - A free open-source GIS software that can do almost anything that ArcGIS can do. You can [download QGIS here](https://qgis.org/en/site/forusers/download.html)

Using R as a GIS can seem more intimidating at first because instead of “point-and-click”, it has a “command-line interface” (you must code to acquire the desired outcome). However, this is a major advantage if you need to repetitively produce maps or create an analysis that is reproducible.

### Spatial data

The two primary forms of spatial data used in GIS are vector and raster data:

**Vector Data** - The most common format of spatial data used in GIS, vector data are comprised of geometric features of vertices and paths. Vector spatial data can be further divided into three widely-used types:

* Points - A point consists of a coordinate pair (x,y) representing a specific location in a coordinate system. Points are the most basic form of spatial data, and may be used to denote a case (i.e. patient home) or a location (i.e. hospital) on a map.
* Lines - A line is composed of two connected points. Lines have a length, and may be used to denote things like roads or rivers.
* Polygons - A polygon is composed of at least three line segments connected by points. Polygon features have a length (i.e. the perimeter of the area) as well as an area measurement. Polygons may be used to note an area (i.e. a village) or a structure (i.e. the actual area of a hospital).

**Raster Data** - An alternative format for spatial data, raster data is a matrix of cells (e.g. pixels) with each cell containing information such as height, temperature, slope, forest cover, etc. These are often aerial photographs, satellite imagery, etc. Rasters can also be used as “base maps” below vector data.

### Visualizing spatial data

To visually represent spatial data on a map, GIS software requires you to provide sufficient information about where different features should be, in relation to one another. If you are using vector data, which will be true for most use cases, this information will typically be stored in a shapefile:

**Shapefiles** - A shapefile is a common data format for storing “vector” spatial data consisting or lines, points, or polygons. A single shapefile is actually a collection of at least three files - .shp, .shx, and .dbf. All of these sub-component files must be present in a given directory (folder) for the shapefile to be readable. These associated files can be compressed into a ZIP folder to be sent via email or download from a website.

The shapefile will contain information about the features themselves, as well as where to locate them on the Earth’s surface. This is important because while the Earth is a globe, maps are typically two-dimensional; choices about how to “flatten” spatial data can have a big impact on the look and interpretation of the resulting map.

**Coordinate Reference Systems (CRS)** - A CRS is a coordinate-based system used to locate geographical features on the Earth’s surface. It has a few key components:

* Coordinate System - There are many many different coordinate systems, so make sure you know which system your coordinates are from. Degrees of latitude/longitude are common, but you could also see [UTM](https://www.maptools.com/tutorials/utm/quick_guide) coordinates.
* Units - Know what the units are for your coordinate system (e.g. decimal degrees, meters)
* Datum - A particular modeled version of the Earth. These have been revised over the years, so ensure that your map layers are using the same datum.
* Projection - A reference to the mathematical equation that was used to project the truly round earth onto a flat surface (map).

Remember that you can summarise spatial data without using the mapping tools shown below. Sometimes a simple table by geography (e.g. district, country, etc.) is all that is needed!

## Getting started with GIS

There are a couple of key items you will need to have and to think about to make a map. These include:

* A **dataset** – this can be in a spatial data format (such as shapefiles, as noted above) or it may not be in a spatial format (for instance just as a csv).
* If your dataset is not in a spatial format you will also need a **reference dataset**. Reference data consists of the spatial representation of the data and the related **attributes**, which would include material containing the location and address information of specific features.
  + If you are working with pre-defined geographic boundaries (for example, administrative regions), reference shapefiles are often freely available to download from a government agency or data sharing organization. When in doubt, a good place to start is to Google “[regions] shapefile”
  + If you have address information, but no latitude and longitude, you may need to use a **geocoding engine** to get the spatial reference data for your records.
* An idea about **how you want to present** the information in your datasets to your target audience. There are many different types of maps, and it is important to think about which type of map best fits your needs.

### Types of maps for visualizing your data

**Choropleth map** - a type of thematic map where colors, shading, or patterns are used to represent geographic regions in relation to their value of an attribute. For instance a larger value could be indicated by a darker colour than a smaller value. This type of map is particularly useful when visualizing a variable and how it changes across defined regions or geopolitical areas.

**Case density heatmap** - a type of thematic map where colours are used to represent intensity of a value, however, it does not use defined regions or geopolitical boundaries to group data. This type of map is typically used for showing ‘hot spots’ or areas with a high density or concentration of points.

**Dot density map** - a thematic map type that uses dots to represent attribute values in your data. This type of map is best used to visualize the scatter of your data and visually scan for clusters.

**Proportional symbols map (graduated symbols map)** - a thematic map similar to a choropleth map, but instead of using colour to indicate the value of an attribute it uses a symbol (usually a circle) in relation to the value. For instance a larger value could be indicated by a larger symbol than a smaller value. This type of map is best used when you want to visualize the size or quantity of your data across geographic regions.

You can also combine several different types of visualizations to show complex geographic patterns. For example, the cases (dots) in the map below are colored according to their closest health facility (see legend). The large red circles show health facility catchment areas of a certain radius, and the bright red case-dots those that were outside any catchment range:

Note: The primary focus of this GIS page is based on the context of field outbreak response. Therefore the contents of the page will cover the basic spatial data manipulations, visualizations, and analyses.

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

You can see an overview of all the R packages that deal with spatial data at the [CRAN “Spatial Task View”](https://cran.r-project.org/web/views/Spatial.html).

### Sample case data

For demonstration purposes, we will work with a random sample of 1000 cases from the simulated Ebola epidemic linelist dataframe (computationally, working with fewer cases is easier to display in this handbook). If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file).

Since we are taking a random sample of the cases, your results may look slightly different from what is demonstrated here when you run the codes on your own.

Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

Next we select a random sample of 1000 rows using sample() from **base** R.

Now we want to convert this linelist which is class dataframe, to an object of class “sf” (spatial features). Given that the linelist has two columns “lon” and “lat” representing the longitude and latitude of each case’s residence, this will be easy.

We use the package **sf** (spatial features) and its function st\_as\_sf() to create the new object we call linelist\_sf. This new object looks essentially the same as the linelist, but the columns lon and lat have been designated as coordinate columns, and a coordinate reference system (CRS) has been assigned for when the points are displayed. 4326 identifies our coordinates as based on the [World Geodetic System 1984 (WGS84)](https://gisgeography.com/wgs84-world-geodetic-system/) - which is standard for GPS coordinates.

This is how the original linelist dataframe looks like. In this demonstration, we will only use the column date\_onset and geometry (which was constructed from the longitude and latitude fields above and is the last column in the data frame).

### Admin boundary shapefiles

**Sierra Leone: Admin boundary shapefiles**

In advance, we have downloaded all administrative boundaries for Sierra Leone from the Humanitarian Data Exchange (HDX) [website here](https://data.humdata.org/dataset/sierra-leone-all-ad-min-level-boundaries). Alternatively, you can download these and all other example data for this handbook via our R package, as explained in the [Download handbook and data](#download-handbook-and-data) page.

Now we are going to do the following to save the Admin Level 3 shapefile in R:

1. Import the shapefile
2. Clean the column names
3. Filter rows to keep only areas of interest

To import a shapefile we use the read\_sf() function from **sf**. It is provided the filepath via here(). - in our case the file is within our R project in the “data”, “gis”, and “shp” subfolders, with filename “sle\_adm3.shp” (see pages on [Import and export](#import-and-export) and [R projects](#r-projects) for more information). You will need to provide your own file path.

Next we use clean\_names() from the **janitor** package to standardize the column names of the shapefile. We also use filter() to keep only the rows with admin2name of “Western Area Urban” or “Western Area Rural”.

Below you can see the how the shapefile looks after import and cleaning. Scroll to the right to see how there are columns with admin level 0 (country), admin level 1, admin level 2, and finally admin level 3. Each level has a character name and a unique identifier “pcode”. The pcode expands with each increasing admin level e.g. SL (Sierra Leone) -> SL04 (Western) -> SL0410 (Western Area Rural) -> SL040101 (Koya Rural).

### Population data

**Sierra Leone: Population by ADM3**

These data can again be downloaded from HDX (link [here](https://data.humdata.org/dataset/sierra-leone-population)) or via our **epirhandbook** R package as explained [in this page](#download-handbook-and-data). We use import() to load the .csv file. We also pass the imported file to clean\_names() to standardize the column name syntax.

Here is what the population file looks like. Scroll to the right to see how each jurisdiction has columns with male population, female populaton, total population, and the population break-down in columns by age group.

### Health Facilities

**Sierra Leone: Health facility data from OpenStreetMap**

Again we have downloaded the locations of health facilities from HDX [here](https://data.humdata.org/dataset/hotosm_sierra_leone_health_facilities) or via instructions in the [Download handbook and data](#download-handbook-and-data) page.

We import the facility points shapefile with read\_sf(), again clean the column names, and then filter to keep only the points tagged as either “hospital”, “clinic”, or “doctors”.

Here is the resulting dataframe - scroll right to see the facility name and geometry coordinates.

## Plotting coordinates

The easiest way to plot X-Y coordinates (longitude/latitude, points), in this case of cases, is to draw them as points directly from the linelist\_sf object which we created in the preparation section.

The package **tmap** offers simple mapping capabilities for both static (“plot” mode) and interactive (“view” mode) with just a few lines of code. The **tmap** syntax is similar to that of **ggplot2**, such that commands are added to each other with +. Read more detail in this [vignette](https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html).

1. Set the **tmap** mode. In this case we will use “plot” mode, which produces static outputs.

Below, the points are plotted alone.tm\_shape() is provided with the linelist\_sf objects. We then add points via tm\_dots(), specifying the size and color. Because linelist\_sf is an sf object, we have already designated the two columns that contain the lat/long coordinates and the coordinate reference system (CRS):

Alone, the points do not tell us much. So we should also map the administrative boundaries:

Again we use tm\_shape() (see [documentation](https://www.rdocumentation.org/packages/tmap/versions/3.3/topics/tm_shape)) but instead of providing the case points shapefile, we provide the administrative boundary shapefile (polygons).

With the bbox = argument (bbox stands for “bounding box”) we can specify the coordinate boundaries. First we show the map display without bbox, and then with it.

And now both points and polygons together:

To read a good comparison of mapping options in R, see this [blog post](https://rstudio-pubs-static.s3.amazonaws.com/324400_69a673183ba449e9af4011b1eeb456b9.html).

## Spatial joins

You may be familiar with joining data from one dataset to another one. Several methods are discussed in the [Joining data](#joining-data) page of this handbook. A spatial join serves a similar purpose but leverages spatial relationships. Instead of relying on common values in columns to correctly match observations, you can utilize their spatial relationships, such as one feature being within another, or the nearest neighbor to another, or within a buffer of a certain radius from another, etc.

The **sf** package offers various methods for spatial joins. See more documentation about the st\_join method and spatial join types in this [reference](https://r-spatial.github.io/sf/reference/geos_binary_pred.html).

### Points in polygon

**Spatial assign administrative units to cases**

Here is an interesting conundrum: the case linelist does not contain any information about the administrative units of the cases. Although it is ideal to collect such information during the initial data collection phase, we can also assign administrative units to individual cases based on their spatial relationships (i.e. point intersects with a polygon).

Below, we will spatially intersect our case locations (points) with the ADM3 boundaries (polygons):

1. Begin with the linelist (points)
2. Spatial join to the boundaries, setting the type of join at “st\_intersects”
3. Use select() to keep only certain of the new administrative boundary columns

All the columns from sle\_adms have been added to the linelist! Each case now has columns detailing the administrative levels that it falls within. In this example, we only want to keep two of the new columns (admin level 3), so we select() the old column names and just the two additional of interest:

Below, just for display purposes you can see the first ten cases and that their admin level 3 (ADM3) jurisdictions that have been attached, based on where the point spatially intersected with the polygon shapes.

Now we can describe our cases by administrative unit - something we were not able to do before the spatial join!

We can also create a bar plot of case counts by administrative unit.

In this example, we begin the ggplot() with the linelist\_adm, so that we can apply factor functions like fct\_infreq() which orders the bars by frequency (see page on [Factors](#factors) for tips).

### Nearest neighbor

**Finding the nearest health facility / catchment area**

It might be useful to know where the health facilities are located in relation to the disease hot spots.

We can use the st\_nearest\_feature join method from the st\_join() function (**sf** package) to visualize the closest health facility to individual cases.

1. We begin with the shapefile linelist linelist\_sf
2. We spatially join with sle\_hf, which is the locations of health facilities and clinics (points)

We can see below (first 50 rows) that the each case now has data on the nearest clinic/hospital

We can see that “Den Clinic” is the closest health facility for about ~30% of the cases.

To visualize the results, we can use **tmap** - this time interactive mode for easier viewing

### Buffers

We can also explore how many cases are located within 2.5km (~30 mins) walking distance from the closest health facility.

Note: For more accurate distance calculations, it is better to re-project your sf object to the respective local map projection system such as UTM (Earth projected onto a planar surface). In this example, for simplicity we will stick to the World Geodetic System (WGS84) Geograhpic coordinate system (Earth represented in a spherical / round surface, therefore the units are in decimal degrees). We will use a general conversion of: 1 decimal degree = ~111km.

See more information about map projections and coordinate systems at this [esri article](https://www.esri.com/arcgis-blog/products/arcgis-pro/mapping/gcs_vs_pcs/). This [blog](http://www.geo.hunter.cuny.edu/~jochen/gtech201/lectures/lec6concepts/map%20coordinate%20systems/how%20to%20choose%20a%20projection.htm) talks about different types of map projection and how one can choose a suitable projection depending on the area of interest and the context of your map / analysis.

**First**, create a circular buffer with a radius of ~2.5km around each health facility. This is done with the function st\_buffer() from **tmap**. Because the unit of the map is in lat/long decimal degrees, that is how “0.02” is interpreted. If your map coordinate system is in meters, the number must be provided in meters.

Below we plot the buffer zones themselves, with the :

\*\*Second, we intersect these buffers with the cases (points) using *st\_join()* and the join type of st\_intersects\*. That is, the data from the buffers are joined to the points that they intersect with.

Now we can count the results: nrow(linelist\_sf\_hf\_2k[is.na(linelist\_sf\_hf\_2k$osm\_id.y),]) out of 1000 cases did not intersect with any buffer (that value is missing), and so live more than 30 mins walk from the nearest health facility.

We can visualize the results such that cases that did not intersect with any buffer appear in red.

### Other spatial joins

Alternative values for argument join include (from the [documentation](https://r-spatial.github.io/sf/reference/st_join.html))

* st\_contains\_properly
* st\_contains
* st\_covered\_by
* st\_covers
* st\_crosses
* st\_disjoint
* st\_equals\_exact
* st\_equals
* st\_is\_within\_distance
* st\_nearest\_feature
* st\_overlaps
* st\_touches
* st\_within

## Choropleth maps

Choropleth maps can be useful to visualize your data by pre-defined area, usually administrative unit or health area. In outbreak response this can help to target resource allocation for specific areas with high incidence rates, for example.

Now that we have the administrative unit names assigned to all cases (see section on spatial joins, above), we can start mapping the case counts by area (choropleth maps).

Since we also have population data by ADM3, we can add this information to the case\_adm3 table created previously.

We begin with the dataframe created in the previous step case\_adm3, which is a summary table of each administrative unit and its number of cases.

1. The population data sle\_adm3\_pop are joined using a left\_join() from **dplyr** on the basis of common values across column admin3pcod in the case\_adm3 dataframe, and column adm\_pcode in the sle\_adm3\_pop dataframe. See page on [Joining data](#joining-data)).
2. select() is applied to the new dataframe, to keep only the useful columns - total is total population
3. Cases per 10,000 populaton is calculated as a new column with mutate()

Join this table with the ADM3 polygons shapefile for mapping

Mapping the results

We can also map the incidence rates

## Mapping with ggplot2

If you are already familiar with using **ggplot2**, you can use that package instead to create static maps of your data. The geom\_sf() function will draw different objects based on which features (points, lines, or polygons) are in your data. For example, you can use geom\_sf() in a ggplot() using sf data with polygon geometry to create a choropleth map.

To illustrate how this works, we can start with the ADM3 polygons shapefile that we used earlier. Recall that these are Admin Level 3 regions in Sierra Leone:

We can use the left\_join() function from **dplyr** to add the data we would like to map to the shapefile object. In this case, we are going to use the case\_adm3 data frame that we created earlier to summarize case counts by administrative region; however, we can use this same approach to map any data stored in a data frame.

To make a column chart of case counts by region, using **ggplot2**, we could then call geom\_col() as follows:

If we want to use **ggplot2** to instead make a choropleth map of case counts, we can use similar syntax to call the geom\_sf() function:

We can then customize the appearance of our map using grammar that is consistent across **ggplot2**, for example:

For R users who are comfortable working with **ggplot2**, geom\_sf() offers a simple and direct implementation that is suitable for basic map visualizations. To learn more, read the [geom\_sf() vignette](https://ggplot2.tidyverse.org/reference/ggsf.html) or the [ggplot2 book](https://ggplot2-book.org/maps.html).

## Basemaps

### OpenStreetMap

Below we describe how to achieve a basemap for a **ggplot2** map using OpenStreetMap features. Alternative methods include using **ggmap** which requires free registration with Google ([details](https://www.earthdatascience.org/courses/earth-analytics/lidar-raster-data-r/ggmap-basemap/)).

[**OpenStreetMap**](https://en.wikipedia.org/wiki/OpenStreetMap) is a collaborative project to create a free editable map of the world. The underlying geolocation data (e.g. locations of cities, roads, natural features, airports, schools, hospitals, roads etc) are considered the primary output of the project.

First we load the **OpenStreetMap** package, from which we will get our basemap.

Then, we create the object map, which we define using the function openmap() from **OpenStreetMap** package ([documentation](https://www.rdocumentation.org/packages/OpenStreetMap/versions/0.3.4/topics/openmap)). We provide the following:

* upperLeft and lowerRight Two coordinate pairs specifying the limits of the basemap tile
  + In this case we’ve put in the max and min from the linelist rows, so the map will respond dynamically to the data
* zoom = (if null it is determined automatically)
* type = which type of basemap - we have listed several possibilities here and the code is currently using the first one ([1]) “osm”
* mergeTiles = we chose TRUE so the basetiles are all merged into one

If we plot this basemap right now, using autoplot.OpenStreetMap() from **OpenStreetMap** package, you see that the units on the axes are not latitude/longitude coordinates. It is using a different coordinate system. To correctly display the case residences (which are stored in lat/long), this must be changed.

Thus, we want to convert the map to latitude/longitude with the openproj() function from **OpenStreetMap** package. We provide the basemap map and also provide the Coordinate Reference System (CRS) we want. We do this by providing the “proj.4” character string for the WGS 1984 projection, but you can provide the CRS in other ways as well. (see [this page](https://www.earthdatascience.org/courses/earth-analytics/spatial-data-r/understand-epsg-wkt-and-other-crs-definition-file-types/) to better understand what a proj.4 string is)

Now when we create the plot we see that along the axes are latitude and longitude coordinate. The coordinate system has been converted. Now our cases will plot correctly if overlaid!

See the tutorials [here](http://data-analytics.net/cep/Schedule_files/geospatial.html) and [here](https://www.rdocumentation.org/packages/OpenStreetMap/versions/0.3.4/topics/autoplot.OpenStreetMap) for more info.

## Contoured density heatmaps

Below we describe how to achieve a contoured density heatmap of cases, over a basemap, beginning with a linelist (one row per case).

1. Create basemap tile from OpenStreetMap, as described above
2. Plot the cases from linelist using the latitude and longitude columns
3. Convert the points to a density heatmap with stat\_density\_2d() from **ggplot2**,

When we have a basemap with lat/long coordinates, we can plot our cases on top using the lat/long coordinates of their residence.

Building on the function autoplot.OpenStreetMap() to create the basemap, **ggplot2** functions will easily add on top, as shown with geom\_point() below:

The map above might be difficult to interpret, especially with the points overlapping. So you can instead plot a 2d density map using the **ggplot2** function stat\_density\_2d(). You are still using the linelist lat/lon coordinates, but a 2D kernel density estimation is performed and the results are displayed with contour lines - like a topographical map. Read the full [documentation here](https://ggplot2.tidyverse.org/reference/geom_density_2d.html).

### Time series heatmap

The density heatmap above shows cumulative cases. We can examine the outbreak over time and space by faceting the heatmap based on the month of symptom onset, as derived from the linelist.

We begin in the linelist, creating a new column with the Year and Month of onset. The format() function from **base** R changes how a date is displayed. In this case we want “YYYY-MM”.

Now, we simply introduce facetting via **ggplot2** to the density heatmap. facet\_wrap() is applied, using the new column as rows. We set the number of facet columns to 3 for clarity.

## Spatial statistics

Most of our discussion so far has focused on visualization of spatial data. In some cases, you may also be interested in using spatial statistics to quantify the spatial relationships of attributes in your data. This section will provide a very brief overview of some key concepts in spatial statistics, and suggest some resources that will be helpful to explore if you wish to do more comprehensive spatial analyses.

### Spatial relationships

Before we can calculate any spatial statistics, we need to specify the relationships between features in our data. There are many ways to conceptualize spatial relationships, but a simple and commonly-applicable model to use is that of adjacency - specifically, that we expect a geographic relationship between areas that share a border or “neighbour” one another.

We can quantify adjacency relationships between administrative region polygons in the sle\_adm3 data we have been using with the **spdep** package. We will specify queen contiguity, which means that regions will be neighbors if they share at least one point along their borders. The alternative would be rook contiguity, which requires that regions share an edge - in our case, with irregular polygons, the distinction is trivial, but in some cases the choice between queen and rook can be influential.

The matrix printed above shows the relationships between the 9 regions in our sle\_adm3 data. A score of 0 indicates two regions are not neighbors, while any value other than 0 indicates a neighbor relationship. The values in the matrix are scaled so that each region has a total row weight of 1.

A better way to visualize these neighbor relationships is by plotting them:

We have used an adjacency approach to identify neighboring polygons; the neighbors we identified are also sometimes called **contiguity-based neighbors**. But this is just one way of choosing which regions are expected to have a geographic relationship. The most common alternative approaches for identifying geographic relationships generate **distance-based neighbors**; briefly, these are:

* **K-nearest neighbors** - Based on the distance between centroids (the geographically-weighted center of each polygon region), select the n closest regions as neighbors. A maximum-distance proximity threshold may also be specified. In **spdep**, you can use knearneigh() (see [documentation](https://r-spatial.github.io/spdep/reference/knearneigh.html)).
* **Distance threshold neighbors** - Select all neighbors within a distance threshold. In **spdep**, these neighbor relationships can be identified using dnearneigh() (see [documentation](https://www.rdocumentation.org/packages/spdep/versions/1.1-7/topics/dnearneigh)).

### Spatial autocorrelation

Tobler’s oft-cited first law of geography states that “everything is related to everything else, but near things are more related than distant things.” In epidemiology, this often means that risk of a particular health outcome in a given region is more similar to its neighboring regions than to those far away. This concept has been formalized as **spatial autocorrelation** - the statistical property that geographic features with similar values are clustered together in space. Statistical measures of spatial autocorrelation can be used to quantify the extent of spatial clustering in your data, locate where clustering occurs, and identify shared patterns of spatial autocorrelation between distinct variables in your data. This section gives an overview of some common measures of spatial autocorrelation and how to calculate them in R.

**Moran’s I** - This is a global summary statistic of the correlation between the value of a variable in one region, and the values of the same variable in neighboring regions. The Moran’s I statistic typically ranges from -1 to 1. A value of 0 indicates no pattern of spatial correlation, while values closer to 1 or -1 indicate stronger spatial autocorrelation (similar values close together) or spatial dispersion (dissimilar values close together), respectively.

For an example, we will calculate a Moran’s I statistic to quantify the spatial autocorrelation in Ebola cases we mapped earlier (remember, this is a subset of cases from the simulated epidemic linelist dataframe). The **spdep** package has a function, moran.test, that can do this calculation for us:

The output from the moran.test() function shows us a Moran I statistic of round(moran\_i$estimate[1],2). This indicates the presence of spatial autocorrelation in our data - specifically, that regions with similar numbers of Ebola cases are likely to be close together. The p-value provided by moran.test() is generated by comparison to the expectation under null hypothesis of no spatial autocorrelation, and can be used if you need to report the results of a formal hypothesis test.

**Local Moran’s I** - We can decompose the (global) Moran’s I statistic calculated above to identify localized spatial autocorrelation; that is, to identify specific clusters in our data. This statistic, which is sometimes called a **Local Indicator of Spatial Association (LISA)** statistic, summarizes the extent of spatial autocorrelation around each individual region. It can be useful for finding “hot” and “cold” spots on the map.

To show an example, we can calculate and map Local Moran’s I for the Ebola case counts used above, with the local\_moran() function from **spdep**:

**Getis-Ord Gi\*** - This is another statistic that is commonly used for hotspot analysis; in large part, the popularity of this statistic relates to its use in the Hot Spot Analysis tool in ArcGIS. It is based on the assumption that typically, the difference in a variable’s value between neighboring regions should follow a normal distribution. It uses a z-score approach to identify regions that have significantly higher (hot spot) or significantly lower (cold spot) values of a specified variable, compared to their neighbors.

We can calculate and map the Gi\* statistic using the localG() function from **spdep**:

As you can see, the map of Getis-Ord Gi\* looks slightly different from the map of Local Moran’s I produced earlier. This reflects that the method used to calculate these two statistics are slightly different; which one you should use depends on your specific use case and the research question of interest.

**Lee’s L test** - This is a statistical test for bivariate spatial correlation. It allows you to test whether the spatial pattern for a given variable x is similar to the spatial pattern of another variable, y, that is hypothesized to be related spatially to x.

To give an example, let’s test whether the spatial pattern of Ebola cases from the simulated epidemic is correlated with the spatial pattern of population. To start, we need to have a population variable in our sle\_adm3 data. We can use the total variable from the sle\_adm3\_pop dataframe that we loaded earlier.

We can quickly visualize the spatial patterns of the two variables side by side, to see whether they look similar:

Visually, the patterns seem dissimilar. We can use the lee.test() function in **spdep** to test statistically whether the pattern of spatial autocorrelation in the two variables is related. The L statistic will be close to 0 if there is no correlation between the patterns, close to 1 if there is a strong positive correlation (i.e. the patterns are similar), and close to -1 if there is a strong negative correlation (i.e. the patterns are inverse).

The output above shows that the Lee’s L statistic for our two variables was round(lee\_test$estimate[1],2), which indicates weak negative correlation. This confirms our visual assessment that the pattern of cases and population are not related to one another, and provides evidence that the spatial pattern of cases is not strictly a result of population density in high-risk areas.

The Lee L statistic can be useful for making these kinds of inferences about the relationship between spatially distributed variables; however, to describe the nature of the relationship between two variables in more detail, or adjust for confounding, spatial regression techniques will be needed. These are described briefly in the following section.

### Spatial regression

You may wish to make statistical inferences about the relationships between variables in your spatial data. In these cases, it is useful to consider spatial regression techniques - that is, approaches to regression that explicitly consider the spatial organization of units in your data. Some reasons that you may need to consider spatial regression models, rather than standard regression models such as GLMs, include:

* Standard regression models assume that residuals are independent from one another. In the presence of strong spatial autocorrelation, the residuals of a standard regression model are likely to be spatially autocorrelated as well, thus violating this assumption. This can lead to problems with interpreting the model results, in which case a spatial model would be preferred.
* Regression models also typically assume that the effect of a variable x is constant over all observations. In the case of spatial heterogeneity, the effects we wish to estimate may vary over space, and we may be interested in quantifying those differences. In this case, spatial regression models offer more flexibility for estimating and interpreting effects.

The details of spatial regression approaches are beyond the scope of this handbook. This section will instead provide an overview of the most common spatial regression models and their uses, and refer you to references that may of use if you wish to explore this area further.

**Spatial error models** - These models assume that the error terms across spatial units are correlated, in which case the data would violate the assumptions of a standard OLS model. Spatial error models are also sometimes referred to as **simultaneous autoregressive (SAR) models**. They can be fit using the errorsarlm() function in the **spatialreg** package (spatial regression functions which used to be a part of **spdep**).

**Spatial lag models** - These models assume that the dependent variable for a region i is influenced not only by value of independent variables in i, but also by the values of those variables in regions neighboring i. Like spatial error models, spatial lag models are also sometimes described as **simultaneous autoregressive (SAR) models**. They can be fit using the lagsarlm() function in the **spatialreg** package.

The **spdep** package contains several useful diagnostic tests for deciding between standard OLS, spatial lag, and spatial error models. These tests, called Lagrange Multiplier diagnostics, can be used to identify the type of spatial dependence in your data and choose which model is most appropriate. The function lm.LMtests() can be used to calculate all of the Lagrange Multiplier tests. Anselin (1988) also provides a useful flow chart tool to decide which spatial regression model to use based on the results of the Lagrange Multiplier tests:

**Bayesian hierarchical models** - Bayesian approaches are commonly used for some applications in spatial analysis, most commonly for [disease mapping](https://pubmed.ncbi.nlm.nih.gov/15690999/). They are preferred in cases where case data are sparsely distributed (for example, in the case of a rare outcome) or statistically “noisy”, as they can be used to generate “smoothed” estimates of disease risk by accounting for the underlying latent spatial process. This may improve the quality of estimates. They also allow investigator pre-specification (via choice of prior) of complex spatial correlation patterns that may exist in the data, which can account for spatially-dependent and -independent variation in both independent and dependent variables. In R, Bayesian hierarchical models can be fit using the **CARbayes** package (see [vignette](https://cran.r-project.org/web/packages/CARBayes/vignettes/CARBayes.pdf)) or R-INLA (see [website](https://www.r-inla.org/home) and [textbook](https://becarioprecario.bitbucket.io/inla-gitbook/)). R can also be used to call external software that does Bayesian estimation, such as JAGS or WinBUGS.

## Resources

* R Simple Features and sf package [vignette](https://cran.r-project.org/web/packages/sf/vignettes/sf1.html)
* R tmap package [vignette](https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html)
* ggmap: [Spatial Visualization with ggplot2](https://journal.r-project.org/archive/2013-1/kahle-wickham.pdf)
* [Intro to making maps with R, overview of different packages](https://bookdown.org/nicohahn/making_maps_with_r5/docs/introduction.html)
* Spatial Data in R [(EarthLab course)](https://www.earthdatascience.org/courses/earth-analytics/spatial-data-r/)
* Applied Spatial Data Analysis in R [textbook](https://link.springer.com/book/10.1007/978-1-4614-7618-4)
* **SpatialEpiApp** - a [Shiny app that is downloadable as an R package](https://github.com/Paula-Moraga/SpatialEpiApp), allowing you to provide your own data and conduct mapping, cluster analysis, and spatial statistics.
* An Introduction to Spatial Econometrics in R [workshop](http://www.econ.uiuc.edu/~lab/workshop/Spatial_in_R.html)

# V Data Visualization

# Tables for presentation

This page demonstrates how to convert summary data frames into presentation-ready tables with the **flextable** package. These tables can be inserted into powerpoint slides, HTML pages, PDF or Word documents, etc.

Understand that before using **flextable**, you must create the summary table as a data frame. Use methods from the [Descriptive tables](#descriptive-tables) and [Pivoting data](#pivoting-data) pages such as tabulations, cross-tabulations, pivoting, and calculating descriptive statistics. The resulting data frame can then be passed to **flextable** for display formatting.

There are many other R packages that can be used to craft tables for presentation - we chose to highlight **flextable** in this page. An example using the **knitr** package and its kable() function can be found in the [Contact Tracing](#contact-tracing-1) page. Likewise, the **DT** package is highlighted in the page [Dashboards with Shiny](#dashboards-with-shiny). Others such as **GT** and **huxtable** are mentione in the [Suggested packages](#suggested-packages-1) page.

## Preparation

### Load packages

Install and load **flextable**. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

To begin, we import the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

### Prepare table

Before beginning to use **flextable** you will need to create your table as a data frame. See the page on [Descriptive tables](#descriptive-tables) and [Pivoting data](#pivoting-data) to learn how to create a data frame using packages such as **janitor** and **dplyr**. You must arrange the content in rows and columns as you want it displayed. Then, the data frame will be passed to **flextable** to display it with colors, headers, fonts, etc.

Below is an example from the [Descriptive tables](#descriptive-tables) page of converting the case linelist into a data frame that summarises patient outcomes and CT values by hospital, with a Totals row at the bottom. The output is saved as table.

## Basic flextable

### Create a flextable

To create and manage **flextable** objects, we first pass the data frame through the flextable() function. We save the result as my\_table.

After doing this, we can progressively pipe the my\_table object through more **flextable** formatting functions.

In this page for sake of clarity we will save the table at intermediate steps as my\_table, adding **flextable** functions bit-by-bit. If you want to see all the code from beginning to end written in one chunk, visit the [All code together](#tbl_pres_all) section below.

The general syntax of each line of **flextable** code is as follows:

* function(table, i = X, j = X, part = "X"), where:
  + The ‘function’ can be one of many different functions, such as width() to determine column widths, bg() to set background colours, align() to set whether text is centre/right/left aligned, and so on.
  + table = is the name of the data frame, although does not need to be stated if the data frame is piped into the function.
  + part = refers to which part of the table the function is being applied to. E.g. “header”, “body” or “all”.
  + i = specifies the row to apply the function to, where ‘X’ is the row number. If multiple rows, e.g. the first to third rows, one can specify: i = c(1:3). Note if ‘body’ is selected, the first row starts from underneath the header section.
  + j = specifies the column to apply the function to, where ‘x’ is the column number or name. If multiple columns, e.g. the fifth and sixth, one can specify: j = c(5,6).

You can find the complete list of **flextable** formatting function [here](https://davidgohel.github.io/flextable/reference/index.html) or review the documentation by entering ?flextable.

### Column width

We can use the autofit() function, which nicely stretches out the table so that each cell only has one row of text. The function qflextable() is a convenient shorthand for flextable() and autofit().

However, this might not always be appropriate, especially if there are very long values within cells, meaning the table might not fit on the page.

Instead, we can specify widths with the width() function. It can take some playing around to know what width value to put. In the example below, we specify different widths for column 1, column 2, and columns 4 to 8.

### Column headers

We want more clearer headers for easier interpretation of the table contents.

For this table, we will want to add a second header layer so that columns covering the same subgroups can be grouped together. We do this with the add\_header\_row() function with top = TRUE. We provide the new name of each column to values =, leaving empty values "" for columns we know we will merge together later.

We also rename the header names in the now-second header in a separate set\_header\_labels() command.

Finally, to “combine” certain column headers in the top header we use merge\_at() to merge the column headers in the top header row.

### Borders and background

You can adjust the borders, internal lines, etc. with various **flextable** functions. It is often easier to start by removing all existing borders with border\_remove().

Then, you can apply default border themes by passing the table to theme\_box(), theme\_booktabs(), or theme\_alafoli().

You can add vertical and horizontal lines with a variety of functions. hline() and vline() add lines to a specified row or column, respectively. Within each, you must specify the part = as either “all”, “body”, or “header”. For vertical lines, specify the column to j =, and for horizontal lines the row to i =. Other functions like vline\_right(), vline\_left(), hline\_top(), and hline\_bottom() add lines to the outsides only.

In all of these functions, the actual line style itself must be specified to border = and must be the output of a separate command using the fp\_border() function from the **officer** package. This function helps you define the width and color of the line. You can define this above the table commands, as shown below.

### Font and alignment

We centre-align all columns aside from the left-most column with the hospital names, using the align() function from **flextable**.

Additionally, we can increase the header font size and change then to bold. We can also change the total row to bold.

We can ensure that the proportion columns display only one decimal place using the function colformat\_num(). Note this could also have been done at data management stage with the round() function.

### Merge cells

Just as we merge cells horizontally in the header row, we can also merge cells vertically using merge\_at() and specifying the rows (i) and column (j). Here we merge the “Hospital” and “Total cases with known outcome” values vertically to give them more space.

### Background color

To distinguish the content of the table from the headers, we may want to add additional formatting. e.g. changing the background color. In this example we change the table body to gray.

## Conditional formatting

We can highlight all values in a column that meet a certain rule, e.g. where more than 55% of cases died. Simply put the criteria to the i = or j = argument, preceded by a tilde ~. Reference the column in the data frame, not the display heading values.

Or, we can highlight the entire row meeting a certain criterion, such as a hospital of interest. To do this we just remove the column (j) specification so the criteria apply to all columns.

## All code together

Below we show all the code from the above sections together.

## Saving your table

There are different ways the table can be integrated into your output.

### Save single table

You can export the tables to Word, PowerPoint or HTML or as an image (PNG) files. To do this, use one of the following functions:

* save\_as\_docx()
* save\_as\_pptx()
* save\_as\_image()
* save\_as\_html()

For instance below we save our table as a word document. Note the syntax of the first argument - you can just provide the name of your flextable object e.g. my\_table, or you can give is a “name” as shown below (the name is “my table”). If name, this will appear as the title of the table in Word. We also demonstrate code to save as PNG image.

Note the packages webshot or webshot2 are required to save a flextable as an image. Images may come out with transparent backgrounds.

If you want to view a ‘live’ version of the **flextable** output in the intended document format, use print() and specify one of the below to preview =. The document will “pop-up” open on your computer in the specified software program, but will not be saved. This can be useful to check if the table fits in one page/slide or so you can quickly copy it into another document, you can use the print method with the argument preview set to “pptx” or “docx”.

### Print table in R markdown

This table can be integrated into your an automated document, an R markdown output, if the table object is called within the R markdown chunk. This means the table can be updated as part of a report where the data might change, so the numbers can be refreshed.

See detail in the [Reports with R Markdown](#reports-with-r-markdown) page of this handbook.

## Resources

The full **flextable** book is here: <https://ardata-fr.github.io/flextable-book/> The Github site is [here](https://davidgohel.github.io/flextable/)  
A manual of all the **flextable** functions can be found [here](https://davidgohel.github.io/flextable/reference/index.html)

A gallery of beautiful example **flextable** tables with code can be accessed [here](https://ardata-fr.github.io/flextable-gallery/gallery/)

# ggplot basics

**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 syntax is significantly different from **base** R plotting, and has a learning curve associated with it. Using **ggplot2** generally requires the user to format their data in a way that is highly **tidyverse** compatible, which ultimately makes using these packages together very effective.

In this page we will cover the fundamentals of plotting with **ggplot2**. See the page [ggplot tips](#ggplot-tips) for suggestions and advanced techniques to make your plots really look nice.

There are several extensive **ggplot2** tutorials linked in the resources section. You can also download this [data visualization with ggplot cheatsheet](https://github.com/rstudio/cheatsheets/raw/master/data-visualization-2.1.pdf) from the RStudio website. If you want inspiration for ways to creatively visualise your data, we suggest reviewing websites like the [R graph gallery](https://www.r-graph-gallery.com/) and [Data-to-viz](https://www.data-to-viz.com/caveats.html).

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

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).

### General cleaning

When preparing data to plot, it is best to make the data adhere to [“tidy” data standards](https://r4ds.had.co.nz/tidy-data.html) as much as possible. How to achieve this is expanded on in the data management pages of this handbook, such as [Cleaning data and core functions](#cleaning-data-and-core-functions).

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 (see [Characters and strings](#characters-and-strings))

Here are some examples of this in action:

### 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](#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.

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

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:

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.

## 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.

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

A simple example of skeleton code is as follows. We will explain each component in the sections below.

## 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.

## 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](https://exts.ggplot2.tidyverse.org/gallery/). Some common geoms are listed below:

* Histograms - geom\_histogram()
* Bar charts - geom\_bar() or geom\_col() (see [“Bar plot” section](#ggplot_basics_bars))
* 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.

## 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.

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.

### 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](#ggplot_basics_themes)).

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.

### 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](#ggplotgroups) 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](##ggplot_basics_map_loc)).

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.

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.

### 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:

### 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:

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](#ggplot-tips) page or the page on [Factors](#factors). There are many examples of grouped plots in the sections below on plotting continuous and categorical data.

## 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](#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.

1. 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 (see [Download handbook and data](#download-handbook-and-data)) which consists of daily case counts of malaria for facilities, by age group.

Below we import and do some quick modifications for simplicity:

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).

### 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](#ggplot_basics_bars)).

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.

### 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](#pivoting-data) for more information on this process.

Now the first 50 rows of data look like this:

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.

### 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.

### Factor level order in facets

See this [post](https://juliasilge.com/blog/reorder-within/) on how to re-order factor levels within facets.

## Storing plots

### Saving plots

By default when you run a ggplot() command, the plot will be printed to the Plots RStudio pane. However, you can also save the plot as an object by using the assignment operator <- and giving it a name. Then it will not print unless the object name itself is run. You can also print it by wrapping the plot name with print(), but this is only necessary in certain circumstances such as if the plot is created inside a for loop used to print multiple plots at once (see [Iteration, loops, and lists](#iteration-loops-and-lists) page).

### Modifying saved plots

One nice thing about **ggplot2** is that you can define a plot (as above), and then add layers to it starting with its name. You do not have to repeat all the commands that created the original plot!

For example, to modify the plot age\_by\_wt that was defined above, to include a vertical line at age 50, we would just add a + and begin adding additional layers to the plot.

### 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](https://ggplot2.tidyverse.org/reference/ggsave.html).

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

## 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:

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. Read more about this in the page on [Characters and strings](#characters-and-strings).

A note on specifying the legend title: There is no one “legend title” argument, as you could have multiple scales in your legend. Within labs(), you can write the argument for the plot aesthetic used to create the legend, and provide the title this way. For example, above we assigned color = age to create the legend. Therefore, we provide color = to labs() and assign the legend title desired (“Age” with capital A). If you create the legend with aes(fill = COLUMN), then in labs() you would write fill = to adjust the title of that legend. The section on color scales in the [ggplot tips](#ggplot-tips) page provides more details on editing legends, and an alternative approach using scales\_() functions.

## 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](https://ggplot2.tidyverse.org/reference/ggtheme.html) 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.

### 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

1. 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")

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 |
| 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](https://ggplot2.tidyverse.org/reference/theme.html), 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".

## Colors

Please see this [section on color scales of the ggplot tips page](#ggplot_tips_colors).

## Piping into ****ggplot2****

When using pipes to clean and transform your data, it is easy to pass the transformed data into ggplot().

The pipes that pass the dataset from function-to-function will transition to + once the ggplot() function is called. Note that in this case, there is no need to specify the data = argument, as this is automatically defined as the piped-in dataset.

This is how that might look:

## 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](https://www.data-to-viz.com/caveat/boxplot.html)).
  + **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](#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](#ggplot_basics_bars) 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:

To get smoothed proportions, you can use geom\_density():

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):

If you want to have some fun, try geom\_density\_ridges from the **ggridges** package ([vignette here](https://cran.r-project.org/web/packages/ggridges/vignettes/introduction.html).

Read more in detail about histograms at the **tidyverse** [page on geom\_histogram()](https://ggplot2.tidyverse.org/reference/geom_histogram.html).

### 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](https://www.r-graph-gallery.com/boxplot.html) and this [data-to-viz article](https://www.data-to-viz.com/caveat/boxplot.html) 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.

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

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.

### 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.

### 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](#ggplot-tips) page and the page on on [Heat plots](#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.

## Plot categorical data

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

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](#factors) page.

### 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.

### 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:

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.

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](#grouping-data) page for grouping tips).

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.

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.

### geom\_histogram()

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(). There are arguments specific to histograms such as bin\_width = and breaks = to specify how the data should be binned. The section above on continuous data and the page on [Epidemic curves](#epidemic-curves) provide additional detail.

## Resources

There is a huge amount of help online, especially with ggplot. See:

* [ggplot2 cheat sheet](http://r-statistics.co/ggplot2-cheatsheet.html)
* [another cheat sheet](https://biostats.w.uib.no/the-ggplot2-cheat-sheet-by-rstudio/)
* [tidyverse ggplot basics page](https://ggplot2.tidyverse.org/reference/)
* [plotting continuous variables](http://www.sthda.com/english/articles/32-r-graphics-essentials/131-plot-two-continuous-variables-scatter-graph-and-alternatives/)
* R for Data Science pages on [data visualization](https://r4ds.had.co.nz/data-visualisation.html)
* [graphics for communicaton](https://r4ds.had.co.nz/graphics-for-communication.html)

# ggplot tips

In this page we will cover tips and tricks to make your ggplots sharp and fancy. See the page on [ggplot basics](#ggplot-basics) for the fundamentals.

There a several extensive [**ggplot2** tutorials](https://ggplot2.tidyverse.org/) linked in the Resources section. You can also download this [data visualization with ggplot cheatsheet](https://rstudio.com/resources/cheatsheets/) from the RStudio website. We strongly recommend that you peruse for inspiration at the [R graph gallery](https://www.r-graph-gallery.com/) and [Data-to-viz](https://www.data-to-viz.com/caveats.html).

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

For this page, we import the dataset of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## 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.

### 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](#ggplot_basics_themes) section of the [ggplot basics](#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](http://sape.inf.usi.ch/quick-reference/ggplot2/colour) or enter ?colors), or a specific hex colour such as "#ff0505".

As explained the [ggplot basics](#ggplot-basics) section on [mapping data to the plot](#ggplot_basics_mapping), 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:

### 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](#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).

### 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 =.

#### 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.

#### 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.

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.

### 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](#contact-tracing-1) page that contains the ages of cases and of their source cases.

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](https://ggplot2.tidyverse.org/reference/geom_density_2d.html). Note how the fill scale is continuous.

Now we show some variations on the fill scale:

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()](https://www.rdocumentation.org/packages/scales/versions/0.4.1/topics/rescale) to adjust how colors are positioned along the gradient; it rescales your vector of positions to be between 0 and 1.

### 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](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\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](https://ggplot2.tidyverse.org/reference/scale_viridis.html) and [here](https://bids.github.io/colormap/)). 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](https://www.color-blindness.com/coblis-color-blindness-simulator/). If you have a red/green color scheme, try a “hot-cold” (red-blue) scheme instead as described [here](https://www.visualisingdata.com/2019/08/five-ways-to-design-for-red-green-colour-blindness/#:~:text=The%20pink%2Dred%20through%20to,green%20hues%20used%20by%20default.)

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

## 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.

#### ****ggthemr****

Also consider using the **ggthemr** package. You can download this package from Github using the instructions [here](https://github.com/Mikata-Project/ggthemr). It offers palettes that are very aesthetically pleasing, but be aware that these typically have a maximum number of values that can be limiting if you want more than 7 or 8 colors.

## Contour lines

Contour plots are helpful when you have many points that might cover each other (“overplotting”). 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](https://ggplot2.tidyverse.org/reference/geom_density_2d.html).

## 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](https://cran.r-project.org/web/packages/ggExtra/vignettes/ggExtra.html), in the [R Graph Gallery](https://www.r-graph-gallery.com/277-marginal-histogram-for-ggplot2.html) or the function R documentation ?ggMarginal.

To add marginal histograms use type = "histogram". You can optionally set groupFill = TRUE to get stacked histograms.

Marginal density plot with grouped/colored values:

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:

## 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](https://ggrepel.slowkow.com/articles/examples.html) for more.

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.

## 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](#working-with-dates-1) page for more information on this, or [Epidemic curves](#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"):

## 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.

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).

## 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)

## 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](https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html). You can read the more extensive documentation for each function online [here](https://www.rdocumentation.org/packages/cowplot/versions/1.1.1). 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.

While faceting (described in the [ggplot basics](#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.

### 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):

Here is how the two plots look when combined using plot\_grid() without combining their legends:

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!

This solution was learned from [this post](https://stackoverflow.com/questions/52060601/ggplot-multiple-legends-arrangement) with a minor fix to align legends from [this post](https://github.com/wilkelab/cowplot/issues/33).

**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.

This technique is explained more in these two vignettes:

[Wilke lab](https://wilkelab.org/cowplot/articles/drawing_with_on_plots.html)  
[draw\_plot() documentation](https://www.rdocumentation.org/packages/cowplot/versions/1.1.1/topics/draw_plot)

## Dual axes

A secondary y-axis is often a requested addition to a ggplot2 graph. While there is a robust debate about the validity of such graphs in the data visualization community, and they are often not recommended, your manager may still want them. Below, we present one method to achieve them: using the **cowplot** package to combine two separate plots.

This approach involves creating two separate plots - one with a y-axis on the left, and the other with y-axis on the right. Both will use a specific theme\_cowplot() and must have the same x-axis. Then in a third command the two plots are aligned and overlaid on top of each other. The functionalities of **cowplot**, of which this is only one, are described in depth at this [site](https://wilkelab.org/cowplot/articles/aligning_plots.html).

To demonstrate this technique we will overlay the epidemic curve with a line of the weekly percent of patients who died. We use this example because the alignment of dates on the x-axis is more complex than say, aligning a bar chart with another plot. Some things to note:

* The epicurve and the line are aggregated into weeks prior to plotting and the date\_breaks and date\_labels are identical - we do this so that the x-axes of the two plots are the same when they are overlaid.
* The y-axis is moved to the right-side for plot 2 with the position = argument of scale\_y\_continuous().
* Both plots make use of theme\_cowplot()

Note there is another example of this technique in the [Epidemic curves](#epidemic-curves) page - overlaying cumulative incidence on top of the epicurve.

**Make plot 1**  
This is essentially the epicurve. We use geom\_area() just to demonstrate its use (area under a line, by default)

**Make plot 2**  
Create the second plot showing a line of the weekly percent of cases who died.

Now we align the plot using the function align\_plots(), specifying horizontal and vertical alignment (“hv”, could also be “h”, “v”, “none”). We specify alignment of all axes as well (top, bottom, left, and right) with “tblr”. The output is of class list (2 elements).

Then we draw the two plots together using ggdraw() (from **cowplot**) and referencing the two parts of the aligned\_plots object.

## Packages to help you

There are some really neat R packages specifically designed to help you navigate **ggplot2**:

### Point-and-click ****ggplot2**** with ****equisse****

“This addin allows you to interactively explore your data by visualizing it with the ggplot2 package. It allows you to draw bar plots, curves, scatter plots, histograms, boxplot and sf objects, then export the graph or retrieve the code to reproduce the graph.”

Install and then launch the addin via the RStudio menu or with esquisse::esquisser().

See the [Github page](https://github.com/dreamRs/esquisse)

[Documentation](https://dreamrs.github.io/esquisse/index.html)

## Miscellaneous

### Numeric display

You can disable scientific notation by running this command prior to plotting.

Or apply number\_format() from the **scales** package to a specific value or column, as shown below.

Use functions from the package **scales** to easily adjust how numbers are displayed. These can be applied to columns in your data frame, but are shown on individual numbers for purpose of example.

## Resources

Inspiration [ggplot graph gallery](https://www.tidyverse.org/blog/2018/07/ggplot2-3-0-0/)

Presentation of data European Centre for Disease Prevention and Control [Guidelines of presentation of surveillance data](https://ecdc.europa.eu/sites/portal/files/documents/Guidelines%20for%20presentation%20of%20surveillance%20data-final-with-cover-for-we....pdf)

Facets and labellers [Using labellers for facet strips](http://www.cookbook-r.com/Graphs/Facets_(ggplot2)/#modifying-facet-label-text) [Labellers](https://ggplot2.tidyverse.org/reference/labellers.html)

Adjusting order with factors [fct\_reorder](https://forcats.tidyverse.org/reference/fct_reorder.html)  
[fct\_inorder](https://forcats.tidyverse.org/reference/fct_inorder.html)  
[How to reorder a boxplot](https://cmdlinetips.com/2019/02/how-to-reorder-a-boxplot-in-r/)  
[Reorder a variable in ggplot2](https://www.r-graph-gallery.com/267-reorder-a-variable-in-ggplot2.html)  
[R for Data Science - Factors](https://r4ds.had.co.nz/factors.html)

Legends  
[Adjust legend order](https://stackoverflow.com/questions/38425908/reverse-stacking-order-without-affecting-legend-order-in-ggplot2-bar-charts)

Captions [Caption alignment](https://stackoverflow.com/questions/64701500/left-align-ggplot-caption)

Labels  
[ggrepel](https://ggrepel.slowkow.com/articles/examples.html)

Cheatsheets  
[Beautiful plotting with ggplot2](http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/)

# Epidemic curves

An epidemic curve (also known as an “epi curve”) is a core epidemiological chart typically used to visualize the temporal pattern of illness onset among a cluster or epidemic of cases.

Analysis of the epicurve can reveal temporal trends, outliers, the magnitude of the outbreak, the most likely time period of exposure, time intervals between case generations, and can even help identify the mode of transmission of an unidentified disease (e.g. point source, continuous common source, person-to-person propagation). One online lesson on interpretation of epi curves can be found at the website of the [US CDC](https://www.cdc.gov/training/quicklearns/epimode/index.html).

In this page we demonstrate two approaches to producing epicurves in R:

* The **incidence2** package, which can produce an epi curve with simple commands
* The **ggplot2** package, which allows for advanced customizability via more complex commands

Also addressed are specific use-cases such as:

* Plotting aggregated count data
* Faceting or producing small-multiples
* Applying moving averages
* Showing which data are “tentative” or subject to reporting delays
* Overlaying cumulative case incidence using a second axis

## Preparation

### Packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

Two example datasets are used in this section:

* Linelist of individual cases from a simulated epidemic
* Aggregated counts by hospital from the same simulated epidemic

The datasets are imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data.

**Case linelist**

We import the dataset of cases from a simulated Ebola epidemic. If you want to download the data to follow step-by-step, see instruction in the [Download handbook and data](#download-handbook-and-data) page. We assume the file is in the working directory so no sub-folders are specified in this file path.

The first 50 rows are displayed below.

**Case counts aggregated by hospital**

For the purposes of the handbook, the dataset of weekly aggregated counts by hospital is created from the linelist with the following code.

The first 50 rows are displayed below:

### Set parameters

For production of a report, you may want to set editable parameters such as the date for which the data is current (the “data date”). You can then reference the object data\_date in your code when applying filters or in dynamic captions.

### Verify dates

Verify that each relevant date column is class Date and has an appropriate range of values. You can do this simply using hist() for histograms, or range() with na.rm=TRUE, or with ggplot() as below.

## Epicurves with ****incidence2**** package

Below we demonstrate how to make epicurves using the **incidence2** package. The authors of this package have tried to allow the user to create and modify epicurves without needing to know **ggplot2** syntax. Much of this page is adapted from the package vignettes, which can be found at the **incidence2** [github page](https://github.com/reconhub/incidence2).

### Simple example

**2 steps are required to plot an epidemic curve with the incidence2 package:**

1. **Create** an incidence object (using the function incidence())
   * Provide the data
   * Specify the date column to date\_index =
   * Specify the interval = into which the cases should be aggregated (daily, weekly, monthly..)
   * Specify any grouping columns (e.g. gender, hospital, outcome)
2. **Plot** the incidence object
   * Specify labels, colors, titles, etc.

Below, we load the **incidence2** package, create the incidence object from the linelist on column date\_onset and aggregated cases by day. We then print a summary of the incidence object.

The **incidence2** object itself looks like a tibble (like a data frame) and can be printed or further manipulated like a data frame.

Here is what it looks like when printed. It has a date\_index column and a count column.

You can also print a summary of the object:

To plot the incidence object, use plot() on the name of the incidence object. In the background, the function plot.incidence2() is called, so to read the **incidence2**-specific documentation you would run ?plot.incidence2.

If you notice lots of tiny white vertical lines, try to adjust the size of your image. For example, if you export your plot with ggsave(), you can provide numbers to width = and height =. If you widen the plot those lines may disappear.

### Change time interval of case aggregation

The interval = argument of incidence() defines how the observations are grouped into vertical bars.

**Specify interval**

**incidence2** provides flexibility and understandable syntax for specifying how you want to aggregate your cases into epicurve bars. Provide a value like the ones below to the interval = argument. You can write any of the below as plural (e.g. “week**s**”), and you can add numbers before (e.g. “3 months”).

| **Argument option** | **Further explanation** |
| --- | --- |
| Number (1, 7, 13, 14, etc.) | Number of days per interval |
| “week” | note: Monday start day is default |
| “2 weeks” | or 3, 4, 5… |
| “Sunday week” | weeks beginning on Sundays (could also use Thursday, etc.) |
| “2 Sunday weeks” | or 3, 4, 5… |
| “MMWRweek” | week starts on Sundays - see US CDC |
| “month” | 1st of month |
| “quarter” | 1st of month of quarter |
| “2 months” | or 3, 4, 5… |
| “year” | 1st day of calendar year |

Below are examples of how different intervals look when applied to the linelist. Note how the default format and frequency of the date labels on the x-axis change as the date interval changes.

**First date**

You can optionally specify a value of class Date (e.g. as.Date("2016-05-01")) to firstdate = in the incidence() command. If given, the data will be trimmed to this range and the intervals will begin on this date.

### Groups

Groups are specified in the incidence() command, and can be used to color the bars or to facet the data. To specify groups in your data provide the column name(s) to the groups = argument in the incidence() command (no quotes around the column name). If specifying multiple columns, put their names within c().

You can specify that cases with missing values in the grouping columns be listed as a distinct NA group by setting na\_as\_group = TRUE. Otherwise, they will be excluded from the plot.

* To color the bars by a grouping column, you must again provide the column name to fill = in the plot() command.
* To facet based on a grouping column, see the section below on facets with **incidence2**.

In the example below, the cases in the whole outbreak are grouped by their age category. Missing values are included as a group. The epicurve interval is weeks.

**TIP:** Change the title of the legend by adding + the **ggplot2** command labs(fill = "your title") to your **incidence2** plot.

You can also have the grouped bars display side-by-side by setting stack = FALSE in plot(), as shown below:

You can set the na\_as\_group = argument to FALSE in the incidence() command to remove rows with missing values from the plot.

### Filtered data

To plot the epicurve of a subset of data:

1. Filter the linelist data
2. Provide the filtered data to the incidence() command
3. Plot the incidence object

The example below uses data filtered to show only cases at Central Hospital.

### Aggregated counts

If your original data are aggregated (counts), provide the name of the column that contains the case counts to the count = argument when creating the incidence object with incidence().

For example, this data frame count\_data is the linelist aggregated into daily counts by hospital. The first 50 rows look like this:

If you are beginning your analysis with daily count data like the dataset above, your incidence() command to convert this to a weekly epicurve by hospital would look like this:

### Facets/small multiples

To facet the data by group (i.e. produce “small multiples”):

1. Specify the faceting column to groups = when you create the incidence object
2. Use the facet\_plot() command instead of plot()
3. Specify which grouping columns to use as fill = and which to use as facets =

Below, we set both columns hospital and outcome as grouping columns in the incidence() command. Then, in facet\_plot() we plot the epicurve, specifying that we want a different epicurve for each hospital and that within each epicurve the bars should be stacked and colored by outcome.

Note that the package **ggtree** (used for displaying phylogenetic trees) also has a function facet\_plot() - this is why we specified incidence2::facet\_plot() above.

### Modifications with plot()

An epicurve produced by **incidence2** can be modified via these arguments within the *plot()* function.

**Here are plot() arguments that modify the appearance of the bars:**

| **Argument** | **Description** | **Examples** |
| --- | --- | --- |
| fill = | Bar color. Either a color name or a column name previously specified to groups = in the incidence() command | fill = "red", or fill = gender |
| color = | Color around each bar, or around each grouping within a bar | border = "white" |
| legend = | Location of legend | One of “bottom”, “top”, “left”, “right”, or “none” |
| alpha = | Transparency of bars/boxes | 1 is fully opaque, 0 is fully transparent |
| width = | Value between 0 and 1 indicating the relative size of the bars to their time interval | width = .7 |
| show\_cases = | Logical; if TRUE, each case shows as a box. Displays best on smaller outbreaks. | show\_cases = TRUE |

**Here are plot() arguments that modify the date axis:**

| **Argument(s)** | **Description** |
| --- | --- |
| centre\_dates = | TRUE/FALSE as to whether date displays appear under center of bars, or at beginning of bars |
| date\_format = | Adjust the date display format using strptime (“%”) syntax. Only works if centre\_dates = FALSE (details below). |
| n.breaks = | Approximate number of x-axis label breaks desired. |
| angle = | Angle of x-axis date labels (number of degrees) |
| size = | Size of text in points |

Note that the date\_breaks = argument only works if centre\_dates = FALSE. Provide a character value in quotation marks using the strptime syntax below, as detailed in the [Working with dates](#working-with-dates-1) page. You can use \n for a “newline”.

%d = Day number of month (5, 17, 28, etc.)  
%j = Day number of the year (Julian day 001-366)  
%a = Abbreviated weekday (Mon, Tue, Wed, etc.)  
%A = Full weekday (Monday, Tuesday, etc.)  
%w = Weekday number (0-6, Sunday is 0)  
%u = Weekday number (1-7, Monday is 1)  
%W = Week number (00-53, Monday is week start)  
%U = Week number (01-53, Sunday is week start)  
%m = Month number (e.g. 01, 02, 03, 04)  
%b = Abbreviated month (Jan, Feb, etc.)  
%B = Full month (January, February, etc.)  
%y = 2-digit year (e.g. 89)  
%Y = 4-digit year (e.g. 1989)  
%h = hours (24-hr clock)  
%m = minutes  
%s = seconds  
%z = offset from GMT  
%Z = Time zone (character)

**Here are plot() arguments that modify plot labels:**

| **Argument(s)** | **Description** |
| --- | --- |
| title = | Title of plot |
| xlab = | Title of x-axis |
| ylab = | Title of y-axis |
| size = | Size of x-axis text in pts (use ggplot’s theme() to adjust other sizes) |

An example using many of the above arguments:

To further adjust plot appearance, see the section below on modifications with ggplot().

### Modifications with ggplot2

You can further modify an **incidence2** plot by adding **ggplot2** modifications with a + after the close of the incidence plot() function, as demonstrated below.

Below, the **incidence2** plot finishes and then **ggplot2** commands are used to modify the axes, add a caption, and adjust the bold font and text size.

Note that if you add scale\_x\_date(), most date formatting from plot() will be overwritten. See the ggplot() epicurves section and the Handbook page [ggplot tips](#ggplot-tips) for more options.

### Change colors

#### Specify a palette

Provide the name of a pre-defined palette to the col\_pal = argument in plot(). The **incidence2** package comes with 2 pre-defined paletted: “vibrant” and “muted”. In “vibrant” the first 6 colors and distinct and in “muted” the first 9 colors are distinct. After these numbers, the colors are interpolations/intermediaries of other colors. These pre-defined palettes can be found at [this website](https://personal.sron.nl/~pault/#sec:qualitative). The palettes exclude grey, which is reserved for missing data (use na\_color = to change this default).

You can also use one of the **base** R palettes (put the name of the palette without quotes).

You can also add a color palette from the **viridis** package or **RColorBrewer** package. First those packages must be loaded, then add their respective scale\_fill\_\*() functions with a +, as shown below.

#### Specify manually

To specify colors manually, add the **ggplot2** function scale\_fill\_manual() to the plot() with a + and provide the vector of colors names or HEX codes to the argument values =. The number of colors listed must equal the number of groups. Be aware of whether missing values are a group - they can be converted to a character value like “Missing” during your data preparation with the function fct\_explicit\_na() as explained in the page on [Factors](#factors).

As mentioned in the [ggplot tips](#ggplot-tips) page, you can create your own palettes using colorRampPalette() on a vector of colors and specifying the number of colors you want in return. This is a good way to get many colors in a ramp by specifying a few.

### Adjust level order

To adjust the order of group appearance (on plot and in legend), the grouping column must be class Factor. See the page on [Factors](#factors) for more information.

First, let’s see a weekly epicurve by hospital with the default ordering:

Now, to adjust the order so that “Missing” and “Other” are at the top of the epicurve we can do the following:

* Load the package **forcats**, to work with factors
* Adjust the dataset - in this case we’ll define a new dataset (plot\_data) in which:
  + the gender column is defined as a factor the order of levels are set with fct\_relevel() so that “Other” and “Missing” are first, so they appear at the top of the bars
* The incidence object is created and plotted as before
* We add **ggplot2** modifications
  + scale\_fill\_manual() to manually assign colors so that “Missing” is grey and “Other” is beige

**TIP:** If you want to reverse the order of the legend only, add this **ggplot2** command guides(fill = guide\_legend(reverse = TRUE)).

### Vertical gridlines

If you plot with default **incidence2** settings, you may notice that the vertical gridlines appear at each date label and once between each date label. This can result in gridlines intersecting with the top of some bars.

You can remove all gridlines by adding the **ggplot2** command theme\_classic().

Note however, that if using weeks, the date\_breaks and date\_minor\_breaks arguments only work for Monday weeks. If your weeks are by another day of the week you will need to manually provide a vector of dates to the breaks = and minor\_breaks = arguments instead. See the **ggplot2** section for examples of this using seq.Date().

### Cumulative incidence

You can easily produce a plot of cumulative incidence by passing the incidence object to the **incidence2** command cumulate() and then to plot(). This also works with facet\_plot().

See the section farther down on this page for alternative method to plot cumulative incidence with **ggplot2** - for example to overlay a cumulative incidence line over an epicurve.

### Rolling average

You can add a rolling average to an **incidence2** plot easily with add\_rolling\_average() from the **i2extras** package. Pass your incidence2 object to this function, and then to plot(). Set before = as the number of previous days you want included in the rolling average (default is 2). If your data are grouped, the rolling average will be calculated per group.

To learn how to apply rolling averages more generally on data, see the Handbook page on [Moving averages](#moving-averages-1).

## Epicurves with ggplot2

Using ggplot() to build your epicurve allows for more flexibility and customization, but requires more effort and understanding of how ggplot() works.

Unlike using the **incidence2** package, you must manually control the aggregation of the cases by time (into weeks, months, etc) and the intervals of the labels on the date axis. This must be carefully managed.

These examples use a subset of the linelist dataset - only the cases from Central Hospital.

To produce an epicurve with ggplot() there are three main elements:

* A histogram, with linelist cases aggregated into “bins” distinguished by specific “break” points
* Scales for the axes and their labels
* Themes for the plot appearance, including titles, labels, captions, etc.

### Specify case bins

Here we show how to specify how cases will be aggregated into histogram bins (“bars”). It is important to recognize that the aggregation of cases into histogram bins is **not** necessarily the same intervals as the dates that will appear on the x-axis.

Below is perhaps the most simple code to produce daily and weekly epicurves.

In the over-arching ggplot() command the dataset is provided to data =. Onto this foundation, the geometry of a histogram is added with a +. Within the geom\_histogram(), we map the aesthetics such that the column date\_onset is mapped to the x-axis. Also within the geom\_histogram() but not within aes() we set the binwidth = of the histogram bins, in days. If this **ggplot2** syntax is confusing, review the page on [ggplot basics](#ggplot-basics).

**CAUTION:** Plotting weekly cases by using binwidth = 7 starts the first 7-day bin at the first case, which could be any day of the week! To create specific weeks, see section below .

Let us note that the first case in this Central Hospital dataset had symptom onset on:

**To manually specify the histogram bin breaks, do not use the binwidth = argument, and instead supply a vector of dates to breaks =.**

Create the vector of dates with the **base** R function seq.Date(). This function expects arguments to =, from =, and by =. For example, the command below returns monthly dates starting at Jan 15 and ending by June 28.

This vector can be provided to geom\_histogram() as breaks =:

A simple weekly date sequence can be returned by setting by = "week". For example:

An alternative to supplying specific start and end dates is to write dynamic code so that weekly bins begin the Monday before the first case. **We will use these date vectors throughout the examples below.**

Let’s unpack the rather daunting code above:

* The “from” value (earliest date of the sequence) is created as follows: the minimum date value (min() with na.rm=TRUE) in the column date\_onset is fed to floor\_date() from the **lubridate** package. floor\_date() set to “week” returns the start date of that cases’s “week”, given that the start day of each week is a Monday (week\_start = 1).
* Likewise, the “to” value (end date of the sequence) is created using the inverse function ceiling\_date() to return the Monday after the last case.
* The “by” argument of seq.Date() can be set to any number of days, weeks, or months.
* Use week\_start = 7 for Sunday weeks

As we will use these date vectors throughout this page, we also define one for the whole outbreak (the above is for Central Hospital only).

These seq.Date() outputs can be used to create histogram bin breaks, but also the breaks for the date labels, which may be independent from the bins. Read more about the date labels in later sections.

**TIP:** For a more simple ggplot() command, save the bin breaks and date label breaks as named vectors in advance, and simply provide their names to breaks =.

### Weekly epicurve example

**Below is detailed example code to produce weekly epicurves for Monday weeks, with aligned bars, date labels, and vertical gridlines.** This section is for the user who needs code quickly. To understand each aspect (themes, date labels, etc.) in-depth, continue to the subsequent sections. Of note:

* The histogram bin breaks are defined with seq.Date() as explained above to begin the Monday before the earliest case and to end the Monday after the last case
* The interval of date labels is specified by date\_breaks = within scale\_x\_date()
* The interval of minor vertical gridlines between date labels is specified to date\_minor\_breaks =
* expand = c(0,0) in the x and y scales removes excess space on each side of the axes, which also ensures the date labels begin from the first bar.

#### Sunday weeks

To achieve the above plot for Sunday weeks a few modifications are needed, because the date\_breaks = "weeks" only work for Monday weeks.

* The break points of the histogram bins must be set to Sundays (week\_start = 7)
* Within scale\_x\_date(), the similar date breaks should be provided to breaks = and minor\_breaks = to ensure the date labels and vertical gridlines align on Sundays.

For example, the scale\_x\_date() command for Sunday weeks could look like this:

### Group/color by value

The histogram bars can be colored by group and “stacked”. To designate the grouping column, make the following changes. See the [ggplot basics](#ggplot-basics) page for details.

* Within the histogram aesthetic mapping aes(), map the column name to the group = and fill = arguments
* Remove any fill = argument outside of aes(), as it will override the one inside
* Arguments inside aes() will apply by group, whereas any outside will apply to all bars (e.g. you may still want color = outside, so each bar has the same border)

Here is what the aes() command would look like to group and color the bars by gender:

Here it is applied:

### Adjust colors

* To manually set the fill for each group, use scale\_fill\_manual() (note: scale\_color\_manual() is different!).
  + Use the values = argument to apply a vector of colors.
  + Use na.value = to specify a color for NA values.
  + Use the labels = argument to change the text of legend items. To be safe, provide as a named vector like c("old" = "new", "old" = "new") or adjust the values in the data itself.
  + Use name = to give a proper title to the legend
* For more tips on color scales and palettes, see the page on [ggplot basics](#ggplot-basics).

### Adjust level order

The order in which grouped bars are stacked is best adjusted by classifying the grouping column as class Factor. You can then designate the factor level order (and their display labels). See the page on [Factors](#factors) or [ggplot tips](#ggplot-tips) for details.

Before making the plot, use the fct\_relevel() function from **forcats** package to convert the grouping column to class factor and manually adjust the level order, as detailed in the page on [Factors](#factors).

In the below plot, the only differences from previous is that column hospital has been consolidated as above, and we use guides() to reverse the legend order, so that “Missing” is on the bottom of the legend.

**TIP:** To reverse the order of the legend only, add this **ggplot2** command: guides(fill = guide\_legend(reverse = TRUE)).

### Adjust legend

Read more about legends and scales in the [ggplot tips](#ggplot-tips) page. Here are a few highlights:

* Edit legend title either in the scale function or with labs(fill = "Legend title") (if your are using color = aesthetic, then use labs(color = ""))
* theme(legend.title = element\_blank()) to have no legend title
* theme(legend.position = "top") (“bottom”, “left”, “right”, or “none” to remove the legend)
* theme(legend.direction = "horizontal") horizontal legend
* guides(fill = guide\_legend(reverse = TRUE)) to reverse order of the legend

### Bars side-by-side

Side-by-side display of group bars (as opposed to stacked) is specified within geom\_histogram() with position = "dodge" placed outside of aes().

If there are more than two value groups, these can become difficult to read. Consider instead using a faceted plot (small multiples). To improve readability in this example, missing gender values are removed.

### Axis limits

There are two ways to limit the extent of axis values.

Generally the preferred way is to use the command coord\_cartesian(), which accepts xlim = c(min, max) and ylim = c(min, max) (where you provide the min and max values). This acts as a “zoom” without actually removing any data, which is important for statistics and summary measures.

Alternatively, you can set maximum and minimum date values using limits = c() within scale\_x\_date(). For example:

Likewise, if you want to the x-axis to extend to a specific date (e.g. current date), even if no new cases have been reported, you can use:

**DANGER:** Be cautious setting the y-axis scale breaks or limits (e.g. 0 to 30 by 5: seq(0, 30, 5)). Such static numbers can cut-off your plot too short if the data changes to exceed the limit!.

### Date-axis labels/gridlines

**TIP:** Remember that date-axis **labels** are independent from the aggregation of the data into bars, but visually it can be important to align bins, date labels, and vertical grid lines.

To **modify the date labels and grid lines**, use scale\_x\_date() in one of these ways:

* **If your histogram bins are days, Monday weeks, months, or years**:
  + Use date\_breaks = to specify the interval of labels and major gridlines (e.g. “day”, “week”, “3 weeks”, “month”, or “year”)
  + Use date\_minor\_breaks = to specify interval of minor vertical gridlines (between date labels)
  + Add expand = c(0,0) to begin the labels at the first bar
  + Use date\_labels = to specify format of date labels - see the Dates page for tips (use \n for a new line)
* **If your histogram bins are Sunday weeks**:
  + Use breaks = and minor\_breaks = by providing a sequence of date breaks for each
  + You can still use date\_labels = and expand = for formatting as described above

Some notes:

* See the opening ggplot section for instructions on how to create a sequence of dates using seq.Date().
* See [this page](https://rdrr.io/r/base/strptime.html) or the [Working with dates](#working-with-dates-1) page for tips on creating date labels.

#### Demonstrations

Below is a demonstration of plots where the bins and the plot labels/grid lines are aligned and not aligned:

### Aggregated data

Often instead of a linelist, you begin with aggregated counts from facilities, districts, etc. You can make an epicurve with ggplot() but the code will be slightly different. This section will utilize the count\_data dataset that was imported earlier, in the data preparation section. This dataset is the linelist aggregated to day-hospital counts. The first 50 rows are displayed below.

#### Plotting daily counts

We can plot a daily epicurve from these daily counts. Here are the differences to the code:

* Within the aesthetic mapping aes(), specify y = as the counts column (in this case, the column name is n\_cases)
* Add the argument stat = "identity" within geom\_histogram(), which specifies that bar height should be the y = value, not the number of rows as is the default
* Add the argument width = to avoid vertical white lines between the bars. For daily data set to 1. For weekly count data set to 7. For monthly count data, white lines are an issue (each month has different number of days) - consider transforming your x-axis to a categorical ordered factor (months) and using geom\_col().

#### Plotting weekly counts

If your data are already case counts by week, they might look like this dataset (called count\_data\_weekly):

The first 50 rows of count\_data\_weekly are displayed below. You can see that the counts have been aggregated into weeks. Each week is displayed by the first day of the week (Monday by default).

Now plot so that x = the epiweek column. Remember to add y = the counts column to the aesthetic mapping, and add stat = "identity" as explained above.

### Moving averages

See the page on [Moving averages](#moving-averages-1) for a detailed description and several options. Below is one option for calculating moving averages with the package **slider**. In this approach, the moving average is calculated in the dataset prior to plotting:

1. Aggregate the data into counts as necessary (daily, weekly, etc.) (see [Grouping data](#grouping-data) page)
2. Create a new column to hold the moving average, created with slide\_index() from **slider** package
3. Plot the moving average as a geom\_line() on top of (after) the epicurve histogram

See the helpful online [vignette for the **slider** package](https://cran.r-project.org/web/packages/slider/vignettes/slider.html)

### Faceting/small-multiples

As with other ggplots, you can create facetted plots (“small multiples”). As explained in the [ggplot tips](#ggplot-tips) page of this handbook, you can use either facet\_wrap() or facet\_grid(). Here we demonstrate with facet\_wrap(). For epicurves, facet\_wrap() is typically easier as it is likely that you only need to facet on one column.

The general syntax is facet\_wrap(rows ~ cols), where to the left of the tilde (~) is the name of a column to be spread across the “rows” of the facetted plot, and to the right of the tilde is the name of a column to be spread across the “columns” of the facetted plot. Most simply, just use one column name, to the right of the tilde: facet\_wrap(~age\_cat).

**Free axes**  
You will need to decide whether the scales of the axes for each facet are “fixed” to the same dimensions (default), or “free” (meaning they will change based on the data within the facet). Do this with the scales = argument within facet\_wrap() by specifying “free\_x” or “free\_y”, or “free”.

**Number of cols and rows of facets**  
This can be specified with ncol = and nrow = within facet\_wrap().

**Order of panels**  
To change the order of appearance, change the underlying order of the levels of the factor column used to create the facets.

**Aesthetics**  
Font size and face, strip color, etc. can be modified through theme() with arguments like:

* strip.text = element\_text() (size, colour, face, angle…)
* strip.background = element\_rect() (e.g. element\_rect(fill=“grey”))
* strip.position = (position of the strip “bottom”, “top”, “left”, or “right”)

**Strip labels**  
Labels of the facet plots can be modified through the “labels” of the column as a factor, or by the use of a “labeller”.

Make a labeller like this, using the function as\_labeller() from **ggplot2**. Then provide the labeller to the labeller = argument of facet\_wrap() as shown below.

**An example facetted plot** - facetted by column age\_cat.

See this [link](https://ggplot2.tidyverse.org/reference/labellers.html) for more information on labellers.

#### Total epidemic in facet background

To show the total epidemic in the background of each facet, add the function gghighlight() with empty parentheses to the ggplot. This is from the package **gghighlight**. Note that the y-axis maximum in all facets is now based on the peak of the entire epidemic. There are more examples of this package in the [ggplot tips](#ggplot-tips) page.

#### One facet with data

If you want to have one facet box that contains all the data, duplicate the entire dataset and treat the duplicates as one faceting value. A “helper” function CreateAllFacet() below can assist with this (thanks to this [blog post](https://stackoverflow.com/questions/18933575/easily-add-an-all-facet-to-facet-wrap-in-ggplot2)). When it is run, the number of rows doubles and there will be a new column called facet in which the duplicated rows will have the value “all”, and the original rows have the their original value of the faceting colum. Now you just have to facet on the facet column.

Here is the helper function. Run it so that it is available to you.

Now apply the helper function to the dataset, on column age\_cat:

Notable changes to the ggplot() command are:

* The data used is now central\_data2 (double the rows, with new column “facet”)
* Labeller will need to be updated, if used
* Optional: to achieve vertically stacked facets: the facet column is moved to rows side of equation and on right is replaced by “.” (facet\_wrap(facet~.)), and ncol = 1. You may also need to adjust the width and height of the saved png plot image (see ggsave() in [ggplot tips](#ggplot-tips)).

## Tentative data

The most recent data shown in epicurves should often be marked as tentative, or subject to reporting delays. This can be done in by adding a vertical line and/or rectangle over a specified number of days. Here are two options:

1. Use annotate():
   * For a line use annotate(geom = "segment"). Provide x, xend, y, and yend. Adjust size, linetype (lty), and color.
   * For a rectangle use annotate(geom = "rect"). Provide xmin/xmax/ymin/ymax. Adjust color and alpha.
2. Group the data by tentative status and color those bars differently

**CAUTION:** You might try geom\_rect() to draw a rectangle, but adjusting the transparency does not work in a linelist context. This function overlays one rectangle for each observation/row!. Use either a very low alpha (e.g. 0.01), or another approach.

### Using annotate()

* Within annotate(geom = "rect"), the xmin and xmax arguments must be given inputs of class Date.
* Note that because these data are aggregated into weekly bars, and the last bar extends to the Monday after the last data point, the shaded region may appear to cover 4 weeks
* Here is an annotate() [online example](https://ggplot2.tidyverse.org/reference/annotate.html)

The same black vertical line can be achieved with the code below, but using geom\_vline() you lose the ability to control the height:

### Bars color

An alternative approach could be to adjust the color or display of the tentative bars of data themselves. You could create a new column in the data preparation stage and use it to group the data, such that the aes(fill = ) of tentative data can be a different color or alpha than the other bars.

## Multi-level date labels

If you want multi-level date labels (e.g. month and year) without duplicating the lower label levels, consider one of the approaches below:

Remember - you can can use tools like \n within the date\_labels or labels arguments to put parts of each label on a new line below. However, the code below helps you take years or months (for example) on a lower line and only once. A few notes on the code below:

* Case counts are aggregated into weeks for aesthetic reasons. See Epicurves page (aggregated data tab) for details.
* A geom\_area() line is used instead of a histogram, as the faceting approach below does not work well with histograms.

**Aggregate to weekly counts**

**Make plots**

The above techniques were adapted from [this](https://stackoverflow.com/questions/44616530/axis-labels-on-two-lines-with-nested-x-variables-year-below-months) and [this](https://stackoverflow.com/questions/20571306/multi-row-x-axis-labels-in-ggplot-line-chart) post on stackoverflow.com.

## Dual-axis

Although there are fierce discussions about the validity of dual axes within the data visualization community, many epi supervisors still want to see an epicurve or similar chart with a percent overlaid with a second axis. This is discussed more extensively in the [ggplot tips](#ggplot-tips) page, but one example using the **cowplot** method is shown below:

* Two distinct plots are made, and then combined with **cowplot** package.
* The plots must have the exact same x-axis (set limits) or else the data and labels will not align
* Each uses theme\_cowplot() and one has the y-axis moved to the right side of the plot

Now use **cowplot** to overlay the two plots. Attention has been paid to the x-axis alignment, side of the y-axis, and use of theme\_cowplot().

## Cumulative Incidence

Note: If using **incidence2**, see the section on how you can produce cumulative incidence with a simple function. This page will address how to calculate cumulative incidence and plot it with ggplot().

If beginning with a case linelist, create a new column containing the cumulative number of cases per day in an outbreak using cumsum() from **base** R:

The first 10 rows are shown below:

This cumulative column can then be plotted against date\_onset, using geom\_line():

It can also be overlaid onto the epicurve, with dual-axis using the **cowplot** method described above and in the [ggplot tips](#ggplot-tips) page:

Now use **cowplot** to overlay the two plots. Attention has been paid to the x-axis alignment, side of the y-axis, and use of theme\_cowplot().

## Resources

# Demographic pyramids and Likert-scales

Demographic pyramids are useful to show distributions of age and gender. Similar code can be used to visualize the results of Likert-style survey questions (e.g. “Strongly agree”, “Agree”, “Neutral”, “Disagree”, “Strongly disagree”). In this page we cover the following:

* Fast & easy pyramids using the **apyramid** package
* More customizeable pyramids using ggplot()
* Displaying “baseline” demographics in the background of the pyramid
* Using pyramid-style plots to show other types of data (e.g responses to **Likert-style** survey questions)

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

To begin, we import the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

### Cleaning

To make a traditional age/gender demographic pyramid, the data must first be cleaned in the following ways:

* The gender column must be cleaned.
* Depending on your method, age should be stored as either a numeric or in an age category column.

If using age categories, the column values should be corrected ordered, either by default alpha-numeric or intentionally set by converting to class factor.

Below we use tabyl() from **janitor** to inspect the columns gender and age\_cat5.

We also run a quick histogram on the age column to ensure it is clean and correctly classified:

## ****apyramid**** package

The package **apyramid** is a product of the [R4Epis](https://r4epis.netlify.com/) project. You can read more about this package [here](https://cran.r-project.org/web/packages/apyramid/vignettes/intro.html). It allows you to quickly make an age pyramid. For more nuanced situations, see the section below [using ggplot()](#demo_pyr_gg). You can read more about the **apyramid** package in its Help page by entering ?age\_pyramid in your R console.

### Linelist data

Using the cleaned linelist dataset, we can create an age pyramid with one simple age\_pyramid() command. In this command:

* The data = argument is set as the linelist data frame
* The age\_group = argument (for y-axis) is set to the name of the categorical age column (in quotes)
* The split\_by = argument (for x-axis) is set to the gender column

The pyramid can be displayed with percent of all cases on the x-axis, instead of counts, by including proportional = TRUE.

When using **agepyramid** package, if the split\_by column is binary (e.g. male/female, or yes/no), then the result will appear as a pyramid. However if there are more than two values in the split\_by column (not including NA), the pyramid will appears as a faceted bar plot with grey bars in the “background” indicating the range of the un-faceted data for that age group. In this case, values of split\_by = will appear as labels at top of each facet panel. For example, below is what occurs if the split\_by = is assigned the column hospital.

#### Missing values

Rows that have NA missing values in the split\_by = or age\_group = columns, if coded as NA, will not trigger the faceting shown above. By default these rows will not be shown. However you can specify that they appear, in an adjacent barplot and as a separate age group at the top, by specifying na.rm = FALSE.

#### Proportions, colors, & aesthetics

By default, the bars display counts (not %), a dashed mid-line for each group is shown, and the colors are green/purple. Each of these parameters can be adjusted, as shown below:

You can also add additional ggplot() commands to the plot using the standard ggplot() “+” syntax, such as aesthetic themes and label adjustments:

### Aggregated data

The examples above assume your data are in a linelist format, with one row per observation. If your data are already aggregated into counts by age category, you can still use the **apyramid** package, as shown below.

For demonstration, we aggregate the linelist data into counts by age category and gender, into a “wide” format. This will simulate as if your data were in counts to begin with. Learn more about [Grouping data](#grouping-data) and [Pivoting data](#pivoting-data) in their respective pages.

…which makes the dataset looks like this: with columns for age category, and male counts, female counts, and missing counts.

To set-up these data for the age pyramid, we will pivot the data to be “long” with the pivot\_longer() function from **dplyr**. This is because ggplot() generally prefers “long” data, and **apyramid** is using ggplot().

Then use the split\_by = and count = arguments of age\_pyramid() to specify the respective columns in the data:

Note in the above, that the factor order of “m” and “f” is different (pyramid reversed). To adjust the order you must re-define gender in the aggregated data as a Factor and order the levels as desired. See the [Factors](#factors) page.

## ggplot()

Using ggplot() to build your age pyramid allows for more flexibility, but requires more effort and understanding of how ggplot() works. It is also easier to accidentally make mistakes.

To use ggplot() to make demographic pyramids, you create two bar plots (one for each gender), convert the values in one plot to negative, and finally flip the x and y axes to display the bar plots vertically, their bases meeting in the plot middle.

### Preparation

This approach uses the numeric age column, not the categorical column of age\_cat5. So we will check to ensure the class of this column is indeed numeric.

You could use the same logic below to build a pyramid from categorical data using geom\_col() instead of geom\_histogram().

### Constructing the plot

First, understand that to make such a pyramid using ggplot() the approach is as follows:

* Within the ggplot(), create **two** histograms using the numeric age column. Create one for each of the two grouping values (in this case genders male and female). To do this, the data for each histogram are specified within their respective geom\_histogram() commands, with the respective filters applied to linelist.
* One graph will have positive count values, while the other will have its counts converted to negative values - this creates the “pyramid” with the 0 value in the middle of the plot. The negative values are created using a special **ggplot2** term ..count.. and multiplying by -1.
* The command coord\_flip() switches the X and Y axes, resulting in the graphs turning vertical and creating the pyramid.
* Lastly, the counts-axis value labels must be altered so they appear as “positive” counts on both sides of the pyramid (despite the underlying values on one side being negative).

A **simple** version of this, using geom\_histogram(), is below:

**DANGER:** If the **limits** of your counts axis are set too low, and a counts bar exceeds them, the bar will disappear entirely or be artificially shortened! Watch for this if analyzing data which is routinely updated. Prevent it by having your count-axis limits auto-adjust to your data, as below.

There are many things you can change/add to this simple version, including:

* Auto adjust counts-axis scale to your data (avoid errors discussed in warning below)
* Manually specify colors and legend labels

**Convert counts to percents**

To convert counts to percents (of total), do this in your data prior to plotting. Below, we get the age-gender counts, then ungroup(), and then mutate() to create new percent columns. If you want percents by gender, skip the ungroup step.

Importantly, we save the max and min values so we know what the limits of the scale should be. These will be used in the ggplot() command below.

Finally we make the ggplot() on the percent data. We specify scale\_y\_continuous() to extend the pre-defined lengths in each direction (positive and “negative”). We use floor() and ceiling() to round decimals the appropriate direction (down or up) for the side of the axis.

### Compare to baseline

With the flexibility of ggplot(), you can have a second layer of bars in the background that represent the “true” or “baseline” population pyramid. This can provide a nice visualization to compare the observed with the baseline.

Import and view the population data (see [Download handbook and data](#download-handbook-and-data) page):

First some data management steps:

Here we record the order of age categories that we want to appear. Due to some quirks the way the ggplot() is implemented, in this specific scenario it is easiest to store these as a character vector and use them later in the plotting function.

Combine the population and case data through the **dplyr** function bind\_rows():

* First, ensure they have the exact same column names, age categories values, and gender values
* Make them have the same data structure: columns of age category, gender, counts, and percent of total
* Bind them together, one on-top of the other (bind\_rows())

Review the changed population dataset

Now implement the same for the case linelist. Slightly different because it begins with case-rows, not counts.

Review the changed case dataset

Now the two data frames are combined, one on top of the other (they have the same column names). We can “name” each of the data frame, and use the .id = argument to create a new column “data\_source” that will indicate which data frame each row originated from. We can use this column to filter in the ggplot().

Store the maximum and minimum percent values, used in the plotting function to define the extent of the plot (and not cut short any bars!)

Now the plot is made with ggplot():

* One bar graph of population data (wider, more transparent bars)
* One bar graph of case data (small, more solid bars)

## Likert scale

The techniques used to make a population pyramid with ggplot() can also be used to make plots of Likert-scale survey data.

Import the data (see [Download handbook and data](#download-handbook-and-data) page if desired).

Start with data that looks like this, with a categorical classification of each respondent (status) and their answers to 8 questions on a 4-point Likert-type scale (“Very poor”, “Poor”, “Good”, “Very good”).

First, some data management steps:

* Pivot the data longer
* Create new column direction depending on whether response was generally “positive” or “negative”
* Set the Factor level order for the status column and the Response column
* Store the max count value so limits of plot are appropriate

Now make the plot. As in the age pyramids above, we are creating two bar plots and inverting the values of one of them to negative.

We use geom\_bar() because our data are one row per observation, not aggregated counts. We use the special **ggplot2** term ..count.. in one of the bar plots to invert the values negative (\*-1), and we set position = "stack" so the values stack on top of each other.

## Resources

[apyramid documentation](https://cran.r-project.org/web/packages/apyramid/vignettes/intro.html)

# Heat plots

Heat plots, also known as “heat maps” or “heat tiles”, can be useful visualizations when trying to display 3 variables (x-axis, y-axis, and fill). Below we demonstrate two examples:

* A visual matrix of transmission events by age (“who infected whom”)
* Tracking reporting metrics across many facilities/jurisdictions over time

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

**Datasets**

This page utilizes the case linelist of a simulated outbreak for the transmission matrix section, and a separate dataset of daily malaria case counts by facility for the metrics tracking section. They are loaded and cleaned in their individual sections.

## Transmission matrix

Heat tiles can be useful to visualize matrices. One example is to display “who-infected-whom” in an outbreak. This assumes that you have information on transmission events.

Note that the [Contact tracing](#contact-tracing-1) page contains another example of making a heat tile contact matrix, using a different (perhaps more simple) dataset where the ages of cases and their sources are neatly aligned in the same row of the data frame. This same data is used to make a density map in the [ggplot tips](#ggplot-tips) page. This example below begins from a case linelist and so involves considerable data manipulation prior to achieving a plotable data frame. So there are many scenarios to chose from…

We begin from the case linelist of a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import your data with the import() function from the **rio** package (it accepts many file types like .xlsx, .rds, .csv - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are shown below for demonstration:

In this linelist:

* There is one row per case, as identified by case\_id
* There is a later column infector that contains the case\_id of the infector, who is also a case in the linelist

### Data preparation

**Objective**: We need to achieve a “long”-style data frame that contains one row per possible age-to-age transmission route, with a numeric column containing that row’s proportion of all observed transmission events in the linelist.

This will take several data manuipulation steps to achieve:

#### Make cases data frame

To begin, we create a data frame of the cases, their ages, and their infectors - we call the data frame case\_ages. The first 50 rows are displayed below.

#### Make infectors data frame

Next, we create a data frame of the infectors - at the moment it consists of a single column. These are the infector IDs from the linelist. Not every case has a known infector, so we remove missing values. The first 50 rows are displayed below.

Next, we use joins to procure the ages of the infectors. This is not simple, because in the linelist, the infector’s ages are not listed as such. We achieve this result by joining the case linelist to the infectors. We begin with the infectors, and left\_join() (add) the case linelist such that the infector id column left-side “baseline” data frame joins to the case\_id column in the right-side linelist data frame.

Thus, the data from the infector’s case record in the linelist (including age) is added to the infector row. The 50 first rows are displayed below.

Then, we combine the cases and their ages with the infectors and their ages. Each of these data frame has the column infector, so it is used for the join. The first rows are displayed below:

Below, a simple cross-tabulation of counts between the case and infector age groups. Labels added for clarity.

We can convert this table to a data frame with data.frame() from **base** R, which also automatically converts it to “long” format, which is desired for the ggplot(). The first rows are shown below.

Now we do the same, but apply prop.table() from **base** R to the table so instead of counts we get proportions of the total. The first 50 rows are shown below.

### Create heat plot

Now finally we can create the heat plot with **ggplot2** package, using the geom\_tile() function. See the [ggplot tips](#ggplot-tips) page to learn more extensively about color/fill scales, especially the scale\_fill\_gradient() function.

* In the aesthetics aes() of geom\_tile() set the x and y as the case age and infector age
* Also in aes() set the argument fill = to the Freq column - this is the value that will be converted to a tile color
* Set a scale color with scale\_fill\_gradient() - you can specify the high/low colors
  + Note that scale\_color\_gradient() is different! In this case you want the fill
* Because the color is made via “fill”, you can use the fill = argument in labs() to change the legend title

## Reporting metrics over time

Often in public health, one objective is to assess trends over time for many entities (facilities, jurisdictions, etc.). One way to visualize such trends over time is a heat plot where the x-axis is time and on the y-axis are the many entities.

### Data preparation

We begin by importing a dataset of daily malaria reports from many facilities. The reports contain a date, province, district, and malaria counts. See the page on [Download handbook and data](#download-handbook-and-data) for information on how to download these data. Below are the first 30 rows:

#### Aggregate and summarize

**The objective in this example** is to transform the daily facility total malaria case counts (seen in previous tab) into weekly summary statistics of facility reporting performance - in this case the proportion of days per week that the facility reported any data. For this example we will show data only for **Spring District**.

To achieve this we will do the following data management steps:

1. Filter the data as appropriate (by place, date)
2. Create a week column using floor\_date() from package **lubridate**
   * This function returns the start-date of a given date’s week, using a specified start date of each week (e.g. “Mondays”)
3. The data are grouped by columns “location” and “week” to create analysis units of “facility-week”
4. The function summarise() creates new columns to reflecting summary statistics per facility-week group:
   * Number of days per week (7 - a static value)
   * Number of reports received from the facility-week (could be more than 7!)
   * Sum of malaria cases reported by the facility-week (just for interest)
   * Number of unique days in the facility-week for which there is data reported
   * **Percent of the 7 days per facility-week for which data was reported**
5. The data frame is joined with right\_join() to a comprehensive list of all possible facility-week combinations, to make the dataset complete. The matrix of all possible combinations is created by applying expand() to those two columns of the data frame as it is at that moment in the pipe chain (represented by .). Because a right\_join() is used, all rows in the expand() data frame are kept, and added to agg\_weeks if necessary. These new rows appear with NA (missing) summarized values.

Below we demonstrate step-by-step:

Now the dataset has nrow(agg\_weeks) rows, when it previously had nrow(facility\_count\_data).

Next we create a week column reflecting the start date of the week for each record. This is achieved with the **lubridate** package and the function floor\_date(), which is set to “week” and for the weeks to begin on Mondays (day 1 of the week - Sundays would be 7). The top rows are shown below.

The new week column can be seen on the far right of the data frame

Now we group the data into facility-weeks and summarise them to produce statistics per facility-week. See the page on [Descriptive tables](#descriptive-tables) for tips. The grouping itself doesn’t change the data frame, but it impacts how the subsequent summary statistics are calculated.

The top rows are shown below. Note how the columns have completely changed to reflect the desired summary statistics. Each row reflects one facility-week.

Finally, we run the command below to ensure that ALL possible facility-weeks are present in the data, even if they were missing before.

We are using a right\_join() on itself (the dataset is represented by “.”) but having been expanded to include all possible combinations of the columns week and location\_name. See documentation on the expand() function in the page on [Pivoting]. Before running this code the dataset contains nrow(agg\_weeks) rows.

Here is expanded\_weeks:

Before running this code, agg\_weeks contains nrow(agg\_weeks) rows.

After running this code, agg\_weeks contains nrow(agg\_weeks) rows.

### Create heat plot

The ggplot() is made using geom\_tile() from the **ggplot2** package:

* Weeks on the x-axis is transformed to dates, allowing use of scale\_x\_date()
* location\_name on the y-axis will show all facility names
* The fill is p\_days\_reported, the performance for that facility-week (numeric)
* scale\_fill\_gradient() is used on the numeric fill, specifying colors for high, low, and NA
* scale\_x\_date() is used on the x-axis specifying labels every 2 weeks and their format
* Display themes and labels can be adjusted as necessary

### Basic

A basic heat plot is produced below, using the default colors, scales, etc. As explained above, within the aes() for geom\_tile() you must provide an x-axis column, y-axis column, **and** a column for the the fill =. The fill is the numeric value that presents as tile color.

### Cleaned plot

We can make this plot look better by adding additional **ggplot2** functions, as shown below. See the page on [ggplot tips](#ggplot-tips) for details.

### Ordered y-axis

Currently, the facilities are ordered “alpha-numerically” from the bottom to the top. If you want to adjust the order the y-axis facilities, convert them to class factor and provide the order. See the page on [Factors](#factors) for tips.

Since there are many facilities and we don’t want to write them all out, we will try another approach - ordering the facilities in a data frame and using the resulting column of names as the factor level order. Below, the column location\_name is converted to a factor, and the order of its levels is set based on the total number of reporting days filed by the facility across the whole time-span.

To do this, we create a data frame which represents the total number of reports per facility, arranged in ascending order. We can use this vector to order the factor levels in the plot.

See the data frame below:

Now use a column from the above data frame (facility\_order$location\_name) to be the order of the factor levels of location\_name in the data frame agg\_weeks:

And now the data are re-plotted, with location\_name being an ordered factor:

### Display values

You can add a geom\_text() layer on top of the tiles, to display the actual numbers of each tile. Be aware this may not look pretty if you have many small tiles!

The following code has been added: geom\_text(aes(label = p\_days\_reported)). This adds text onto every tile. The text displayed is the value assigned to the argument label =, which in this case has been set to the same numeric column p\_days\_reported that is also used to create the color gradient.

## Resources

[scale\_fill\_gradient()](https://ggplot2.tidyverse.org/reference/scale_gradient.html)

[R graph gallery - heatmap](https://ggplot2.tidyverse.org/reference/scale_gradient.html)

# Diagrams and charts

This page covers code to produce:

* Flow diagrams using **DiagrammeR** and the DOT language
* Alluvial/Sankey diagrams
* Event timelines

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

Most of the content in this page does not require a dataset. However, in the Sankey diagram section, we will use the case linelist from a simulated Ebola epidemic. If you want to follow along for this part, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Flow diagrams

One can use the R package **DiagrammeR** to create charts/flow charts. They can be static, or they can adjust somewhat dynamically based on changes in a dataset.

**Tools**

The function grViz() is used to create a “Graphviz” diagram. This function accepts a character string input containing instructions for making the diagram. Within that string, the instructions are written in a different language, called [DOT](https://graphviz.org/doc/info/lang.html) - it is quite easy to learn the basics.

**Basic structure**

1. Open the instructions grViz("
2. Specify directionality and name of the graph, and open brackets, e.g. digraph my\_flow\_chart {
3. Graph statement (layout, rank direction)
4. Nodes statements (create nodes)
5. Edges statements (gives links between nodes)
6. Close the instructions }")

### Simple examples

Below are two simple examples

A very minimal example:

An example with perhaps a bit more applied public health context:

### Syntax

**Basic syntax**

Node names, or edge statements, can be separated with spaces, semicolons, or newlines.

**Rank direction**

A plot can be re-oriented to move left-to-right by adjusting the rankdir argument within the graph statement. The default is TB (top-to-bottom), but it can be LR (left-to-right), RL, or BT.

**Node names**

Node names can be single words, as in the simple example above. To use multi-word names or special characters (e.g. parentheses, dashes), put the node name within single quotes (’ ’). It may be easier to have a short node name, and assign a label, as shown below within brackets [ ]. If you want to have a newline within the node’s name, you must do it via a label - use \n in the node label within single quotes, as shown below.

**Subgroups**  
Within edge statements, subgroups can be created on either side of the edge with curly brackets ({ }). The edge then applies to all nodes in the bracket - it is a shorthand.

**Layouts**

* dot (set rankdir to either TB, LR, RL, BT, )
* neato
* twopi
* circo

**Nodes - editable attributes**

* label (text, in single quotes if multi-word)
* fillcolor (many possible colors)
* fontcolor
* alpha (transparency 0-1)
* shape (ellipse, oval, diamond, egg, plaintext, point, square, triangle)
* style
* sides
* peripheries
* fixedsize (h x w)
* height
* width
* distortion
* penwidth (width of shape border)
* x (displacement left/right)
* y (displacement up/down)
* fontname
* fontsize
* icon

**Edges - editable attributes**

* arrowsize
* arrowhead (normal, box, crow, curve, diamond, dot, inv, none, tee, vee)
* arrowtail
* dir (direction, )
* style (dashed, …)
* color
* alpha
* headport (text in front of arrowhead)
* tailport (text in behind arrowtail)
* fontname
* fontsize
* fontcolor
* penwidth (width of arrow)
* minlen (minimum length)

**Color names**: hexadecimal values or ‘X11’ color names, see [here for X11 details](http://rich-iannone.github.io/DiagrammeR/graphviz_and_mermaid.html)

### Complex examples

The example below expands on the surveillance\_diagram, adding complex node names, grouped edges, colors and styling

DiagrammeR::grViz(" # All instructions are within a large character string

digraph surveillance\_diagram { # 'digraph' means 'directional graph', then the graph name

# graph statement

#################

graph [layout = dot,

rankdir = TB, # layout top-to-bottom

fontsize = 10]

# nodes (circles)

#################

node [shape = circle, # shape = circle

fixedsize = true

width = 1.3]

Primary [label = 'Primary\nFacility']

Secondary [label = 'Secondary\nFacility']

Tertiary [label = 'Tertiary\nFacility']

SC [label = 'Surveillance\nCoordination',

fontcolor = darkgreen]

# edges

#######

Primary -> Secondary [label = ' case transfer',

fontcolor = red,

color = red]

Secondary -> Tertiary [label = ' case transfer',

fontcolor = red,

color = red]

# grouped edge

{Primary Secondary Tertiary} -> SC [label = 'case reporting',

fontcolor = darkgreen,

color = darkgreen,

style = dashed]

}

")

**Sub-graph clusters**

To group nodes into boxed clusters, put them within the same named subgraph (subgraph name {}). To have each subgraph identified within a bounding box, begin the name of the subgraph with “cluster”, as shown with the 4 boxes below.

DiagrammeR::grViz(" # All instructions are within a large character string

digraph surveillance\_diagram { # 'digraph' means 'directional graph', then the graph name

# graph statement

#################

graph [layout = dot,

rankdir = TB,

overlap = true,

fontsize = 10]

# nodes (circles)

#################

node [shape = circle, # shape = circle

fixedsize = true

width = 1.3] # width of circles

subgraph cluster\_passive {

Primary [label = 'Primary\nFacility']

Secondary [label = 'Secondary\nFacility']

Tertiary [label = 'Tertiary\nFacility']

SC [label = 'Surveillance\nCoordination',

fontcolor = darkgreen]

}

# nodes (boxes)

###############

node [shape = box, # node shape

fontname = Helvetica] # text font in node

subgraph cluster\_active {

Active [label = 'Active\nSurveillance']

HCF\_active [label = 'HCF\nActive Search']

}

subgraph cluster\_EBD {

EBS [label = 'Event-Based\nSurveillance (EBS)']

'Social Media'

Radio

}

subgraph cluster\_CBS {

CBS [label = 'Community-Based\nSurveillance (CBS)']

RECOs

}

# edges

#######

{Primary Secondary Tertiary} -> SC [label = 'case reporting']

Primary -> Secondary [label = 'case transfer',

fontcolor = red]

Secondary -> Tertiary [label = 'case transfer',

fontcolor = red]

HCF\_active -> Active

{'Social Media' Radio} -> EBS

RECOs -> CBS

}

")

**Node shapes**

The example below, borrowed from [this tutorial](http://rich-iannone.github.io/DiagrammeR/), shows applied node shapes and a shorthand for serial edge connections

### Outputs

How to handle and save outputs

* Outputs will appear in RStudio’s Viewer pane, by default in the lower-right alongside Files, Plots, Packages, and Help.
* To export you can “Save as image” or “Copy to clipboard” from the Viewer. The graphic will adjust to the specified size.

### Parameterized figures

Here is a quote from this tutorial: <https://mikeyharper.uk/flowcharts-in-r-using-diagrammer/>

“Parameterized figures: A great benefit of designing figures within R is that we are able to connect the figures directly with our analysis by reading R values directly into our flowcharts. For example, suppose you have created a filtering process which removes values after each stage of a process, you can have a figure show the number of values left in the dataset after each stage of your process. To do this we, you can use the @@X symbol directly within the figure, then refer to this in the footer of the plot using [X]:, where X is the a unique numeric index.”

We encourage you to review this tutorial if parameterization is something you are interested in.

## Alluvial/Sankey Diagrams

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

We load the **networkD3** package to produce the diagram, and also **tidyverse** for the data preparation steps.

### Plotting from dataset

Plotting the connections in a dataset. Below we demonstrate using this package on the case linelist. Here is an [online tutorial](https://www.r-graph-gallery.com/321-introduction-to-interactive-sankey-diagram-2.html).

We begin by getting the case counts for each unique age category and hospital combination. We’ve removed values with missing age category for clarity. We also re-label the hospital and age\_cat columns as source and target respectively. These will be the two sides of the alluvial diagram.

The dataset now look like this:

Now we create a data frame of all the diagram nodes, under the column name. This consists of all the values for hospital and age\_cat. Note that we ensure they are all class Character before combining them. and adjust the ID columns to be numbers instead of labels:

The we edit the links data frame, which we created above with count(). We add two numeric columns IDsource and IDtarget which will actually reflect/create the links between the nodes. These columns will hold the rownumbers (position) of the source and target nodes. 1 is subtracted so that these position numbers begin at 0 (not 1).

The links dataset now looks like this:

Now plot the Sankey diagram with sankeyNetwork(). You can read more about each argument by running ?sankeyNetwork in the console. Note that unless you set iterations = 0 the order of your nodes may not be as expected.

Here is an example where the patient Outcome is included as well. Note in the data preparation step we have to calculate the counts of cases between age and hospital, and separately between hospital and outcome - and then bind all these counts together with bind\_rows().

<https://www.displayr.com/sankey-diagrams-r/>

## Event timelines

To make a timeline showing specific events, you can use the vistime package.

See this [vignette](https://cran.r-project.org/web/packages/vistime/vignettes/vistime-vignette.html#ex.-2-project-planning)

Here is the events dataset we begin with:

## DAGs

You can build a DAG manually using the **DiagammeR** package and DOT language as described above.

Alternatively, there are packages like **ggdag** and **daggity**

[Introduction to DAGs ggdag vignette](https://cran.r-project.org/web/packages/ggdag/vignettes/intro-to-dags.html)

[Causal inference with dags in R](https://www.r-bloggers.com/2019/08/causal-inference-with-dags-in-r/#:~:text=In%20a%20DAG%20all%20the,for%20drawing%20and%20analyzing%20DAGs.)

## Resources

Much of the above regarding the DOT language is adapted from the tutorial [at this site](https://mikeyharper.uk/flowcharts-in-r-using-diagrammer/)

Another more in-depth [tutorial on DiagammeR](http://rich-iannone.github.io/DiagrammeR/)

This page on [Sankey diagrams](https://www.displayr.com/sankey-diagrams-r/)

# Combinations analysis

This analysis plots the frequency of different **combinations** of values/responses. In this example, we plot the frequency at which cases exhibited various combinations of symptoms.

This analysis is also often called:

* **“Multiple response analysis”**
* **“Sets analysis”**
* **“Combinations analysis”**

In the example plot above, five symptoms are shown. Below each vertical bar is a line and dots indicating the combination of symptoms reflected by the bar above. To the right, horizontal bars reflect the frequency of each individual symptom.

The first method we show uses the package **ggupset**, and the second uses the package **UpSetR**.

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

To begin, we import the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

This linelist includes five “yes/no” variables on reported symptoms. We will need to transform these variables a bit to use the **ggupset** package to make our plot. View the data (scroll to the right to see the symptoms variables).

### Re-format values

To align with the format expected by **ggupset** we convert the “yes” and “no” the the actual symptom name, using case\_when() from **dplyr**. If “no”, we set the value as blank, so the values are eiter NA or the symptom.

Now we make two final columns:

1. Concatenating (gluing together) all the symptoms of the patient (a character column)
2. Convert the above column to class list, so it can be accepted by **ggupset** to make the plot

See the page on [Characters and strings](#characters-and-strings) to learn more about the unite() function from **stringr**

View the new data. Note the two columns towards the right end - the pasted combined values, and the list

## ****ggupset****

Load the package

Create the plot. We begin with a ggplot() and geom\_bar(), but then we add the special function scale\_x\_upset() from the **ggupset**.

More information on **ggupset** can be found [online](https://rdrr.io/cran/ggupset/man/scale_x_upset.html) or offline in the package documentation in your RStudio Help tab ?ggupset.

## UpSetR

The **UpSetR** package allows more customization of the plot, but it can be more difficult to execute:

**Load package**

**Data cleaning**

We must convert the linelist symptoms values to 1 / 0.

Now make the plot using the custom function upset() - using only the symptoms columns. You must designate which “sets” to compare (the names of the symptom columns). Alternatively, use nsets = and order.by = "freq" to only show the top X combinations.

## Resources

[The github page on UpSetR](https://github.com/hms-dbmi/UpSetR)

[A Shiny App version - you can upload your own data](https://gehlenborglab.shinyapps.io/upsetr/)

[\*documentation - difficult to interpret](https://cran.r-project.org/web/packages/UpSetR/UpSetR.pdf)

# Transmission chains

## Overview

The primary tool to handle, analyse and visualise transmission chains and contact tracing data is the package **epicontacts**, developed by the folks at RECON. Try out the interactive plot below by hovering over the nodes for more information, dragging them to move them and clicking on them to highlight downstream cases.

## Preparation

### Load packages

First load the standard packages required for data import and manipulation. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

You will require the development version of **epicontacts**, which can be installed from github using the p\_install\_github() function from **pacman**. You only need to run this command below once, not every time you use the package (thereafter, you can use p\_load() as usual).

### Import data

We import the dataset of cases from a simulated Ebola epidemic. If you want to download the data to follow step-by-step, see instructions in the [Download handbook and data](#download-handbook-and-data) page. The dataset is imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data.

The first 50 rows of the linelist are displayed below. Of particular interest are the columns case\_id, generation, infector, and source.

### Creating an epicontacts object

We then need to create an **epicontacts** object, which requires two types of data:

* a linelist documenting cases where columns are variables and rows correspond to unique cases
* a list of edges defining links between cases on the basis of their unique IDs (these can be contacts, transmission events, etc.)

As we already have a linelist, we just need to create a list of edges between cases, more specifically between their IDs. We can extract transmission links from the linelist by linking the infector column with the case\_id column. At this point we can also add “edge properties”, by which we mean any variable describing the link between the two cases, not the cases themselves. For illustration, we will add a location variable describing the location of the transmission event, and a duration variable describing the duration of the contact in days.

In the code below, the **dplyr** function transmute is similar to mutate, except it only keeps the columns we have specified within the function. The drop\_na function will filter out any rows where the specified columns have an NA value; in this case, we only want to keep the rows where the infector is known.

We can now create the **epicontacts** object using the make\_epicontacts function. We need to specify which column in the linelist points to the unique case identifier, as well as which columns in the contacts point to the unique identifiers of the cases involved in each link. These links are directional in that infection is going from the infector to the case, so we need to specify the from and to arguments accordingly. We therefore also set the directed argument to TRUE, which will affect future operations.

Upon examining the **epicontacts** objects, we can see that the case\_id column in the linelist has been renamed to id and the case\_id and infector columns in the contacts have been renamed to from and to. This ensures consistency in subsequent handling, visualisation and analysis operations.

## Handling

### Subsetting

The subset() method for epicontacts objects allows for, among other things, filtering of networks based on properties of the linelist (“node attributes”) and the contacts database (“edge attributes”). These values must be passed as named lists to the respective argument. For example, in the code below we are keeping only the male cases in the linelist that have an infection date between April and July 2014 (dates are specified as ranges), and transmission links that occured in the hospital.

We can use the thin function to either filter the linelist to include cases that are found in the contacts by setting the argument what = "linelist", or filter the contacts to include cases that are found in the linelist by setting the argument what = "contacts". In the code below, we are further filtering the epicontacts object to keep only the transmission links involving the male cases infected between April and July which we had filtered for above. We can see that only two known transmission links fit that specification.

In addition to subsetting by node and edge attributes, networks can be pruned to only include components that are connected to certain nodes. The cluster\_id argument takes a vector of case IDs and returns the linelist of individuals that are linked, directly or indirectly, to those IDs. In the code below, we can see that a total of 13 linelist cases are involved in the clusters containing 2ae019 and 71577a.

The subset() method for epicontacts objects also allows filtering by cluster size using the cs, cs\_min and cs\_max arguments. In the code below, we are keeping only the cases linked to clusters of 10 cases or larger, and can see that 271 linelist cases are involved in such clusters.

### Accessing IDs

The get\_id() function retrieves information on case IDs in the dataset, and can be parameterized as follows:

* **linelist**: IDs in the line list data
* **contacts**: IDs in the contact dataset (“from” and “to” combined)
* **from**: IDs in the “from” column of contact datset
* **to** IDs in the “to” column of contact dataset
* **all**: IDs that appear anywhere in either dataset
* **common**: IDs that appear in both contacts dataset and line list

For example, what are the first ten IDs in the contacts dataset?

How many IDs are found in both the linelist and the contacts?

## Visualization

### Basic plotting

All visualisations of **epicontacts** objects are handled by the plot function. We will first filter the **epicontacts** object to include only the cases with onset dates in June 2014 using the subset function, and only include the contacts linked to those cases using the thin function.

We can then create the basic, interactive plot very simply as follows:

You can move the nodes around by dragging them, hover over them for more information and click on them to highlight connected cases.

There are a large number of arguments to further modify this plot. We will cover the main ones here, but check out the documentation via ?vis\_epicontacts (the function called when using plot on an **epicontacts** object) to get a full description of the function arguments.

#### Visualising node attributes

Node color, node shape and node size can be mapped to a given column in the linelist using the node\_color, node\_shape and node\_size arguments. This is similar to the aes syntax you may recognise from **ggplot2**.

The specific colors, shapes and sizes of nodes can be specified as follows:

* **Colors** via the col\_pal argument, either by providing a name list for manual specification of each color as done below, or by providing a color palette function such as colorRampPalette(c("black", "red", "orange")), which would provide a gradient of colours between the ones specified.
* **Shapes** by passing a named list to the shapes argument, specifying one shape for each unique element in the linelist column specified by the node\_shape argument. See codeawesome for available shapes.
* **Size** by passing a size range of the nodes to the size\_range argument.

Here an example, where color represents the outcome, shape the gender and size the age:

#### Visualising edge attributes

Edge color, width and linetype can be mapped to a given column in the contacts dataframe using the edge\_color, edge\_width and edge\_linetype arguments. The specific colors and widths of the edges can be specified as follows:

* **Colors** via the edge\_col\_pal argument, in the same manner used for col\_pal.
* **Widths** by passing a size range of the nodes to the width\_range argument.

Here an example:

### Temporal axis

We can also visualise the network along a temporal axis by mapping the x\_axis argument to a column in the linelist. In the example below, the x-axis represents the date of symptom onset. We have also specified the arrow\_size argument to ensure the arrows are not too large, and set label = FALSE to make the figure less cluttered.

There are a large number of additional arguments to futher specify how this network is visualised along a temporal axis, which you can check out via ?vis\_temporal\_interactive (the function called when using plot on an **epicontacts** object with x\_axis specified). We’ll go through some below.

#### Specifying transmission tree shape

There are two main shapes that the transmission tree can assume, specified using the network\_shape argument. The first is a branching shape as shown above, where a straight edge connects any two nodes. This is the most intuitive representation, however can result in overlapping edges in a densely connected network. The second shape is rectangle, which produces a tree resembling a phylogeny. For example:

Each case node can be assigned a unique vertical position by toggling the position\_dodge argument. The position of unconnected cases (i.e. with no reported contacts) is specified using the unlinked\_pos argument.

The position of the parent node relative to the children nodes can be specified using the parent\_pos argument. The default option is to place the parent node in the middle, however it can be placed at the bottom (parent\_pos = 'bottom') or at the top (parent\_pos = 'top').

#### Saving plots and figures

You can save a plot as an interactive, self-contained html file with the visSave function from the **VisNetwork** package:

Saving these network outputs as an image is unfortunately less easy and requires you to save the file as an html and then take a screenshot of this file using the webshot package. In the code below, we are converting the html file saved above into a PNG:

### Timelines

You can also case timelines to the network, which are represented on the x-axis of each case. This can be used to visualise case locations, for example, or time to outcome. To generate a timeline, we need to create a data.frame of at least three columns indicating the case ID, the start date of the “event” and the end of date of the “event”. You can also add any number of other columns which can then be mapped to node and edge properties of the timeline. In the code below, we generate a timeline going from the date of symptom onset to the date of outcome, and keep the outcome and hospital variables which we use to define the node shape and colour. Note that you can have more than one timeline row/event per case, for example if a case is transferred between multiple hospitals.

We then pass the timeline element to the timeline argument. We can map timeline attributes to timeline node colours, shapes and sizes in the same way defined in previous sections, except that we have two nodes: the start and end node of each timeline, which have seperate arguments. For example, tl\_start\_node\_color defines which timeline column is mapped to the colour of the start node, while tl\_end\_node\_shape defines which timeline column is mapped to the shape of the end node. We can also map colour, width, linetype and labels to the timeline edge via the tl\_edge\_\* arguments.

See ?vis\_temporal\_interactive (the function called when plotting an epicontacts object) for detailed documentation on the arguments. Each argument is annotated in the code below too:

## Analysis

### Summarising

We can get an overview of some of the network properties using the summary function.

For example, we can see that only 57% of contacts have both cases in the linelist; this means that the we do not have linelist data on a significant number of cases involved in these transmission chains.

### Pairwise characteristics

The get\_pairwise() function allows processing of variable(s) in the line list according to each pair in the contact dataset. For the following example, date of onset of disease is extracted from the line list in order to compute the difference between disease date of onset for each pair. The value that is produced from this comparison represents the **serial interval (si)**.

The get\_pairwise() will interpret the class of the column being used for comparison, and will adjust its method of comparing the values accordingly. For numbers and dates (like the **si** example above), the function will subtract the values. When applied to columns that are characters or categorical, get\_pairwise() will paste values together. Because the function also allows for arbitrary processing (see “f” argument), these discrete combinations can be easily tabulated and analyzed.

Here, we see a significant association between transmission links and gender.

### Identifying clusters

The get\_clusters() function can be used for to identify connected components in an epicontacts object. First, we use it to retrieve a data.frame containing the cluster information:

Let us look at the largest clusters. For this, we add cluster information to the epicontacts object and then subset it to keep only the largest clusters:

### Calculating degrees

The degree of a node corresponds to its number of edges or connections to other nodes. get\_degree() provides an easy method for calculating this value for epicontacts networks. A high degree in this context indicates an individual who was in contact with many others. The type argument indicates that we want to count both the in-degree and out-degree, the only\_linelist argument indicates that we only want to calculate the degree for cases in the linelist.

Which individuals have the ten most contacts?

What is the mean number of contacts?

## Resources

The [epicontacts page](https://www.repidemicsconsortium.org/epicontacts/index.html) provides an overview of the package functions and includes some more in-depth vignettes.

The [github page](http://github.com/reconhub/epicontacts) can be used to raise issues and request features.

# Phylogenetic trees

## Overview

**Phylogenetic trees** are used to visualize and describe the relatedness and evolution of organisms based on the sequence of their genetic code.

They can be constructed from genetic sequences using distance-based methods (such as neighbor-joining method) or character-based methods (such as maximum likelihood and Bayesian Markov Chain Monte Carlo method). Next-generation sequencing (NGS) has become more affordable and is becoming more widely used in public health to describe pathogens causing infectious diseases. Portable sequencing devices decrease the turn around time and hold promises to make data available for the support of outbreak investigation in real-time. NGS data can be used to identify the origin or source of an outbreak strain and its propagation, as well as determine presence of antimicrobial resistance genes. To visualize the genetic relatedness between samples a phylogenetic tree is constructed.

In this page we will learn how to use the **ggtree** package, which allows for combined visualization of phylogenetic trees with additional sample data in form of a dataframe. This will enable us to observe patterns and improve understanding of the outbreak dynamic.

## Preparation

### Load packages

This code chunk shows the loading of required packages. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

The data for this page can be downloaded with the instructions on the [Download handbook and data](#download-handbook-and-data) page.

There are several different formats in which a phylogenetic tree can be stored (eg. Newick, NEXUS, Phylip). A common one is the Newick file format (.nwk), which is the standard for representing trees in computer-readable form. This means an entire tree can be expressed in a string format such as “((t2:0.04,t1:0.34):0.89,(t5:0.37,(t4:0.03,t3:0.67):0.9):0.59);”, listing all nodes and tips and their relationship (branch length) to each other.

Note: It is important to understand that the phylogenetic tree file in itself does not contain sequencing data, but is merely the result of the genetic distances between the sequences. We therefore cannot extract sequencing data from a tree file.

First, we use the read.tree() function from **ape** package to import a Newick phylogenetic tree file in .txt format, and store it in a list object of class “phylo”. If necessary, use the here() function from the **here** package to specify the relative file path.

Note: In this case the newick tree is saved as a .txt file for easier handling and downloading from Github.

tree <- ape::read.tree("Shigella\_tree.txt")

We inspect our tree object and see it contains 299 tips (or samples) and 236 nodes.

Second, we import a table stored as a .csv file with additional information for each sequenced sample, such as gender, country of origin and attributes for antimicrobial resistance, using the import() function from the **rio** package:

sample\_data <- import("sample\_data\_Shigella\_tree.csv")

Below are the first 50 rows of the data:

### Clean and inspect

We clean and inspect our data: In order to assign the correct sample data to the phylogenetic tree, the values in the column Sample\_ID in the sample\_data data frame need to match the tip.labels values in the tree file:

We check the formatting of the tip.labels in the tree file by looking at the first 6 entries using with head() from **base** R.

We also make sure the first column in our sample\_data data frame is Sample\_ID. We look at the column names of our dataframe using colnames() from **base** R.

We look at the Sample\_IDs in the data frame to make sure the formatting is the same than in the tip.label (eg. letters are all capitals, no extra underscores \_ between letters and numbers, etc.)

We can also compare if all samples are present in the tree file and vice versa by generating a logical vector of TRUE or FALSE where they do or do not match. These are not printed here, for simplicity.

We can use these vectors to show any sample IDs that are not on the tree (there are none).

Upon inspection we can see that the format of Sample\_ID in the dataframe corresponds to the format of sample names at the tip.labels. These do not have to be sorted in the same order to be matched.

We are ready to go!

## Simple tree visualization

### Different tree layouts

**ggtree** offers many different layout formats and some may be more suitable for your specific purpose than others. Below are a few demonstrations. For other options see this [online book](http://yulab-smu.top/treedata-book/chapter4.html).

Here are some example tree layouts:

### Simple tree plus sample data

The **%<+%** operator is used to connect the sample\_data data frame to the tree file. The most easy annotation of your tree is the addition of the sample names at the tips, as well as coloring of tip points and if desired the branches:

Here is an example of a circular tree:

You can export your tree plot with ggsave() as you would any other ggplot object. Written this way, ggsave() saves the last image produced to the file path you specify. Remember that you can use here() and relative file paths to easily save in subfolders, etc.

## Tree manipulation

Sometimes you may have a very large phylogenetic tree and you are only interested in one part of the tree. For example, if you produced a tree including historical or international samples to get a large overview of where your dataset might fit in the bigger picture. But then to look closer at your data you want to inspect only that portion of the bigger tree.

Since the phylogenetic tree file is just the output of sequencing data analysis, we can not manipulate the order of the nodes and branches in the file itself. These have already been determined in previous analysis from the raw NGS data. We are able though to zoom into parts, hide parts and even subset part of the tree.

### Zoom in

If you don’t want to “cut” your tree, but only inspect part of it more closely you can zoom in to view a specific part.

First, we plot the entire tree in linear format and add numeric labels to each node in the tree.

To zoom in to one particular branch (sticking out to the right), use viewClade() on the ggtree object p and provide the node number to get a closer look:

### Collapsing branches

However, we may want to ignore this branch and can collapse it at that same node (node nr. 452) using collapse(). This tree is defined as p\_collapsed.

For clarity, when we print p\_collapsed, we add a geom\_point2() (a blue diamond) at the node of the collapsed branch.

### Subsetting a tree

If we want to make a more permanent change and create a new, reduced tree to work with we can subset part of it with tree\_subset(). Then you can save it as new newick tree file or .txt file.

First, we inspect the tree nodes and tip labels in order to decide what to subset.

Now, say we have decided to subset the tree at node 528 (keep only tips within this branch after node 528) and we save it as a new sub\_tree1 object:

Lets have a look at the subset tree 1:

You can also subset based on one particular sample, specifying how many nodes “backwards” you want to include. Let’s subset the same part of the tree based on a sample, in this case S17BD07692, going back 9 nodes and we save it as a new sub\_tree2 object:

Lets have a look at the subset tree 2:

You can also save your new tree either as a Newick type or even a text file using the write.tree() function from **ape** package:

### Rotating nodes in a tree

As mentioned before we cannot change the order of tips or nodes in the tree, as this is based on their genetic relatedness and is not subject to visual manipulation. But we can rote branches around nodes if that eases our visualization.

First, we plot our new subset tree 2 with node labels to choose the node we want to manipulate and store it an a ggtree plot object p.

We can then manipulate nodes by applying **ggtree::rotate()** or **ggtree::flip()**: Note: to illustrate which nodes we are manipulating we first apply the **geom\_hilight()** function from **ggtree** to highlight the samples in the nodes we are interested in and store that ggtree plot object in a new object p1.

Now we can rotate node 37 in object p1 so that the samples on node 38 move to the top. We store the rotated tree in a new object p2.

Or we can use the flip command to rotate node 36 in object p1 and switch node 37 to the top and node 39 to the bottom. We store the flipped tree in a new object p3.

### Example subtree with sample data annotation

Lets say we are investigating the cluster of cases with clonal expansion which occurred in 2017 and 2018 at node 39 in our sub-tree. We add the year of strain isolation as well as travel history and color by country to see origin of other closely related strains:

Our observation points towards an import event of strains from Asia, which then circulated in Belgium over the years and seem to have caused our latest outbreak.

## More complex trees: adding heatmaps of sample data

We can add more complex information, such as categorical presence of antimicrobial resistance genes and numeric values for actually measured resistance to antimicrobials in form of a heatmap using the **ggtree::gheatmap()** function.

First we need to plot our tree (this can be either linear or circular) and store it in a new ggtree plot object p: We will use the sub\_tree from part 3.)

Second, we prepare our data. To visualize different variables with new color schemes, we subset our dataframe to the desired variable. It is important to add the Sample\_ID as rownames otherwise it cannot match the data to the tree tip.labels:

In our example we want to look at gender and mutations that could confer resistance to Ciprofloxacin, an important first line antibiotic used to treat Shigella infections.

We create a dataframe for gender:

We create a dataframe for mutations in the gyrA gene, which confer Ciprofloxacin resistance:

We create a dataframe for the measured minimum inhibitory concentration (MIC) for Ciprofloxacin from the laboratory:

We create a first plot adding a binary heatmap for gender to the phylogenetic tree and storing it in a new ggtree plot object h1:

Then we add information on mutations in the gyrA gene, which confer resistance to Ciprofloxacin:

Note: The presence of chromosomal point mutations in WGS data was prior determined using the PointFinder tool developed by Zankari et al. (see reference in the additional references section)

First, we assign a new color scheme to our existing plot object h1 and store it in a now object h2. This enables us to define and change the colors for our second variable in the heatmap.

Then we add the second heatmap layer to h2 and store the combined plots in a new object h3:

We repeat the above process, by first adding a new color scale layer to our existing object h3, and then adding the continuous data on the minimum inhibitory concentration (MIC) of Ciprofloxacin for each strain to the resulting object h4 to produce the final object h5:

We can do the same exercise for a linear tree:

First we add gender:

Then we add Ciprofloxacin resistance mutations after adding another color scheme layer:

Then we add the minimum inhibitory concentration determined by the laboratory (MIC):

## Resources

[http://hydrodictyon.eeb.uconn.edu/eebedia/index.php/Ggtree#](http://hydrodictyon.eeb.uconn.edu/eebedia/index.php/Ggtree) Clade\_Colors <https://bioconductor.riken.jp/packages/3.2/bioc/vignettes/ggtree/inst/doc/treeManipulation.html> <https://guangchuangyu.github.io/ggtree-book/chapter-ggtree.html> <https://bioconductor.riken.jp/packages/3.8/bioc/vignettes/ggtree/inst/doc/treeManipulation.html>

Ea Zankari, Rosa Allesøe, Katrine G Joensen, Lina M Cavaco, Ole Lund, Frank M Aarestrup, PointFinder: a novel web tool for WGS-based detection of antimicrobial resistance associated with chromosomal point mutations in bacterial pathogens, Journal of Antimicrobial Chemotherapy, Volume 72, Issue 10, October 2017, Pages 2764–2768, <https://doi.org/10.1093/jac/dkx217>

# Interactive plots

Data visualisation is increasingly required to be interrogable by the audience. Consequently, is is becoming common to create interactive plots. There are several ways to include these but the two most common are **plotly** and **shiny**.

In this page we will focus on converting an existing ggplot() plot into an interactive plot with **plotly**. You can read more about **shiny** in the [Dashboards with Shiny](#dashboards-with-shiny) page. What is worth mentioning is that interactive plots are only useable in HTML format R markdown documents, not PDF or Word documents.

Below is a basic epicurve that has been transformed to be interactive using the integration of **ggplot2** and **plotly** (hover your mouse over the plot, zoom in, or click items in the legend).

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Start with a ggplot()

In this page we assume that you are beginning with a ggplot() plot that you want to convert to be interactive. We will build several of these plots in this page, using the case linelist used in many pages of this handbook.

### Import data

To begin, we import the cleaned linelist of cases from a simulated Ebola epidemic. If you want to follow along, [click to download the “clean” linelist](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Plot with ggplotly()

The function ggplotly() from the **plotly** package makes it easy to convert a ggplot() to be interactive. Simply save your ggplot() and then pipe it to the ggplotly() function.

Below, we plot a simple line representing the proportion of cases who died in a given week:

We begin by creating a summary dataset of each epidemiological week, and the percent of cases with a known outcome that died.

Here is the first 50 rows of the weekly\_deaths dataset.

Then we create the plot with **ggplot2**, using geom\_line().

We can make this interactive by simply passing this plot to ggplotly(), as below. Hover your mouse over the line to show the x and y values. You can zoom in on the plot, and drag it around. You can also see icons in the upper-right of the plot. In order, they allow you to:

* Download the current view as a PNG image
* Zoom in with a select box
* “Pan”, or move across the plot by clicking and dragging the plot
* Zoom in, zoom out, or return to default zoom
* Reset axes to defaults
* Toggle on/off “spike lines” which are dotted lines from the interactive point extending to the x and y axes
* Adjustments to whether data show when you are not hovering on the line

Grouped data work with ggplotly() as well. Below, a weekly epicurve is made, grouped by outcome. The stacked bars are interactive. Try clicking on the different items in the legend (they will appear/disappear).

# Plot interactively

p %>% plotly::ggplotly()

## Modifications

### File size

When exporting in an R Markdown generated HTML (like this book!) you want to make the plot as small data size as possible (with no negative side effects in most cases). For this, just pipe the interactive plot to partial\_bundle(), also from **plotly**.

### Buttons

Some of the buttons on a standard plotly are superfluous and can be distracting, so you can remove them. You can do this simply by piping the output into config() from **plotly** and specifying which buttons to remove. In the below example we specify in advance the names of the buttons to remove, and provide them to the argument modeBarButtonsToRemove =. We also set displaylogo = FALSE to remove the plotly logo.

## Heat tiles

You can make almost any ggplot() plot interactive, including heat tiles. In the page on [Heat plots](#heat-plots) you can read about how to make the below plot, which displays the proportion of days per week that certain facilities reported data to their province.

Here is the code, although we will not describe it in depth here.

Below, we make it interactive and modify it for simple buttons and file size.

–>

## Resources

Plotly is not just for R, but also works well with Python (and really any data science language as it’s built in JavaScript). You can read more about it on the [plotly website](https://plotly.com/r/)

# VI Reports and dashboards

# Reports with R Markdown

R Markdown is a widely-used tool for creating automated, reproducible, and share-worthy outputs, such as reports. It can generate static or interactive outputs, in Word, pdf, html, powerpoint, and other formats.

An R Markdown script intersperces R code and text such that the script actually becomes your output document. You can create an entire formatted document, including narrative text (can be dynamic to change based on your data), tables, figures, bullets/numbers, bibliographies, etc.

Such documents can be produced to update on a routine basis (e.g. daily surveillance reports) and/or run on subsets of data (e.g. reports for each jurisdiction).

Other pages in this handbook expand on this topic:

* The page [Organizing routine reports](#organizing-routine-reports) demonstrates how to routinize your report production with auto-generated time-stamped folders.
* The page [Dashboards with R Markdown](#dashboards-with-r-markdown) explains how to format a R Markdown report as a dashboard.

Of note, the [R4Epis](https://r4epis.netlify.app/) project has developed template R Markdown scripts for common outbreaks and surveys scenarios encountered at MSF project locations.

## Preparation

**Background to R Markdown**

To explain some of the concepts and packages involved:

* **Markdown** is a “language” that allows you to write a document using plain text, that can be converted to html and other formats. It is not specific to R. Files written in Markdown have a ‘.md’ extension.
* **R Markdown**: is a variation on markdown that is specific to R - it allows you to write a document using markdown to produce text and to embed R code and display their outputs. R Markdown files have ‘.Rmd’ extension.
* **rmarkdown - the package**: This is used by R to render the .Rmd file into the desired output. It’s focus is converting the markdown (text) syntax, so we also need…
* **knitr**: This R package will read the code chunks, execute it, and ‘knit’ it back into the document. This is how tables and graphs are included alongside the text.
* **Pandoc**: Finally, pandoc actually convert the output into word/pdf/powerpoint etc. It is a software separate from R but is installed automatically with RStudio.

In sum, the process that happens in the background (you do not need to know all these steps!) involves feeding the .Rmd file to **knitr**, which executes the R code chunks and creates a new .md (markdown) file which includes the R code and its rendered output. The .md file is then processed by pandoc to create the finished product: a Microsoft Word document, HTML file, powerpoint document, pdf, etc.

(source: <https://rmarkdown.rstudio.com/authoring_quick_tour.html>):

**Installation**

To create a R Markdown output, you need to have the following installed:

* The **rmarkdown** package (**knitr** will also be installed automatically)
* Pandoc, which should come installed with RStudio. If you are not using RStudio, you can download Pandoc here: <http://pandoc.org>.
* If you want to generate PDF output (a bit trickier), you will need to install LaTeX. For R Markdown users who have not installed LaTeX before, we recommend that you install TinyTeX (<https://yihui.name/tinytex/>). You can use the following commands:

## Getting started

### Install rmarkdown R package

Install the **rmarkdown** R package. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Starting a new Rmd file

In RStudio, open a new R markdown file, starting with ‘File’, then ‘New file’ then ‘R markdown…’.

R Studio will give you some output options to pick from. In the example below we select “HTML” because we want to create an html document. The title and the author names are not important. If the output document type you want is not one of these, don’t worry - you can just pick any one and change it in the script later.

This will open up a new .Rmd script.

### Important to know

**The working directory**

The working directory of a markdown file is wherever the Rmd file itself is saved. For instance, if the R project is within ~/Documents/projectX and the Rmd file itself is in a subfolder ~/Documents/projectX/markdownfiles/markdown.Rmd, the code read.csv(“data.csv”) within the markdown will look for a csv file in the markdownfiles folder, and not the root project folder where scripts within projects would normally automatically look.

To refer to files elsewhere, you will either need to use the full file path or use the **here** package. The **here** package sets the working directory to the root folder of the R project and is explained in detail in the [R projects](#r-projects) and [Import and export](#import-and-export) pages of this handbook. For instance, to import a file called “data.csv” from within the projectX folder, the code would be import(here(“data.csv”)).

Note that use of setwd() in R Markdown scripts is not recommended – it only applies to the code chunk that it is written in.

**Working on a drive vs your computer**

Because R Markdown can run into pandoc issues when running on a shared network drive, it is recommended that your folder is on your local machine, e.g. in a project within ‘My Documents’. If you use Git (much recommended!), this will be familiar. For more details, see the handbook pages on [R on network drives](#r-on-network-drives) and [Errors and help].

## R Markdown components

An R Markdown document can be edited in RStudio just like a standard R script. When you start a new R Markdown script, RStudio tries to be helpful by showing a template which explains the different section of an R Markdown script.

The below is what appears when starting a new Rmd script intended to produce an html output (as per previous section).

As you can see, there are three basic components to an Rmd file: YAML, Markdown text, and R code chunks.

These will create and become your document output. See the diagram below:

### YAML metadata

Referred to as the ‘YAML metadata’ or just ‘YAML’, this is at the top of the R Markdown document. This section of the script will tell your Rmd file what type of output to produce, formatting preferences, and other metadata such as document title, author, and date. There are other uses not mentioned here (but referred to in ‘Producing an output’). Note that indentation matters; tabs are not accepted but spaces are.

This section must begin with a line containing just three dashes --- and must close with a line containing just three dashes ---. YAML parameters comes in key:value pairs. The placement of colons in YAML is important - the key:value pairs are separated by colons (not equals signs!).

The YAML should begin with metadata for the document. The order of these primary YAML parameters (not indented) does not matter. For example:

title: "My document"

author: "Me"

date: "2021-06-04"

You can use R code in YAML values by writing it as in-line code (preceded by r within back-ticks) but also within quotes (see above example for date:).

In the image above, because we clicked that our default output would be an html file, we can see that the YAML says output: html\_document. However we can also change this to say powerpoint\_presentation or word\_document or even pdf\_document.

### Text

This is the narrative of your document, including the titles and headings. It is written in the “markdown” language, which is used across many different software.

Below are the core ways to write this text. See more extensive documentation available on R Markdown “cheatsheet” at the [RStudio website](https://rstudio.com/resources/cheatsheets/).

#### New lines

Uniquely in R Markdown, to initiate a new line, enter \*two spaces\*\* at the end of the previous line and then Enter/Return.

#### Case

Surround your normal text with these character to change how it appears in the output.

* Underscores (\_text\_) or single asterisk (\*text\*) to italicise
* Double asterisks (\*\*text\*\*) for **bold text**
* Back-ticks (text) to display text as code

The actual appearance of the font can be set by using specific templates (specified in the YAML metadata; see example tabs).

#### Color

There is no simple mechanism to change the color of text in R Markdown. One work-around, IF your output is an HTML file, is to add an HTML line into the markdown text. The below HTML code will print a line of text in bold red.

<span style="color: red;">\*\*\_DANGER:\_\*\* This is a warning.</span>

**DANGER:** This is a warning.

#### Titles and headings

A hash symbol in a text portion of a R Markdown script creates a heading. This is different than in a chunk of R code in the script, in which a hash symbol is a mechanism to comment/annotate/de-activate, as in a normal R script.

Different heading levels are established with different numbers of hash symbols at the start of a new line. One hash symbol is a title or primary heading. Two hash symbols are a second-level heading. Third- and fourth-level headings can be made with successively more hash symbols.

# First-level heading / title

## Second level heading

### Third-level heading

#### Bullets and numbering

Use asterisks (\*) to created a bullets list. Finish the previous sentence, enter two spaces, Enter/Return twice, and then start your bullets. Include a space between the asterisk and your bullet text. After each bullet enter two spaces and then Enter/Return. Sub-bullets work the same way but are indented. Numbers work the same way but instead of an asterisk, write 1), 2), etc. Below is how your R Markdown script text might look.

Here are my bullets (there are two spaces after this colon):

\* Bullet 1 (followed by two spaces and Enter/Return)

\* Bullet 2 (followed by two spaces and Enter/Return)

\* Sub-bullet 1 (followed by two spaces and Enter/Return)

\* Sub-bullet 2 (followed by two spaces and Enter/Return)

#### Comment out text

You can “comment out” R Markdown text just as you can use the “#” to comment out a line of R code in an R chunk. Simply highlight the text and press Ctrl+Shift+c (Cmd+Shift+c for Mac). The text will be surrounded by arrows and turn green. It will not appear in your output.

### Code chunks

Sections of the script that are dedicated to running R code are called “chunks”. This is where you may load packages, import data, and perform the actual data management and visualisation. There may be many code chunks, so they can help you organize your R code into parts, perhaps interspersed with text. To note: These ‘chunks’ will appear to have a slightly different background colour from the narrative part of the document.

Each chunk is opened with a line that starts with three back-ticks, and curly brackets that contain parameters for the chunk ({ }). The chunk ends with three more back-ticks.

You can create a new chunk by typing it out yourself, by using the keyboard shortcut “Ctrl + Alt + i” (or Cmd + Shift + r in Mac), or by clicking the green ‘insert a new code chunk’ icon at the top of your script editor.

Some notes about the contents of the curly brackets { }:

* They start with ‘r’ to indicate that the language name within the chunk is R
* After the r you can optionally write a chunk “name” – these are not necessary but can help you organise your work. Note that if you name your chunks, you should ALWAYS use unique names or else R will complain when you try to render.
* The curly brackets can include other options too, written as tag=value, such as:
* eval = FALSE to not run the R code
* echo = FALSE to not print the chunk’s R source code in the output document
* warning = FALSE to not print warnings produced by the R code
* message = FALSE to not print any messages produced by the R code
* include = either TRUE/FALSE whether to include chunk outputs (e.g. plots) in the document
* out.width = and out.height = - provide in style out.width = "75%"
* fig.align = "center" adjust how a figure is aligned across the page
* fig.show='hold' if your chunk prints multiple figures and you want them printed next to each other (pair with out.width = c("33%", "67%"). Can also set as fig.show='asis' to show them below the code that generates them, 'hide' to hide, or 'animate' to concatenate multiple into an animation.
* A chunk header must be written in one line
* Try to avoid periods, underscores, and spaces. Use hyphens ( - ) instead if you need a separator.

Read more extensively about the **knitr** options [here](https://yihui.org/knitr/options/).

Some of the above options can be configured with point-and-click using the setting buttons at the top right of the chunk. Here, you can specify which parts of the chunk you want the rendered document to include, namely the code, the outputs, and the warnings. This will come out as written preferences within the curly brackets, e.g. echo=FALSE if you specify you want to ‘Show output only’.

There are also two arrows at the top right of each chunk, which are useful to run code within a chunk, or all code in prior chunks. Hover over them to see what they do.

For global options to be applied to all chunks in the script, you can set this up within your very first R code chunk in the script. For instance, so that only the outputs are shown for each code chunk and not the code itself, you can include this command in the R code chunk:

#### In-text R code

You can also include minimal R code within back-ticks. Within the back-ticks, begin the code with “r” and a space, so RStudio knows to evaluate the code as R code. See the example below.

The example below shows multiple heading levels, bullets, and uses R code for the current date (Sys.Date()) to evaluate into a printed date.

The example above is simple (showing the current date), but using the same syntax you can display values produced by more complex R code (e.g. to calculate the min, median, max of a column). You can also integrate R objects or values that were created in R code chunks earlier in the script.

As an example, the script below calculates the proportion of cases that are aged less than 18 years old, using **tidyverse** functions, and creates the objects less18, total, and less18prop. This dynamic value is inserted into subsequent text. We see how it looks when knitted to a word document.

### Images

You can include images in your R Markdown one of two ways:

If the above does not work, try using knitr::include\_graphics()

(remember, your file path could be written using the **here** package)

### Tables

Create a table using hyphens ( - ) and bars ( | ). The number of hyphens before/between bars allow the number of spaces in the cell before the text begins to wrap.

Column 1 |Column 2 |Column 3

---------|----------|--------

Cell A |Cell B |Cell C

Cell D |Cell E |Cell F

The above code produces the table below:

| **Column 1** | **Column 2** | **Column 3** |
| --- | --- | --- |
| Cell A | Cell B | Cell C |
| Cell D | Cell E | Cell F |

### Tabbed sections

For HTML outputs, you can arrange the sections into “tabs”. Simply add .tabset in the curly brackets { } that are placed after a heading. Any sub-headings beneath that heading (until another heading of the same level) will appear as tabs that the user can click through. Read more [here](https://bookdown.org/yihui/rmarkdown-cookbook/html-tabs.html)

You can add an additional option .tabset-pills after .tabset to give the tabs themselves a “pilled” appearance. Be aware that when viewing the tabbed HTML output, the Ctrl+f search functionality will only search “active” tabs, not hidden tabs.

## File structure

There are several ways to structure your R Markdown and any associated R scripts. Each has advantages and disadvantages:

* Self-contained R Markdown - everything needed for the report is imported or created within the R Markdown
  + Source other files - You can run external R scripts with the source() command and use their outputs in the Rmd
  + Child scripts - an alternate mechanism for source()
* Utilize a “runfile” - Run commands in an R script prior to rendering the R Markdown

### Self-contained Rmd

For a relatively simple report, you may elect to organize your R Markdown script such that it is “self-contained” and does not involve any external scripts.

Everything you need to run the R markdown is imported or created within the Rmd file, including all the code chunks and package loading. This “self-contained” approach is appropriate when you do not need to do much data processing (e.g. it brings in a clean or semi-clean data file) and the rendering of the R Markdown will not take too long.

In this scenario, one logical organization of the R Markdown script might be:

1. Set global **knitr** options
2. Load packages
3. Import data
4. Process data
5. Produce outputs (tables, plots, etc.)
6. Save outputs, if applicable (.csv, .png, etc.)

#### Source other files

One variation of the “self-contained” approach is to have R Markdown code chunks “source” (run) other R scripts. This can make your R Markdown script less cluttered, more simple, and easier to organize. It can also help if you want to display final figures at the beginning of the report. In this approach, the final R Markdown script simply combines pre-processed outputs into a document.

One way to do this is by providing the R scripts (file path and name with extension) to the **base** R command source().

Note that when using source() within the R Markdown, the external files will still be run during the course of rendering your Rmd file. Therefore, each script is run every time you render the report. Thus, having these source() commands within the R Markdown does not speed up your run time, nor does it greatly assist with de-bugging, as error produced will still be printed when producing the R Markdown.

An alternative is to utilize the child = **knitr** option. EXPLAIN MORE TO DO

You must be aware of various R environments. Objects created within an environment will not necessarily be available to the environment used by the R Markdown.

### Runfile

This approach involves utilizing the R script that contains the render() command(s) to pre-process objects that feed into the R markdown.

For instance, you can load the packages, load and clean the data, and even create the graphs of interest prior to render(). These steps can occur in the R script, or in other scripts that are sourced. As long as these commands occur in the same RStudio session and objects are saved to the environment, the objects can then be called within the Rmd content. Then the R markdown itself will only be used for the final step - to produce the output with all the pre-processed objects. This is much easier to de-bug if something goes wrong.

This approach is helpful for the following reasons:

* More informative error messages - these messages will be generated from the R script, not the R Markdown. R Markdown errors tend to tell you which chunk had a problem, but will not tell you which line.
* If applicable, you can run long processing steps in advance of the render() command - they will run only once.

In the example below, we have a separate R script in which we pre-process a data object into the R Environment and then render the “create\_output.Rmd” using render().

### Folder strucutre

Workflow also concerns the overall folder structure, such as having an ‘output’ folder for created documents and figures, and ‘data’ or ‘inputs’ folders for cleaned data. We do not go into further detail here, but check out the [Organizing routine reports](#organizing-routine-reports) page.

## Producing the document

You can produce the document in the following ways:

* Manually by pressing the “Knit” button at the top of the RStudio script editor (fast and easy)
* Run the render() command (executed outside the R Markdown script)

### Option 1: “Knit” button

When you have the Rmd file open, press the ‘Knit’ icon/button at the top of the file.

R Studio will you show the progress within an ‘R Markdown’ tab near your R console. The document will automatically open when complete.

The document will be saved in the same folder as your R markdown script, and with the same file name (aside from the extension). This is obviously not ideal for version control (it will be over-written each tim you knit, unless moved manually), as you may then need to rename the file yourself (e.g. add a date).

This is RStudio’s shortcut button for the render() function from **rmarkdown**. This approach only compatible with a self-contained R markdown, where all the needed components exist or are sourced within the file.

### Option 2: render() command

Another way to produce your R Markdown output is to run the render() function (from the **rmarkdown** package). You must execute this command outside the R Markdown script - so either in a separate R script (often called a “run file”), or as a stand-alone command in the R Console.

As with “knit”, the default settings will save the Rmd output to the same folder as the Rmd script, with the same file name (aside from the file extension). For instance “my\_report.Rmd” when knitted will create “my\_report.docx” if you are knitting to a word document. However, by using render() you have the option to use different settings. render() can accept arguments including:

* output\_format = This is the output format to convert to (e.g. "html\_document", "pdf\_document", "word\_document", or "all"). You can also specify this in the YAML inside the R Markdown script.
* output\_file = This is the name of the output file (and file path). This can be created via R functions like here() or str\_glue() as demonstrated below.
* output\_dir = This is an output directory (folder) to save the file. This allows you to chose an alternative other than the directory the Rmd file is saved to.
* output\_options = You can provide a list of options that will override those in the script YAML (e.g. )
* output\_yaml = You can provide path to a .yml file that contains YAML specifications
* params = See the section on parameters below
* See the complete list [here](https://pkgs.rstudio.com/rmarkdown/reference/render.html)

As one example, to improve version control, the following command will save the output file within an ‘outputs’ sub-folder, with the current date in the file name. To create the file name, the function str\_glue() from the **stringr** package is use to ‘glue’ together static strings (written plainly) with dynamic R code (written in curly brackets). For instance if it is April 10th 2021, the file name from below will be “Report\_2021-04-10.docx”. See the page on [Characters and strings](#characters-and-strings) for more details on str\_glue().

As the file renders, the RStudio Console will show you the rendering progress up to 100%, and a final message to indicate that the rendering is complete.

### Options 3: ****reportfactory**** package

The R package **reportfactory** offers an alternative method of organising and compiling R Markdown reports catered to scenarios where you run reports routinely (e.g. daily, weekly…). It eases the compilation of multiple R Markdown files and the organization of their outputs. In essence, it provides a “factory” from which you can run the R Markdown reports, get automatically date- and time-stamped folders for the outputs, and have “light” version control.

Read more about this work flow in the page on [Organizing routine reports](#organizing-routine-reports).

## Parameterised reports

You can use parameterisation to make a report dynamic, such that it can be run with specific setting (e.g. a specific date or place or with certain knitting options). Below, we focus on the basics, but there is more [detail online](https://bookdown.org/yihui/rmarkdown/parameterized-reports.html) about parameterized reports.

Using the Ebola linelist as an example, let’s say we want to run a standard surveillance report for each hospital each day. We show how one can do this using parameters.

Important: dynamic reports are also possible without the formal parameter structure (without *params:*), using simple R objects in an adjacent R script. This is explained at the end of this section.

### Setting parameters

You have several options for specifying parameter values for your R Markdown output.

#### Option 1: Set parameters within YAML

Edit the YAML to include a params: option, with indented statements for each parameter you want to define. In this example we create parameters date and hospital, for which we specify values. These values are subject to change each time the report is run. If you use the “Knit” button to produce the output, the parameters will have these default values. Likewise, if you use render() the parameters will have these default values unless otherwise specified in the render() command.

---

title: Surveillance report

output: html\_document

params:

date: 2021-04-10

hospital: Central Hospital

---

In the background, these parameter values are contained within a read-only list called params. Thus, you can insert the parameter values in R code as you would another R object/value in your environment. Simply type params$ followed by the parameter name. For example params$hospital to represent the hospital name (“Central Hospital” by default).

Note that parameters can also hold values true or false, and so these can be included in your **knitr** options for a R chunk. For example, you can set {r, eval=params$run} instead of {r, eval=FALSE}, and now whether the chunk runs or not depends on the value of a parameter run:.

Note that for parameters that are dates, they will be input as a string. So for params$date to be interpreted in R code it will likely need to be wrapped with as.Date() or a similar function to convert to class Date.

#### Option 2: Set parameters within render()

As mentioned above, as alternative to pressing the “Knit” button to produce the output is to execute the render() function from a separate script. In this later case, you can specify the parameters to be used in that rendering to the params = argument of render().

Note than any parameter values provided here will overwrite their default values if written within the YAML. We write the values in quotation marks as in this case they should be defined as character/string values.

The below command renders “surveillance\_report.Rmd”, specifies a dynamic output file name and folder, and provides a list() of two parameters and their values to the argument params =.

#### Option 3: Set parameters using a Graphical User Interface

For a more interactive feel, you can also use the Graphical User Interface (GUI) to manually select values for parameters. To do this we can click the drop-down menu next to the ‘Knit’ button and choose ‘Knit with parameters’.

A pop-up will appear allowing you to type in values for the parameters that are established in the document’s YAML.

You can achieve the same through a render() command by specifying params = "ask", as demonstrated below.

However, typing values into this pop-up window is subject to error and spelling mistakes. You may prefer to add restrictions to the values that can be entered through drop-down menus. You can do this by adding in the YAML several specifications for each params: entry.

* label: is how the title for that particular drop-down menu
* value: is the default (starting) value
* input: set to select for drop-down menu
* choices: Give the eligible values in the drop-down menu

Below, these specifications are written for the hospital parameter.

---

title: Surveillance report

output: html\_document

params:

date: 2021-04-10

hospital:

label: “Town:”

value: Central Hospital

input: select

choices: [Central Hospital, Military Hospital, Port Hospital, St. Mark's Maternity Hospital (SMMH)]

---

When knitting (either via the ‘knit with parameters’ button or by render()), the pop-up window will have drop-down options to select from.

### Parameterized example

The following code creates parameters for date and hospital, which are used in the R Markdown as params$date and params$hospital, respectively.

In the resulting report output, see how the data are filtered to the specific hospital, and the plot title refers to the correct hospital and date. We use the “linelist\_cleaned.rds” file here, but it would be particularly appropriate if the linelist itself also had a datestamp within it to align with parameterised date.

Knitting this produces the final output with the default font and layout.

### Parameterisation without params

If you are rendering a R Markdown file with render() from a separate script, you can actually create the impact of parameterization without using the params: functionality.

For instance, in the R script that contains the render() command, you can simply define hospital and date as two R objects (values) before the render() command. In the R Markdown, you would not need to have a params: section in the YAML, and we would refer to the date object rather than params$date and hospital rather than params$hospital.

Following this approach means means you can not “knit with parameters”, use the GUI, or include knitting options within the parameters. However it allows for simpler code, which may be advantageous.

## Looping reports

We may want to run a report multiple times, varying the input parameters, to produce a report for each jurisdictions/unit. This can be done using tools for iteration, which are explained in detail in the page on [Iteration, loops, and lists](#iteration-loops-and-lists). Options include the **purrr** package, or use of a for loop as explained below.

Below, we use a simple for loop to generate a surveillance report for all hospitals of interest. This is done with one command (instead of manually changing the hospital parameter one-at-a-time). The command to render the reports must exist in a separate script outside the report Rmd. This script will also contain defined objects to “loop through” - today’s date, and a vector of hospital names to loop through.

We then feed these values one-at-a-time into the render() command using a loop, which runs the command once for each value in the hospitals vector. The letter i represents the index position (1 through 4) of the hospital currently being used in that iteration, such that hospital\_list[1] would be “Central Hospital”. This information is supplied in two places in the render() command:

1. To the file name, such that the file name of the first iteration if produced on 10th April 2021 would be “Report\_Central Hospital\_2021-04-10.docx”, saved in the ‘output’ subfolder of the working directory.
2. To params = such that the Rmd uses the hospital name internally whenever the params$hospital value is called (e.g. to filter the dataset to the particular hospital only). In this example, four files would be created - one for each hospital.

## Templates

By using a template document that contains any desired formatting, you can adjust the aesthetics of how the Rmd output will look. You can create for instance an MS Word or Powerpoint file that contains pages/slides with the desired dimensions, watermarks, backgrounds, and fonts.

### Word documents

To create a template, start a new word document (or use an existing output with formatting the suits you), and edit fonts by defining the Styles. In Style,Headings 1, 2, and 3 refer to the various markdown header levels (# Header 1, ## Header 2 and ### Header 3 respectively). Right click on the style and click ‘modify’ to change the font formatting as well as the paragraph (e.g. you can introduce page breaks before certain styles which can help with spacing). Other aspects of the word document such as margins, page size, headers etc, can be changed like a usual word document you are working directly within.

### Powerpoint documents

As above, create a new slideset or use an existing powerpoint file with the desired formatting. For further editing, click on ‘View’ and ‘Slide Master’. From here you can change the ‘master’ slide appearance by editing the text formatting in the text boxes, as well as the background/page dimensions for the overall page.

Unfortunately, editing powerpoint files is slightly less flexible:

* A first level header (# Header 1) will automatically become the title of a new slide,
* A ## Header 2 text will not come up as a subtitle but text within the slide’s main textbox (unless you find a way to maniuplate the Master view).
* Outputted plots and tables will automatically go into new slides. You will need to combine them, for instance the the **patchwork** function to combine ggplots, so that they show up on the same page. See this [blog post](https://mattherman.info/blog/ppt-patchwork/) about using the **patchwork** package to put multiple images on one slide.

See the [**officer** package](https://davidgohel.github.io/officer/) for a tool to work more in-depth with powerpoint presentations.

### Integrating templates into the YAML

Once a template is prepared, the detail of this can be added in the YAML of the Rmd underneath the ‘output’ line and underneath where the document type is specified (which goes to a separate line itself). Note reference\_doc can be used for powerpoint slide templates.

It is easiest to save the template in the same folder as where the Rmd file is (as in the example below), or in a subfolder within.

---

title: Surveillance report

output:

word\_document:

reference\_docx: "template.docx"

params:

date: 2021-04-10

hospital: Central Hospital

template:

---

### Formatting HTML files

HTML files do not use templates, but can have the styles configured within the YAML. HTMLs are interactive documents, and are particularly flexible. We cover some basic options here.

* Table of contents: We can add a table of contents with toc: true below, and also specify that it remains viewable (“floats”) as you scroll, with toc\_float: true.
* Themes: We can refer to some pre-made themes, which come from a Bootswatch theme library. In the below example we use cerulean. Other options include: journal, flatly, darkly, readable, spacelab, united, cosmo, lumen, paper, sandstone, simplex, and yeti.
* Highlight: Configuring this changes the look of highlighted text (e.g. code within chunks that are shown). Supported styles include default, tango, pygments, kate, monochrome, espresso, zenburn, haddock, breezedark, and textmate.

Here is an example of how to integrate the above options into the YAML.

---

title: "HTML example"

output:

html\_document:

toc: true

toc\_float: true

theme: cerulean

highlight: kate

---

Below are two examples of HTML outputs which both have floating tables of contents, but different theme and highlight styles selected:

## Dynamic content

In an HTML output, your report content can be dynamic. Below are some examples:

### Tables

In an HTML report, you can print data frame / tibbles such that the content is dynamic, with filters and scroll bars. There are several packages that offer this capability.

To do this with the **DT** package, as is used throughout this handbook, you can insert a code chunk like this:

The function datatable() will print the provided data frame as a dynamic table for the reader. You can set rownames = FALSE to simplify the far left-side of the table. filter = "top" provides a filter over each column. In the option() argument provide a list of other specifications. Below we include two: pageLength = 5 set the number of rows that appear as 5 (the remaining rows can be viewed by paging through arrows), and scrollX=TRUE enables a scrollbar on the bottom of the table (for columns that extend too far to the right).

If your dataset is very large, consider only showing the top X rows by wrapping the data frame in head().

### HTML widgets

[HTML widgets for R](http://www.htmlwidgets.org/) are a special class of R packages that enable increased interactivity by utilizing JavaScript libraries. You can embed them in HTML R Markdown outputs.

Some common examples of these widgets include:

* Plotly (used in this handbook page and in the [Interative plots] page)
* visNetwork (used in the [Transmission Chains](#transmission-chains) page of this handbook)
* Leaflet (used in the [GIS Basics](#gis-basics) page of this handbook)
* dygraphs (useful for interactively showing time series data)
* DT (datatable()) (used to show dynamic tables with filter, sort, etc.)

The ggplotly() function from **plotly** is particularly easy to use. See the [Interactive plots](#interactive-plots-1) page.

## Resources

Further information can be found via:

* <https://bookdown.org/yihui/rmarkdown/>
* <https://rmarkdown.rstudio.com/articles_intro.html>

A good explainer of markdown vs knitr vs Rmarkdown is here: <https://stackoverflow.com/questions/40563479/relationship-between-r-markdown-knitr-pandoc-and-bookdown>

# Organizing routine reports

This page covers the **reportfactory** package, which is an accompaniment to using R Markdown for reports.

In scenarios where you run reports routinely (daily, weekly, etc.), it eases the compilation of multiple R Markdown files and the organization of their outputs. In essence, it provides a “factory” from which you can run the R Markdown reports, get automatically date- and time-stamped folders for the outputs, and have “light” version control.

**reportfactory** is one of the packages developed by RECON (R Epidemics Consortium). Here is their [website](https://www.repidemicsconsortium.org/) and [Github](https://github.com/reconverse).

## Preparation

### Load packages

From within RStudio, install the latest version of the **reportfactory** package from Github.

You can do this via the **pacman** package with p\_load\_current\_gh() which will force intall of the latest version from Github. Provide the character string “reconverse/reportfactory”, which specifies the Github organization (reconverse) and repository (reportfactory). You can also use install\_github() from the **remotes** package, as an alternative.

## New factory

To create a new factory, run the function new\_factory(). This will create a new self-contained R project folder. By default:

* The factory will be added to your working directory
* The name of the factory R project will be called “new\_factory.Rproj”
* Your RStudio session will “move in” to this R project

Looking inside the factory, you can see that sub-folders and some files were created automatically.

* The report\_sources folder will hold your R Markdown scripts, which generate your reports
* The outputs folder will hold the report outputs (e.g. HTML, Word, PDF, etc.)
* The scripts folder can be used to store other R scripts (e.g. that are sourced by your Rmd scripts)
* The data folder can be used to hold your data (“raw” and “clean” subfolders are included)
* A .here file, so you can use the **here** package to call files in sub-folders by their relation to this root folder (see [R projects](#r-projects) page for details)
* A gitignore file was created in case you link this R project to a Github repository (see [Version control and collaboration with Github])
* An empty README file, for if you use a Github repository

**CAUTION:** depending on your computer’s setting, files such as “.here” may exist but be invisible.

Of the default settings, below are several that you might want to adjust within the new\_factory() command:

* factory = - Provide a name for the factory folder (default is “new\_factory”)
* path = - Designate a file path for the new factory (default is the working directory)
* report\_sources = Provide an alternate name for the subfolder which holds the R Markdown scripts (default is “report\_sources”)
* outputs = Provide an alternate name for the folder which holds the report outputs (default is “outputs”)

See ?new\_factory for a complete list of the arguments.

When you create the new factory, your R session is transferred to the new R project, so you should again load the **reportfactory** package.

Now you can run a the factory\_overview() command to see the internal structure (all folders and files) in the factory.

The following “tree” of the factory’s folders and files is printed to the R console. Note that in the “data” folder there are sub-folders for “raw” and “clean” data, and example CSV data. There is also “example\_report.Rmd” in the “report\_sources” folder.

## Create a report

From within the factory R project, create a R Markdown report just as you would normally, and save it into the “report\_sources” folder. See the [R Markdown](#reports-with-r-markdown) page for instructions. For purposes of example, we have added the following to the factory:

* A new R markdown script entitled “daily\_sitrep.Rmd”, saved within the “report\_sources” folder
* Data for the report (“linelist\_cleaned.rds”), saved to the “clean” sub-folder within the “data” folder

We can see using factory\_overview() our R Markdown in the “report\_sources” folder and the data file in the “clean” data folder (highlighted):

Below is a screenshot of the beginning of the R Markdown “daily\_sitrep.Rmd”. You can see that the output format is set to be HTML, via the YAML header output: html\_document.

In this simple script, there are commands to:

* Load necessary packages
* Import the linelist data using a filepath from the **here** package (read more in the page on [Import and export](#import-and-export))
* Print a summary table of cases, and export it with export() as a .csv file
* Print an epicurve, and export it with ggsave() as a .png file

You can review just the list of R Markdown reports in the “report\_sources” folder with this command:

## Compile

In a report factory, to “compile” a R Markdown report means that the .Rmd script will be run and the output will be produced (as specified in the script YAML e.g. as HTML, Word, PDF, etc).

The factory will automatically create a date- and time-stamped folder for the outputs in the “outputs” folder.

The report itself and any exported files produced by the script (e.g. csv, png, xlsx) will be saved into this folder. In addition, the Rmd script itself will be saved in this folder, so you have a record of that version of the script.

This contrasts with the normal behavior of a “knitted” R Markdown, which saves outputs to the location of the Rmd script. This default behavior can result in crowded, messy folders. The factory aims to improve organization when one needs to run reports frequently.

### Compile by name

You can compile a specific report by running compile\_reports() and providing the Rmd script name (without .Rmd extension) to reports =. For simplicity, you can skip the reports = and just write the R Markdown name in quotes, as below.

This command would compile only the “daily\_sitrep.Rmd” report, saving the HTML report, and the .csv table and .png epicurve exports into a date- and time-stamped sub-folder specific to the report, within the “outputs” folder.

Note that if you choose to provide the .Rmd extension, you must correctly type the extension as it is saved in the file name (.rmd vs. .Rmd).

Also note that when you compile, you may see several files temporarily appear in the “report\_sources” folder - but they will soon disappear as they are transferred to the correct “outputs” folder.

### Compile by number

You can also specify the Rmd script to compile by providing a number or vector of numbers to reports =. The numbers must align with the order the reports appear when you run list\_reports().

### Compile all

You can compile all the R Markdown reports in the “report\_sources” folder by setting the reports = argument to TRUE.

### Compile from sub-folder

You can add sub-folders to the “report\_sources” folder. To run an R Markdown report from a subfolder, simply provide the name of the folder to subfolder =. Below is an example of code to compile a Rmd report that lives in a sub\_folder of “report\_sources”.

You can compile all Rmd reports within a subfolder by providing the subfolder name to reports =, with a slash on the end, as below.

### Parameterization

As noted in the page on [Reports with R Markdown](#reports-with-r-markdown), you can run reports with specified parameters. You can pass these parameters as a list to compile\_reports() via the params = argument. For example, in this fictional report there are three parameters provided to the R Markdown reports.

### Using a “run-file”

If you have multiple reports to run, consider creating a R script that contains all the compile\_reports() commands. A user can simply run all the commands in this R script and all the reports will compile. You can save this “run-file” to the “scripts” folder.

## Outputs

After we have compiled the reports a few times, the “outputs” folder might look like this (highlights added for clarity):

* Within “outputs”, sub-folders have been created for each Rmd report
* Within those, further sub-folders have been created for each unique compiling
  + These are date- and time-stamped (“2021-04-23\_T11-07-36” means 23rd April 2021 at 11:07:36)
  + You can edit the date/time-stamp format. See ?compile\_reports
* Within each date/time compiled folder, the report output is stored (e.g. HTML, PDF, Word) along with the Rmd script (version control!) and any other exported files (e.g. table.csv, epidemic\_curve.png)

Here is a view inside one of the date/time-stamped folders, for the “daily\_sitrep” report. The file path is highlighted in yellow for emphasis.

Finally, below is a screenshot of the HTML report output.

You can use list\_outputs() to review a list of the outputs.

## Miscellaneous

### Knit

You can still “knit” one of your R Markdown reports by pressing the “Knit” button, if you want. If you do this, as by default, the outputs will appear in the folder where the Rmd is saved - the “report\_sources” folder. In prior versions of **reportfactory**, having any non-Rmd files in “report\_sources” would prevent compiling, but this is no longer the case. You can run compile\_reports() and no error will occur.

### Scripts

We encourage you to utilize the “scripts” folder to store “runfiles” or .R scripts that are sourced by your .Rmd scripts. See the page on [R Markdown](#reports-with-r-markdown) for tips on how to structure your code across several files.

### Extras

* With **reportfactory**, you can use the function list\_deps() to list all packages required across all the reports in the entire factory.
* There is an accompanying package in development called **rfextras** that offers more helper functions to assist you in building reports, such as:
  + load\_scripts() - sources/loads all .R scripts in a given folder (the “scripts” folder by default)
  + find\_latest() - finds the latest version of a file (e.g. the latest dataset)

## Resources

See the **reportfactory** package’s [Github page](https://github.com/reconverse/reportfactory)

See the **rfextras** package’s [Github page](https://github.com/reconhub/rfextras)

# Dashboards with R Markdown

This page will cover the basic use of the **flexdashboard** package. This package allows you to easily format R Markdown output as a dashboard with panels and pages. The dashboard content can be text, static figures/tables or interactive graphics.

Advantages of **flexdashboard**:

* It requires minimal non-standard R coding - with very little practice you can quickly create a dashboard
* The dashboard can usually be emailed to colleagues as a self-contained HTML file - no server required
* You can combine **flexdashboard** with **shiny**, **ggplotly**, and other “html widgets” to add interactivity

Disadvantages of **flexdashboard**:

* Less customization as compared to using **shiny** alone to create a dashboard

Very comprehensive tutorials on using **flexdashboard** that informed this page can be found in the Resources section. Below we describe the core features and give an example of building a dashboard to explore an outbreak, using the case linelist data.

## Preparation

### Load packages

In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### 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](https://github.com/epirhandbook/Epi_R_handbook/raw/master/data/case_linelists/linelist_cleaned.rds) (as .rds file). Import data with the import() function from the **rio** package (it handles many file types like .xlsx, .csv, .rds - see the [Import and export](#import-and-export) page for details).

The first 50 rows of the linelist are displayed below.

## Create new R Markdown

After you have installed the package, create a new R Markdown file by clicking through to File > New file > R Markdown.

In the window that opens, select “From Template” and select the “Flex Dashboard” template. You will then be prompted to name the document. In this page’s example, we will name our R Markdown as “outbreak\_dashboard.Rmd”.

## The script

The script is an R Markdown script, and so has the same components and organization as described in the page on [Reports with R Markdown](#reports-with-r-markdown). We briefly re-visit these and highlight differences from other R Markdown output formats.

### YAML

At the top of the script is the “YAML” header. This must begin with three dashes --- and must close with three dashes ---. YAML parameters comes in key:value pairs. **The indentation and placement of colons in YAML is important** - the key:value pairs are separated by colons (not equals signs!).

The YAML should begin with metadata for the document. The order of these primary YAML parameters (not indented) does not matter. For example:

You can use R code in YAML values by putting it like in-line code (preceeded by r within backticks) but also within quotes (see above for Date).

A required YAML parameter is output:, which specifies the type of file to be produced (e.g. html\_document, pdf\_document, word\_document, or powerpoint\_presentation). For **flexdashboard** this parameter value is a bit confusing - it must be set as output:flexdashboard::flex\_dashboard. Note the single and double colons, and the underscore. This YAML output parameter is often followed by an additional colon and indented sub-parameters (see orientation: and vertical\_layout: parameters below).

As shown above, indentations (2 spaces) are used for sub-parameters. In this case, do not forget to put an additional colon after the primary, like key:value:.

If appropriate, logical values should be given in YAML in lowercase (true, false, null). If a colon is part of your value (e.g. in the title) put the value in quotes. See the examples in sections below.

### Code chunks

An R Markdown script can contain multiple code “chunks” - these are areas of the script where you can write multiple-line R code and they function just like mini R scripts.

Code chunks are created with three back-ticks and curly brackets with a lowercase “r” within. The chunk is closed with three backticks. You can create a new chunk by typing it out yourself, by using the keyboard shortcut “Ctrl + Alt + i” (or Cmd + Shift + r in Mac), or by clicking the green ‘insert a new code chunk’ icon at the top of your script editor. Many examples are given below.

### Narrative text

Outside of an R code “chunk”, you can write narrative text. As described in the page on [Reports with R Markdown](#reports-with-r-markdown), you can italicize text by surrounding it with one asterisk (\*), or bold by surrounding it with two asterisks (\*\*). Recall that bullets and numbering schemes are sensitive to newlines, indentation, and finishing a line with two spaces.

You can also insert in-line R code into text as described in the [Reports with R Markdown](#reports-with-r-markdown) page, by surrounding the code with backticks and starting the command with “r”: ` 1+1`(see example with date above).

### Headings

Different heading levels are established with different numbers of hash symbols, as described in the [Reports with R Markdown](#reports-with-r-markdown) page.

In **flexdashboard**, a primary heading (#) creates a “page” of the dashboard. Second-level headings (##) create a column or a row depending on your orientation: parameter (see details below). Third-level headings (###) create panels for plots, charts, tables, text, etc.

# First-level heading (page)

## Second level heading (row or column)

### Third-level heading (pane for plot, chart, etc.)

## Section attributes

As in a normal R markdown, you can specify attributes to apply to parts of your dashboard by including key=value options after a heading, within curly brackets { }. For example, in a typical HTML R Markdown report you might organize sub-headings into tabs with ## My heading {.tabset}.

Note that these attributes are written after a heading in a text portion of the script. These are different than the **knitr** options inserted within at the top of R code chunks, such as out.height =.

Section attributes specific to **flexdashboard** include:

* {data-orientation=} Set to either rows or columns. If your dashboard has multiple pages, add this attribute to each page to indicate orientation (further explained in [layout section](#layout)).
* {data-width=} and {data-height=} set relative size of charts, columns, rows laid out in the same dimension (horizontal or vertical). Absolute sizes are adjusted to best fill the space on any display device thanks to the [flexbox](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes) engine.
  + Height of charts also depends on whether you set the YAML parameter vertical\_layout: fill or vertical\_layout: scroll. If set to scroll, figure height will reflect the traditional fig.height = option in the R code chunk.
  + See complete size documentation at the [flexdashboard website](https://rmarkdown.rstudio.com/flexdashboard/using.html#sizing)
* {.hidden} Use this to exclude a specific page from the navigation bar
* {data-navbar=} Use this in a page-level heading to nest it within a navigation bar drop-down menu. Provide the name (in quotes) of the drop-down menu. See example below.

## Layout

Adjust the layout of your dashboard in the following ways:

* Add pages, columns/rows, and charts with R Markdown headings (e.g. #, ##, or ###)
* Adjust the YAML parameter orientation: to either rows or columns
* Specify whether the layout fills the browser or allows scrolling
* Add tabs to a particular section heading

### Pages

First-level headings (#) in the R Markdown will represent “pages” of the dashboard. By default, pages will appear in a navigation bar along the top of the dashboard.

You can group pages into a “menu” within the top navigation bar by adding the attribute {data-navmenu=} to the page heading. Be careful - do not include spaces around the equals sign otherwise it will not work!

Here is what the script produces:

You can also convert a page or a column into a “sidebar” on the left side of the dashboard by adding the {.sidebar} attribute. It can hold text (viewable from any page), or if you have integrated **shiny** interactivity it can be useful to hold user-input controls such as sliders or drop-down menus.

Here is what the script produces:

### Orientation

Set the orientation: yaml parameter to indicate how your second-level (##) R Markdown headings should be interpreted - as either orientation: columns or orientation: rows.

Second-level headings (##) will be interpreted as new columns or rows based on this orientation setting.

If you set orientation: columns, second-level headers will create new columns in the dashboard. The below dashboard has one page, containing two columns, with a total of three panels. You can adjust the relative width of the columns with {data-width=} as shown below.

Here is what the script produces:

If you set orientation: rows, second-level headers will create new rows instead of columns. Below is the same script as above, but orientation: rows so that second-level headings produce rows instead of columns. You can adjust the relative height of the rows with {data-height=} as shown below.

Here is what the script produces:

If your dashboard has multiple pages, you can designate the orientation for each specific page by adding the {data-orientation=} attribute the header of each page (specify either rows or columns without quotes).

### Tabs

You can divide content into tabs with the {.tabset} attribute, as in other HTML R Markdown outputs.

Simply add this attribute after the desired heading. Sub-headings under that heading will be displayed as tabs. For example, in the example script below column 2 on the right (##) is modified so that the epidemic curve and table panes (###) are displayed in tabs.

You can do the same with rows if your orientation is rows.

Here is what the script produces:

## Adding content

Let’s begin to build a dashboard. Our simple dashboard will have 1 page, 2 columns, and 4 panels. We will build the panels piece-by-piece for demonstration.

You can easily include standard R outputs such as text, ggplots, and tables (see [Tables for presentation](#tables-for-presentation) page). Simply code them within an R code chunk as you would for any other R Markdown script.

Note: you can download the finished Rmd script and HTML dashboard output - see the [Download handbook and data](#download-handbook-and-data) page.

### Text

You can type in Markdown text and include in-line code as for any other R Markdown output. See the [Reports with R Markdown](#reports-with-r-markdown) page for details.

In this dashboard we include a summary text panel that includes dynamic text showing the latest hospitalisation date and number of cases reported in the outbreak.

### Tables

You can include R code chunks that print outputs such as tables. But the output will look best and respond to the window size if you use the kable() function from **knitr** to display your tables. The **flextable** functions may produce tables that are shortened / cut-off.

For example, below we feed the linelist() through a count() command to produce a summary table of cases by hospital. Ultimately, the table is piped to knitr::kable() and the result has a scroll bar on the right. You can read more about customizing your table with kable() and **kableExtra** [here](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html).

Here is what the script produces:

If you want to show a dynamic table that allows the user to filter, sort, and/or click through “pages” of the data frame, use the package **DT** and it’s function datatable(), as in the code below.

The example code below, the data frame linelist is printed. You can set rownames = FALSE to conserve horizontal space, and filter = "top" to have filters on top of every column. A list of other specifications can be provided to options =. Below, we set pageLength = so that 5 rows appear and scrollX = so the user can use a scroll bar on the bottom to scroll horizontally. The argument class = 'white-space: nowrap' ensures that each row is only one line (not multiple lines). You can read about other possible arguments and values [here](https://rstudio.github.io/DT/?_ga=2.2810736.1321860763.1619286819-369061888.1601594705) or by entering ?datatable

### Plots

You can print plots to a dashboard pane as you would in an R script. In our example, we use the **incidence2** package to create an “epicurve” by age group with two simple commands (see [Epidemic curves](#epidemic-curves) page). However, you could use ggplot() and print a plot in the same manner.

Here is what the script produces:

### Interactive plots

You can also pass a standard ggplot or other plot object to ggplotly() from the **plotly** package (see the [Interactive plots](#interactive-plots-1) page). This will make your plot interactive, allow the reader to “zoom in”, and show-on-hover the value of every data point (in this scenario the number of cases per week and age group in the curve).

Here is what this looks like in the dashboard (gif). This interactive functionality will still work even if you email the dashboard as a static file (not online on a server).

### HTML widgets

[HTML widgets for R](http://www.htmlwidgets.org/) are a special class of R packages that increases interactivity by utilizing JavaScript libraries. You can embed them in R Markdown outputs (such as a flexdashboard) and in Shiny dashboards.

Some common examples of these widgets include:

* Plotly (used in this handbook page and in the [Interative plots] page)
* visNetwork (used in the [Transmission Chains](#transmission-chains) page of this handbook)
* Leaflet (used in the [GIS Basics](#gis-basics) page of this handbook)
* dygraphs (useful for interactively showing time series data)
* DT (datatable()) (used to show dynamic tables with filter, sort, etc.)

Below we demonstrate adding an epidemic transmission chain which uses visNetwork to the dashboard. The script shows only the new code added to the “Column 2” section of the R Markdown script. You can find the code in the [Transmission chains](#transmission-chains) page of this handbook.

Here is what the script produces:

## Code organization

You may elect to have all code within the R Markdown **flexdashboard** script. Alternatively, to have a more clean and concise dashboard script you may choose to call upon code/figures that are hosted or created in external R scripts. This is described in greater detail in the [Reports with R Markdown](#reports-with-r-markdown) page.

## Shiny

Integrating the R package **shiny** can make your dashboards even more reactive to user input. For example, you could have the user select a jurisdiction, or a date range, and have panels react to their choice (e.g. filter the data displayed). To embed **shiny** reactivity into **flexdashboard**, you need only make a few changes to your **flexdashboard** R Markdown script.

You can use **shiny** to produce apps/dashboards without flexdashboard too. The handbook page on [Dashboards with Shiny](#dashboards-with-shiny) gives an overview of this approach, including primers on **shiny** syntax, app file structure, and options for sharing/publishing (including free server options). These syntax and general tips translate into the **flexdashboard** context as well.

Embedding **shiny** in **flexdashboard** is however, a fundamental change to your flexdashboard. It will no longer produce an HTML output that you can send by email and anyone could open and view. Instead, it will be an “app”. The “Knit” button at the top of the script will be replaced by a “Run document” icon, which will open an instance of the interactive the dashboard locally on your computer.

Sharing your dashboard will now require that you either:

* Send the Rmd script to the viewer, they open it in R on their computer, and run the app, or
* The app/dashboard is hosted on a server accessible to the viewer

Thus, there are benefits to integrating **shiny**, but also complications. If easy sharing by email is a priority and you don’t need **shiny** reactive capabilities, consider the reduced interactivity offered by ggplotly() as demonstrated above.

Below we give a very simple example using the same “outbreak\_dashboard.Rmd” as above. Extensive documentation on integrating Shiny into **flexdashboard** is available online [here](https://rmarkdown.rstudio.com/flexdashboard/shiny.html).

### Settings

Enable **shiny** in a **flexdashboard** by adding the YAML parameter runtime: shiny at the same indentation level as output:, as below:

---

title: "Outbreak dashboard (Shiny demo)"

output:

flexdashboard::flex\_dashboard:

orientation: columns

vertical\_layout: fill

runtime: shiny

---

It is also convenient to enable a “side bar” to hold the shiny input widgets that will collect information from the user. As explained above, create a column and indicate the {.sidebar} option to create a side bar on the left side. You can add text and R chunks containing the **shiny** input commands within this column.

If your app/dashboard is hosted on a server and may have multiple simultaneous users, name the first R code chunk as global. Include the commands to import/load your data in this chunk. This special named chunk is treated differently, and the data imported within it are only imported once (not continuously) and are available for all users. This improves the start-up speed of the app.

### Worked example

Here we adapt the flexdashboard script “outbreak\_dashboard.Rmd” to include **shiny**. We will add the capability for the user to select a hospital from a drop-down menu, and have the epidemic curve reflect only cases from that hospital, with a dynamic plot title. We do the following:

* Add runtime: shiny to the YAML
* Re-name the setup chunk as global
* Create a sidebar containing:
  + Code to create a vector of unique hospital names
  + A selectInput() command (**shiny** drop-down menu) with the choice of hospital names. The selection is saved as hospital\_choice, which can be referenced in later code as input$hospital\_choice
* The epidemic curve code (column 2) is wrapped within renderPlot({ }), including:
  + A filter on the dataset restricting the column hospital to the current value of input$hospital\_choice
  + A dynamic plot title that incorporates input$hospital\_choice

Note that any code referencing an input$ value must be within a render({}) function (to be reactive).

Here is the top of the script, including YAML, global chunk, and sidebar:

Here is the Column 2, with the reactive epicurve plot:

And here is the dashboard:

### Other examples

To read a health-related example of a Shiny-**flexdashboard** using the **shiny** interactivity and the **leaflet** mapping widget, see this chapter of the online book [Geospatial Health Data: Modeling and Visualization with R-INLA and Shiny](https://www.paulamoraga.com/book-geospatial/sec-dashboardswithshiny.html).

## Sharing

Dashboards that do not contain Shiny elements will output an HTML file (.html), which can be emailed (if size permits). This is useful, as you can send the “dashboard” report and not have to set up a server to host it as a website.

If you have embedded **shiny**, you will not be able to send an output by email, but you can send the script itself to an R user, or host the dashboard on a server as explained above.

## Resources

Excellent tutorials that informed this page can be found below. If you review these, most likely within an hour you can have your own dashboard.

<https://bookdown.org/yihui/rmarkdown/dashboards.html>

<https://rmarkdown.rstudio.com/flexdashboard/>

<https://rmarkdown.rstudio.com/flexdashboard/using.html>

<https://rmarkdown.rstudio.com/flexdashboard/examples.html>

# Dashboards with Shiny

Dashboards are often a great way to share results from analyses with others. Producing a dashboard with **shiny** requires a relatively advanced knowledge of the R language, but offers incredible customization and possibilities.

It is recommended that someone learning dashboards with **shiny** has good knowledge of data transformation and visualisation, and is comfortable debugging code, and writing functions. Working with dashboards is not intuitive when you’re starting, and is difficult to understand at times, but is a great skill to learn and gets much easier with practice!

This page will give a short overview of how to make dashboards with **shiny** and its extensions. For an alternative method of making dashboards that is faster, easier, but perhaps less customizeable, see the page on **flextable** ([Dashboards with R Markdown](#dashboards-with-r-markdown)).

## Preparation

### Load packages

In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

We begin by installing the **shiny** R package:

### Import data

If you would like to follow-along with this page, see this section of the [Download handbook and data](#data_shiny). There are links to download the R scripts and data files that produce the final Shiny app.

If you try to re-construct the app using these files, please be aware of the R project folder structure that is created over the course of the demonstration (e.g. folders for “data” and for “funcs”).

## The structure of a shiny app

### Basic file structures

To understand shiny, we first need to understand how the file structure of an app works! We should make a brand new directory before we start. This can actually be made easier by choosing New project in Rstudio, and choosing Shiny Web Application. This will create the basic structure of a shiny app for you.

When opening this project, you’ll notice there is a .R file already present called app.R. It is essential that we have one of two basic file structures:

1. One file called app.R, or
2. Two files, one called ui.R and the other server.R

In this page, we will use the first approach of having one file called app.R. Here is an example script:

If you open this file, you’ll notice that two objects are defined - one called ui and another called server. These objects must be defined in every shiny app and are central to the structure of the app itself! In fact, the only difference between the two file structures described above is that in structure 1, both ui and server are defined in one file, whereas in structure 2 they are defined in separate files. Note: we can also (and we should if we have a larger app) have other .R files in our structure that we can source() into our app.

### The server and the ui

We next need to understand what the server and ui objects actually do. Put simply, these are two objects that are interacting with each other whenever the user interacts with the shiny app.

The UI element of a shiny app is, on a basic level, R code that creates an HTML interface. This means everything that is displayed in the UI of an app. This generally includes:

* “Widgets” - dropdown menus, check boxes, sliders, etc that can be interacted with by the user
* Plots, tables, etc - outputs that are generated with R code
* Navigation aspects of an app - tabs, panes, etc.
* Generic text, hyperlinks, etc
* HTML and CSS elements (addressed later)

The most important thing to understand about the UI is that it receives inputs from the user and displays outputs from the server. There is no active code running in the ui at any time - all changes seen in the UI are passed through the server (more or less). So we have to make our plots, downloads, etc in the server

The server of the shiny app is where all code is being run once the app starts up. The way this works is a little confusing. The server function will effectively react to the user interfacing with the UI, and run chunks of code in response. If things change in the server, these will be passed back up to the ui, where the changes can be seen. Importantly, the code in the server will be executed non-consecutively (or it’s best to think of it this way). Basically, whenever a ui input affects a chunk of code in the server, it will run automatically, and that output will be produced and displayed.

This all probably sounds very abstract for now, so we’ll have to dive into some examples to get a clear idea of how this actually works.

### Before you start to build an app

Before you begin to build an app, its immensely helpful to know what you want to build. Since your UI will be written in code, you can’t really visualise what you’re building unless you are aiming for something specific. For this reason, it is immensely helpful to look at lots of examples of shiny apps to get an idea of what you can make - even better if you can look at the source code behind these apps! Some great resources for this are:

* The [Rstudio app gallery](https://shiny.rstudio.com/gallery/)

Once you get an idea for what is possible, it’s also helpful to map out what you want yours to look like - you can do this on paper or in any drawing software (PowerPoint, MS paint, etc.). It’s helpful to start simple for your first app! There’s also no shame in using code you find online of a nice app as a template for your work - its much easier than building something from scratch!

## Building a UI

When building our app, its easier to work on the UI first so we can see what we’re making, and not risk the app failing because of any server errors. As mentioned previously, its often good to use a template when working on the UI. There are a number of standard layouts that can be used with shiny that are available from the base shiny package, but it’s worth noting that there are also a number of package extensions such as shinydashboard. We’ll use an example from base shiny to start with.

A shiny UI is generally defined as a series of nested functions, in the following order

1. A function defining the general layout (the most basic is fluidPage(), but more are available)
2. Panels within the layout such as:
   * a sidebar (sidebarPanel())
   * a “main” panel (mainPanel())
   * a tab (tabPanel())
   * a generic “column” (column())
3. Widgets and outputs - these can confer inputs to the server (widgets) or outputs from the server (outputs)
   * Widgets generally are styled as xxxInput() e.g. selectInput()
   * Outputs are generally styled as xxxOutput() e.g. plotOutput()

It’s worth stating again that these can’t be visualised easily in an abstract way, so it’s best to look at an example! Lets consider making a basic app that visualises our malaria facility count data by district. This data has a lot of differnet parameters, so it would be great if the end user could apply some filters to see the data by age group/district as they see fit! We can use a very simple shiny layout to start - the sidebar layout. This is a layout where widgets are placed in a sidebar on the left, and the plot is placed on the right.

Lets plan our app - we can start with a selector that lets us choose the district where we want to visualise data, and another to let us visualise the age group we are interested in. We’ll aim to use these filters to show an epicurve that reflects these parameters. So for this we need:

1. Two dropdown menus that let us choose the district we want, and the age group we’re interested in.
2. An area where we can show our resulting epicurve.

This might look something like this:

When app.R is run with the above UI code (with no active code in the server portion of app.R) the layout appears looking like this - note that there will be no plot if there is no server to render it, but our inputs are working!

This is a good opportunity to discuss how widgets work - note that each widget is accepting an inputId, a label, and a series of other options that are specific to the widget type. This inputId is extremely important - these are the IDs that are used to pass information from the UI to the server. For this reason, they must be unique. You should make an effort to name them something sensible, and specific to what they are interacting with in cases of larger apps.

You should read documentation carefully for full details on what each of these widgets do. Widgets will pass specific types of data to the server depending on the widget type, and this needs to be fully understood. For example, selectInput() will pass a character type to the server:

* If we select Spring for the first widget here, it will pass the character object "Spring" to the server.
* If we select two items from the dropdown menu, they will come through as a character vector (e.g. c("Spring", "Bolo")).

Other widgets will pass different types of object to the server! For example:

* numericInput() will pass a numeric type object to the server
* checkboxInput() will pass a logical type object to the server (TRUE or FALSE)

It’s also worth noting the named vector we used for the age data here. For many widgets, using a named vector as the choices will display the names of the vector as the display choices, but pass the selected value from the vector to the server. I.e. here someone can select “15+” from the drop-down menu, and the UI will pass "malaria\_rdt\_15" to the server - which happens to be the name of the column we’re interested in!

There are loads of widgets that you can use to do lots of things with your app. Widgets also allow you to upload files into your app, and download outputs. There are also some excellent shiny extensions that give you access to more widgets than base shiny - the **shinyWidgets** package is a great example of this. To look at some examples you can look at the following links:

* [base shiny widget gallery](https://shiny.rstudio.com/gallery/widget-gallery.html)
* [shinyWidgets gallery](https://github.com/dreamRs/shinyWidgets)

## Loading data into our app

The next step in our app development is getting the server up and running. To do this however, we need to get some data into our app, and figure out all the calculations we’re going to do. A shiny app is not straightforward to debug, as it’s often not clear where errors are coming from, so it’s ideal to get all our data processing and visualisation code working before we start making the server itself.

So given we want to make an app that shows epi curves that change based on user input, we should think about what code we would need to run this in a normal R script. We’ll need to:

1. Load our packages
2. Load our data
3. Transform our data
4. Develop a function to visualise our data based on user inputs

This list is pretty straightforward, and shouldn’t be too hard to do. It’s now important to think about which parts of this process need to be done only once and which parts need to run in response to user inputs. This is because shiny apps generally run some code before running, which is only performed once. It will help our app’s performance if as much of our code can be moved to this section. For this example, we only need to load our data/packages and do basic transformations once, so we can put that code outside the server. This means the only thing we’ll need in the server is the code to visualise our data. Lets develop all of these componenets in a script first. However, since we’re visualising our data with a function, we can also put the code for the function outside the server so our function is in the environment when the app runs!

First lets load our data. Since we’re working with a new project, and we want to make it clean, we can create a new directory called data, and add our malaria data in there. We can run this code below in a testing script we will eventually delete when we clean up the structure of our app.

pacman::p\_load("tidyverse", "lubridate")

# read data

malaria\_data <- rio::import(here::here("data", "malaria\_facility\_count\_data.rds")) %>%

as\_tibble()

print(malaria\_data)

It will be easier to work with this data if we use tidy data standards, so we should also transform into a longer data format, where age group is a column, and cases is another column. We can do this easily using what we’ve learned in the [Pivoting data](#pivoting-data) page.

malaria\_data <- malaria\_data %>%

select(-newid) %>%

pivot\_longer(cols = starts\_with("malaria\_"), names\_to = "age\_group", values\_to = "cases\_reported")

print(malaria\_data)

And with that we’ve finished preparing our data! This crosses items 1, 2, and 3 off our list of things to develop for our “testing R script”. The last, and most difficult task will be building a function to produce an epicurve based on user defined parameters. As mentioned previously, it’s highly recommended that anyone learning shiny first look at the section on functional programming ([Writing functions](#writing-functions-1)) to understand how this works!

When defining our function, it might be hard to think about what parameters we want to include. For functional programming with shiny, every relevent parameter will generally have a widget associated with it, so thinking about this is usually quite easy! For example in our current app, we want to be able to filter by district, and have a widget for this, so we can add a district parameter to reflect this. We don’t have any app functionality to filter by facility (for now), so we don’t need to add this as a parameter. Lets start by making a function with three parameters:

1. The core dataset
2. The district of choice
3. The age group of choice

We won’t go into great detail about this function, as it’s relatively simple in how it works. One thing to note however, is we handle errors by returning NULL when it would otherwise give an error. This is because when a shiny server produces a NULL object instead of a plot object, nothing will be shown in the ui! This is important, as otherwise errors will often cause your app to stop working.

Another thing to note is the use of the %in% operator when evaluating the district input. As mentioned above, this could arrive as a character vector with multiple values, so using %in% is more flexible than say, ==.

Let’s test our function!

plot\_epicurve(malaria\_data, district = "Bolo", agegroup = "malaria\_rdt\_0-4")

With our function working, we now have to understand how this all is going to fit into our shiny app. We mentioned the concept of startup code before, but lets look at how we can actually incorporate this into the structure of our app. There are two ways we can do this!

1. Put this code in your app.R file at the start of the script (above the UI), or
2. Create a new file in your app’s directory called global.R, and put the startup code in this file.

It’s worth noting at this point that it’s generally easier, especially with bigger apps, to use the second file structure, as it lets you separate your file structure in a simple way. Lets fully develop a this global.R script now. Here is what it could look like:

Easy! One great feature of shiny is that it will understand what files named app.R, server.R, ui.R, and global.R are for, so there is no need to connect them to each other via any code. So just by having this code in global.R in the directory it will run before we start our app!.

We should also note that it would improve our app’s organisation if we moved the plotting function to its own file - this will be especially helpful as apps become larger. To do this, we could make another directory called funcs, and put this function in as a file called plot\_epicurve.R. We could then read this function in via the following command in global.R

Note that you should always specify local = TRUE in shiny apps, since it will affect sourcing when/if the app is published on a server.

## Developing an app server

Now that we have most of our code, we just have to develop our server. This is the final piece of our app, and is probably the hardest to understand. The server is a large R function, but its helpful to think of it as a series of smaller functions, or tasks that the app can perform. It’s important to understand that these functions are not executed in a linear order. There is an order to them, but it’s not fully necessary to understand when starting out with shiny. At a very basic level, these tasks or functions will activate when there is a change in user inputs that affects them, unless the developer has set them up so they behave differently. Again, this is all quite abstract, but lets first go through the three basic types of shiny objects

1. Reactive sources - this is another term for user inputs. The shiny server has access to the outputs from the UI through the widgets we’ve programmed. Every time the values for these are changed, this is passed down to the server.
2. Reactive conductors - these are objects that exist only inside the shiny server. We don’t actually need these for simple apps, but they produce objects that can only be seen inside the server, and used in other operations. They generally depend on reactive sources.
3. Endpoints - these are outputs that are passed from the server to the UI. In our example, this would be the epi curve we are producing.

With this in mind lets construct our server step-by-step. We’ll show our UI code again here just for reference:

From this code UI we have:

* Two inputs:
  + District selector (with an inputId of select\_district)
  + Age group selector (with an inputId of select\_agegroup)
* One output:
  + The epicurve (with an outputId of malaria\_epicurve)

As stated previously, these unique names we have assigned to our inputs and outputs are crucial. They must be unique and are used to pass information between the ui and server. In our server, we access our inputs via the syntax input$inputID and outputs and passed to the ui through the syntax output$output\_name Lets have a look at an example, because again this is hard to understand otherwise!

The server for a simple app like this is actually quite straightforward! You’ll notice that the server is a function with three parameters - input, output, and session - this isn’t that important to understand for now, but its important to stick to this setup! In our server we only have one task - this renders a plot based on our function we made earlier, and the inputs from the server. Notice how the names of the input and output objects correspond exactly to those in the ui.

To understand the basics of how the server reacts to user inputs, you should note that the output will know (through the underlying package) when inputs change, and rerun this function to create a plot every time they change. Note that we also use the renderPlot() function here - this is one of a family of class-specific functions that pass those objects to a ui output. There are a number of functions that behave similarly, but you need to ensure the function used matches the class of object you’re passing to the ui! For example:

* renderText() - send text to the ui
* renderDataTable - send an interactive table to the ui.

Remember that these also need to match the output function used in the ui - so renderPlot() is paired with plotOutput(), and renderText() is matched with textOutput().

So we’ve finally made a functioning app! We can run this by pressing the Run App button on the top right of the script window in Rstudio. You should note that you can choose to run your app in your default browser (rather than Rstudio) which will more accurately reflect what the app will look like for other users.

It is fun to note that in the R console, the app is “listening”! Talk about reactivity!

## Adding more functionality

At this point we’ve finally got a running app, but we have very little functionality. We also haven’t really scratched the surface of what shiny can do, so there’s a lot more to learn about! Lets continue to build our existing app by adding some extra features. Some things that could be nice to add could be:

1. Some explanatory text
2. A download button for our plot - this would provide the user with a high quality version of the image that they’re generating in the app
3. A selector for specific facilities
4. Another dashboard page - this could show a table of our data.

This is a lot to add, but we can use it to learn about a bunch of different shiny featues on the way. There is so much to learn about shiny (it can get very advanced, but its hopefully the case that once users have a better idea of how to use it they can become more comfortable using external learning sources as well).

### Adding static text

Lets first discuss adding static text to our shiny app. Adding text to our app is extremely easy, once you have a basic grasp of it. Since static text doesn’t change in the shiny app (If you’d like it to change, you can use text rendering functions in the server!), all of shiny’s static text is generally added in the ui of the app. We wont go through this in great detail, but you can add a number of different elements to your ui (and even custom ones) by interfacing R with HTML and css.

HTML and css are languages that are explicitly involved in user interface design. We don’t need to understand these too well, but HTML creates objects in UI (like a text box, or a table), and css is generally used to change the style and aesthetics of those objects. Shiny has access to a large array of HTML tags - these are present for objects that behave in a specific way, such as headers, paragraphs of text, line breaks, tables, etc. We can use some of these examples like this:

* h1() - this a a header tag, which will make enclosed text automatically larger, and change defaults as they pertain to the font face, colour etc (depending on the overall theme of your app). You can access smaller and smaller sub-heading with h2() down to h6() as well. Usage looks like:
  + h1("my header - section 1")
* p() - this is a paragraph tag, which will make enclosed text similar to text in a body of text. This text will automatically wrap, and be of a relatively small size (footers could be smaller for example.) Think of it as the text body of a word document. Usage looks like:
  + p("This is a larger body of text where I am explaining the function of my app")
* tags$b() and tags$i() - these are used to create bold tags$b() and italicised tags$i() with whichever text is enclosed!
* tags$ul(), tags$ol() and tags$li() - these are tags used in creating lists. These are all used within the syntax below, and allow the user to create either an ordered list (tags$ol(); i.e. numbered) or unordered list (tags$ul(), i.e. bullet points). tags$li() is used to denote items in the list, regardless of which type of list is used. e.g.:
* br() and hr() - these tags create linebreaks and horizontal lines (with a linebreak) respectively. Use them to separate out the sections of your app and text! There is no need to pass any items to these tags (parentheses can remain empty).
* div() - this is a generic tag that can contain anything, and can be named anything. Once you progress with ui design, you can use these to compartmentalize your ui, give specific sections specific styles, and create interactions between the server and UI elements. We won’t go into these in detail, but they’re worth being aware of!

Note that every one of these objects can be accessed through tags$... or for some, just the function. These are effectively synonymous, but it may help to use the tags$... style if you’d rather be more explicit and not overwrite the functions accidentally. This is also by no means an exhaustive list of tags available. There is a full list of all tags available in shiny [here](https://shiny.rstudio.com/articles/tag-glossary.html) and even more can be used by inserting HTML directly into your ui!

If you’re feeling confident, you can also add any css styling elements to your HTML tags with the style argument in any of them. We won’t go into how this works in detail, but one tip for testing aesthetic changes to a UI is using the HTML inspector mode in chrome (of your shiny app you are running in browser), and editing the style of objects yourself!

Lets add some text to our app

### Adding a link

To add a link to a website, use tags$a() with the link and display text as shown below. To have as a standalone paragraph, put it within p(). To have only a few words of a sentence linked, break the sentence into parts and use tags$a() for the hyperlinked part. To ensure the link opens in a new browser window, add target = "\_blank" as an argument.

### Adding a download button

Lets move on to the second of the three features. A download button is a fairly common thing to add to an app and is fairly easy to make. We need to add another Widget to our ui, and we need to add another output to our server to attach to it. We can also introduce reactive conductors in this example!

Lets update our ui first - this is easy as shiny comes with a widget called downloadButton() - lets give it an inputId and a label.

Note that we’ve also added in a hr() tag - this adds a horizontal line separating our control widgets from our download widgets. This is another one of the HTML tags that we discussed previously.

Now that we have our ui ready, we need to add the server component. Downloads are done in the server with the downloadHandler() function. Similar to our plot, we need to attach it to an output that has the same inputId as the download button. This function takes two arguments - filename and content - these are both functions. As you might be able to guess, filename is used to specify the name of the downloaded file, and content is used to specify what should be downloaded. content contain a function that you would use to save data locally - so if you were downloading a csv file you could use rio::export(). Since we’re downloading a plot, we’ll use ggplot2::ggsave(). Lets look at how we would program this (we won’t add it to the server yet).

Note that the content function always takes a file argument, which we put where the output file name is specified. You might also notice that we’re repeating code here - we are using our plot\_epicurve() function twice in this server, once for the download and once for the image displayed in the app. While this wont massively affect performance, this means that the code to generate this plot will have to be run when the user changes the widgets specifying the district and age group, and again when you want to download the plot. In larger apps, suboptimal decisions like this one will slow things down more and more, so it’s good to learn how to make our app more efficient in this sense. What would make more sense is if we had a way to run the epicurve code when the districts/age groups are changes, and let that be used by the renderPlot() and downloadHandler() functions. This is where reactive conductors come in!

Reactive conductors are objects that are created in the shiny server in a reactive way, but are not outputted - they can just be used by other parts of the server. There are a number of different kinds of reactive conductors, but we’ll go through the basic two.

1.reactive() - this is the most basic reactive conductor - it will react whenever any inputs used inside of it change (so our district/age group widgets)  
2. eventReactive()- this rective conductor works the same as reactive(), except that the user can specify which inputs cause it to rerun. This is useful if your reactive conductor takes a long time to process, but this will be explained more later.

Lets look at the two examples:

When we use the eventReactive() setup, we can specify which inputs cause this chunk of code to run - this isn’t very useful to us at the moment, so we can leave it for now. Note that you can include multiple inputs with c()

Lets look at how we can integrate this into our server code:

You can see we’re just calling on the output of our reactive we’ve defined in both our download and plot rendering functions. One thing to note that often trips people up is you have to use the outputs of reactives as if they were functions - so you must add empty brackets at the end of them (i.e. malaria\_plot() is correct, and malaria\_plot is not). Now that we’ve added this solution our app is a little tidyer, faster, and easier to change since all our code that runs the epicurve function is in one place.

### Adding a facility selector

Lets move on to our next feature - a selector for specific facilities. We’ll implement another parameter into our function so we can pass this as an argument from our code. Lets look at doing this first - it just operates off the same principles as the other parameters we’ve set up. Lets update and test our function.

plot\_epicurve <- function(data, district = "All", agegroup = "malaria\_tot", facility = "All") {

if (!("All" %in% district)) {

data <- data %>%

filter(District %in% district)

plot\_title\_district <- stringr::str\_glue("{paste0(district, collapse = ', ')} districts")

} else {

plot\_title\_district <- "all districts"

}

# if no remaining data, return NULL

if (nrow(data) == 0) {

return(NULL)

}

data <- data %>%

filter(age\_group == agegroup)

# if no remaining data, return NULL

if (nrow(data) == 0) {

return(NULL)

}

if (agegroup == "malaria\_tot") {

agegroup\_title <- "All ages"

} else {

agegroup\_title <- stringr::str\_glue("{str\_remove(agegroup, 'malaria\_rdt')} years")

}

if (!("All" %in% facility)) {

data <- data %>%

filter(location\_name == facility)

plot\_title\_facility <- facility

} else {

plot\_title\_facility <- "all facilities"

}

# if no remaining data, return NULL

if (nrow(data) == 0) {

return(NULL)

}

ggplot(data, aes(x = data\_date, y = cases\_reported)) +

geom\_col(width = 1, fill = "darkred") +

theme\_minimal() +

labs(

x = "date",

y = "number of cases",

title = stringr::str\_glue("Malaria cases - {plot\_title\_district}; {plot\_title\_facility}"),

subtitle = agegroup\_title

)

}

Let’s test it:

With all the facilites in our data, it isn’t very clear which facilities correspond to which districts - and the end user won’t know either. This might make using the app quite unintuitive. For this reason, we should make the facility options in the UI change dynamically as the user changes the district - so one filters the other! Since we have so many variables that we’re using in the options, we might also want to generate some of our options for the ui in our global.R file from the data. For example, we can add this code chunk to global.R after we’ve read our data in:

Let’s look at them:

We can pass these new variables to the ui without any issue, since they are globally visible by both the server and the ui! Lets update our UI:

Notice how we’re now passing variables for our choices instead of hard coding them in the ui! This might make our code more compact as well! Lastly, we’ll have to update the server. It will be easy to update our function to incorporate our new input (we just have to pass it as an argument to our new parameter), but we should remember we also want the ui to update dynamically when the user changes the selected district. It is important to understand here that we can change the parameters and behaviour of widgets while the app is running, but this needs to be done in the server. We need to understand a new way to output to the server to learn how to do this.

The functions we need to understand how to do this are known as observer functions, and are similar to reactive functions in how they behave. They have one key difference though:

* Reactive functions do not directly affect outputs, and produce objects that can be seen in other locations in the server
* Observer functions can affect server outputs, but do so via side effects of other functions. (They can also do other things, but this is their main function in practice)

Similar to reactive functions, there are two flavours of observer functions, and they are divided by the same logic that divides reactive functions:

1. observe() - this function runs whenever any inputs used inside of it change
2. observeEvent() - this function runs when a user-specified input changes

We also need to understand the shiny-provided functions that update widgets. These are fairly straightforward to run - they first take the session object from the server function (this doesn’t need to be understood for now), and then the inputId of the function to be changed. We then pass new versions of all parameters that are already taken by selectInput() - these will be automatically updated in the widget.

Lets look at an isolated example of how we could use this in our server. When the user changes the district, we want to filter our tibble of facilities by district, and update the choices to only reflect those that are available in that district (and an option for all facilities)

And that’s it! we can add it into our server, and that behaviour will now work. Here’s what our new server should look like:

### Adding another tab with a table

Now we’ll move on to the last component we want to add to our app. We’ll want to separate our ui into two tabs, one of which will have an interactive table where the user can see the data they are making the epidemic curve with. To do this, we can use the packaged ui elements that come with shiny relevant to tabs. On a basic level, we can enclose most of our main panel in this general structure:

Lets apply this to our ui. We also will want to use the **DT** package here - this is a great package for making interactive tables from pre-existing data. We can see it being used for DT::datatableOutput() in this example.

Now our app is arranged into tabs! Lets make the necessary edits to the server as well. Since we dont need to manipulate our dataset at all before we render it this is actually very simple - we just render the malaria\_data dataset via DT::renderDT() to the ui!

## Sharing shiny apps

Now that you’ve developed your app, you probably want to share it with others - this is the main advantage of shiny after all! We can do this by sharing the code directly, or we could publish on a server. If we share the code, others will be able to see what you’ve done and build on it, but this will negate one of the main advantages of shiny - it can eliminate the need for end-users to maintain an R installation. For this reason, if you’re sharing your app with users who are not comfortable with R, it is much easier to share an app that has been published on a server.

If you’d rather share the code, you could make a .zip file of the app, or better yet, publish your app on github and add collaborators. You can refer to the section on github for further information here.

However, if we’re publishing the app online, we need to do a little more work. Ultimately, we want your app to be able to be accessed via a web URL so others can get quick and easy access to it. Unfortunately, to publish you app on a server, you need to have access to a server to publish it on! There are a number of hosting options when it comes to this:

* shinyapps.io: this is the easiest place to publish shiny apps, as it has the smallest amount of configuration work needed, and has some free, but limited licenses.
* RStudio Connect: this is a far more powerful version of an R server, that can perform many operations, including publishing shiny apps. It is however, harder to use, and less recommended for first-time users.

For the purposes of this document, we will use shinyapps.io, since it is easier for first time users. You can make a free account here to start - there are also different price plans for server licesnses if needed. The more users you expect to have, the more expensive your price plan may have to be, so keep this under consideration. If you’re looking to create something for a small set of individuals to use, a free license may be perfectly suitable, but a public facing app may need more licenses.

First we should make sure our app is suitable for publishing on a server. In your app, you should restart your R session, and ensure that it runs without running any extra code. This is important, as an app that requires package loading, or data reading not defined in your app code won’t run on a server. Also note that you can’t have any explicit file paths in your app - these will be invalid in the server setting - using the here package solves this issue very well. Finally, if you’re reading data from a source that requires user-authentication, such as your organisation’s servers, this will not generally work on a server. You will need to liase with your IT department to figure out how to whitelist the shiny server here.

signing up for account

Once you have your account, you can navigate to the tokens page under Accounts. Here you will want to add a new token - this will be used to deploy your app.

From here, you should note that the url of your account will reflect the name of your app - so if your app is called my\_app, the url will be appended as xxx.io/my\_app/. Choose your app name wisely! Now that you are all ready, click deploy - if successful this will run your app on the web url you chose!

something on making apps in documents?

## Further reading

So far, we’ve covered a lot of aspects of shiny, and have barely scratched the surface of what is on offer for shiny. While this guide serves as an introduction, there is loads more to learn to fully understand shiny. You should start making apps and gradually add more and more functionality

## Recommended extension packages

The following represents a selection of high quality shiny extensions that can help you get a lot more out of shiny. In no particular order:

* **shinyWidgets** - this package gives you many many more widgets that can be used in your app. Run shinyWidgets::shinyWidgetsGallery() to see a selection of available widgets with this package. See examples [here](https://github.com/dreamRs/shinyWidgets)
* **shinyjs** - this is an excellent package that gives the user the ability to greatly extend shiny’s utility via a series of javascript. The applications of this package range from very simple to highly advanced, but you might want to first use it to manipulate the ui in simple ways, like hiding/showing elements, or enabling/disabling buttons. Find out more [here](https://deanattali.com/shinyjs/basic)
* **shinydashboard** - this package massively expands the available ui that can be used in shiny, specifically letting the user create a complex dashboard with a variety of complex layouts. See more [here](https://rstudio.github.io/shinydashboard/)
* **shinydashboardPlus** - get even more features out of the **shinydashboard** framework! See more [here](https://rinterface.github.io/shinydashboardPlus/articles/shinydashboardPlus.html)
* **shinythemes** - change the default css theme for your shiny app with a wide range of preset templates! See more [here](https://rstudio.github.io/shinythemes/)

There are also a number of packages that can be used to create interactive outputs that are shiny compatible.

* **DT** is semi-incorporated into base-shiny, but provides a great set of functions to create interactive tables.
* **plotly** is a package for creating interactive plots that the user can manipulate in app. You can also convert your plot to interactive versions via plotly::ggplotly()! As alternatives, **dygraphs** and **highcharter** are also excellent.

## Recommended resources

# VII Miscellaneous

# Writing functions

## Preparation

### Load packages

This code chunk shows the loading of packages required for the analyses. In this handbook we emphasize p\_load() from **pacman**, which installs the package if necessary and loads it for use. You can also load installed packages with library() from **base** R. See the page on [R basics](#r-basics) for more information on R packages.

### Import data

We import the dataset of cases from a simulated Ebola epidemic. If you want to download the data to follow step-by-step, see instructions in the [Download book and data] page. The dataset is imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data.

We will also use in the last part of this page some data on H7N9 flu from 2013.

## Functions

Functions are helpful in programming since they allow to make codes easier to understand, somehow shorter and less prone to errors (given there were no errors in the function itself).

If you have come so far to this handbook, it means you have came across endless functions since in R, every operation is a function call +, for, if, [, $, { …. For example x + y is the same as'+'(x, y)

R is one the languages that offers the most possibility to work with functions and give enough tools to the user to easily write them. We should not think about functions as fixed at the top or at the end of the programming chain, R offers the possibility to use them as if they were vectors and even to use them inside other functions, lists…

Lot of very advanced resources on functional programming exist and we will only give here an insight to help you start with functional programming with short practical examples. You are then encouraged to visit the links on references to read more about it.

## Why would you use a function?

Before answering this question, it is important to note that you have already had tips to get to write your very first R functions in the page on [Iteration, loops, and lists](#iteration-loops-and-lists) of this handbook. In fact, use of “if/else” and loops is often a core part of many of our functions since they easily help to either broaden the application of our code allowing multiple conditions or to iterate codes for repeating tasks.

* I am repeating multiple times the same block of code to apply it to a different variable or data?
* Getting rid of it will it substantially shorten my overall code and make it run quicker?
* Is it possible that the code I have written is used again but with a different value at many places of the code?

If the answer to one of the previous questions is “YES”, then you probably need to write a function

## How does R build functions?

Functions in R have three main components:

* the formals() which is the list of arguments which controls how we can call the function
* the body() that is the code inside the function i.e. within the brackets or following the parenthesis depending on how we write it

and,

* the environment() which will help locate the function’s variables and determines how the function finds value.

Once you have created your function, you can verify each of these components by calling the function associated.

## Basic syntax and structure

* A function will need to be named properly so that its job is easily understandable as soon as we read its name. Actually this is already the case with majority of the base R architecture. Functions like mean(), print(), summary() have names that are very straightforward
* A function will need arguments, such as the data to work on and other objects that can be static values among other options
* And finally a function will give an output based on its core task and the arguments it has been given. Usually we will use the built-in functions as print(), return()… to produce the output. The output can be a logical value, a number, a character, a data frame…in short any kind of R object.

Basically this is the composition of a function:

We can create our first function that will be called contain\_covid19().

We can then verify the components of our newly created function.

Now we will test our function. To call our written function, you use it as you use all R functions i.e by writing the function name and adding the required arguments.

We can write again the name of each argument for precautionary reasons. But without specifying them, the code should work since R has in memory the positioning of each argument. So as long as you put the values of the arguments in the correct order, you can skip writing the arguments names when calling the functions.

Then let’s look what happens if one of the values is "no" or **not** "yes".

If we provide an argument that is not recognized, we get an error:

Error in contain\_covid19(barrier\_gest = "sometimes", wear\_mask = "yes", : could not find function "contain\_covid19"

**NOTE:** Some functions (most of time very short and straightforward) may not need a name and can be used directly on a line of code or inside another function to do quick task. They are called **anonymous functions** .

For instance below is a first anonymous function that keeps only character variables the dataset.

Then another function that selects every second observation of our dataset (may be relevant when we have longitudinal data with many records per patient for instance after having ordered by date or visit). In this case, the proper function writing outside dplyr would be function (x) (x%%2 == 0) to apply to the vector containing all row numbers.

A possible base R code for the same task would be:

**CAUTION:** Though it is true that using functions can help us with our code, it can nevertheless be time consuming to write some functions or to fix one if it has not been thought thoroughly, written adequately and is returning errors as a result. For this reason it is often recommended to first write the R code, make sure it does what we intend it to do, and then transform it into a function with its three main components as listed above.

## Examples

### Return proportion tables for several columns

Yes, we already have nice functions in many packages allowing to summarize information in a very easy and nice way. But we will still try to make our own, in our first steps to getting used to writing functions.

In this example we want to show how writing a simple function would avoid you copy-pasting the same code multiple times.

**TIP:** As shown above, it is very important to comment your functions as you would do for the general programming. Bear in mind that a function’s aim is to make a code ready to read, shorter and more efficient. Then one should be able to understand what the function does just by reading its name and should have more details reading the comments.

A second option is to use this function in another one via a loop to make the process at once:

A simpler way could be using the base R “apply” instead of a “for loop” as expressed below:

**TIP:** R is often defined as a functional programming language and almost anytime you run a line of code you are using some built-in functions. A good habit to be more comfortable with writing functions is to often have an internal look at how the basic functions you are using daily are built. The shortcut to do so is selecting the function name and then clicking onCtrl+F2 or fn+F2 or Cmd+F2 (depending on your computer) .

## Using ****purrr****: writing functions that can be iteratively applied

### Modify class of multiple columns in a dataset

Let’s say many character variables in the original linelist data need to be changes to “factor” for analysis and plotting purposes. Instead of repeating the step several times, we can just use lapply() to do the transformation of all variables concerned on a single line of code.

**CAUTION:** lapply() returns a list, thus its use may require an additional modification as a last step.

The same step can be done using map\_if() function from the **purrr** package

### Iteratively produce graphs for different levels of a variable

We will produce here pie chart to look at the distribution of patient’s outcome in China during the H7N9 outbreak for each province. Instead of repeating the code for each of them, we will just apply a function that we will create.

### Iteratively produce tables for different levels of a variable

Here we will create three indicators to summarize in a table and we would like to produce this table for each of the provinces. Our indicators are the delay between onset and hospitalization, the percentage of recovery and the median age of cases.

## Tips and best Practices for well functioning functions

Functional programming is meant to ease code and facilitates its reading. It should produce the contrary. The tips below will help you having a clean code and easy to read code.

### Naming and syntax

* Avoid using character that could have been easily already taken by other functions already existing in your environment
* It is recommended for the function name to be short and straightforward to understand for another reader
* It is preferred to use verbs as the function name and nouns for the argument names.

### Column names and tidy evaluation

If you want to know how to reference column names that are provided to your code as arguments, read this [tidyverse programming guidance](https://dplyr.tidyverse.org/articles/programming.html). Among the topics covered are tidy evaluation and use of the embrace {{ }} “double braces”

For example, here is a quick skeleton template code from page tutorial mentioned just above:

### Testing and Error handling

The more complicated a function’s task the higher the possibility of errors. Thus it is sometimes necessary to add some verification within the funtion to help quickly understand where the error is from and find a way t fix it.

* It can be more than recommended to introduce a check on the missingness of one argument using missing(argument). This simple check can return “TRUE” or “FALSE” value.
* Use stop() for more detectable errors.
* As we see when we run most of the built-in functions, there are messages and warnings that can pop-up in certain conditions. We can integrate those in our written functions by using the functions message() and warning().
* We can handle errors also by using safely() which takes one function as an argument and executes it in a safe way. In fact the function will execute without stopping if it encounters an error. safely() returns as output a **list** with two objects which are the results and the error it “skipped”.

We can verify by first running the mean() as function, then run it with safely().

As said previously, well commenting our codes is already a good way for having documentation in our work.

## Resources

[R for Data Science link](https://r4ds.had.co.nz/functions.html)

[Cheatsheet advance R programming](https://www.rstudio.com/wp-content/uploads/2016/02/advancedR.pdf)

[Cheatsheet purr Package](https://purrr.tidyverse.org/)

[Video-ACM talk by Hadley Wickham: The joy of functional programming (how does map\_dbl work)](https://youtube.videoken.com/embed/bzUmK0Y07ck)

# Directory interactions

In this page we cover common scenarios where you create, interact with, save, and import with directories (folders).

## Preparation

### ****fs**** package

The **fs** package is a **tidyverse** package that facilitate directory interactions, improving on some of the **base** R functions. In the sections below we will often use functions from **fs**.

### Print directory as a dendrogram tree

Use the function dir\_tree() from **fs**.

Provide the folder filepath to path = and decide whether you want to show only one level (recurse = FALSE) or all files in all sub-levels (recurse = TRUE). Below we use here() as shorthand for the R project and specify its sub-folder “data”, which contains all the data used for this R handbook. We set it to display all files within “data” and its sub-folders (e.g. “cache”, “epidemic models”, “population”, “shp”, and “weather”).

## List files in a directory

To list just the file names in a directory you can use dir() from **base** R. For example, this command lists the file names of the files in the “population” subfolder of the “data” folder in an R project. The relative filepath is provided using here() (which you can read about more in the [Import and export](#import-and-export) page).

To list the full file paths of the directory’s files, you can use you can use dir\_ls() from **fs**. A **base** R alternative is list.files().

To get all the metadata information about each file in a directory, (e.g. path, modification date, etc.) you can use dir\_info() from **fs**.

This can be particularly useful if you want to extract the last modification time of the file, for example if you want to import the most recent version of a file. For an example of this, see the [Import and export](#import-and-export) page.

Here is the data frame returned. Scroll to the right to see all the columns.

## File information

To extract metadata information about a specific file, you can use file\_info() from **fs** (or file.info() from **base** R).

Here we use the $ to index the result and return only the modification\_time value.

## Check if exists

### R objects

You can use exists() from **base** R to check whether an R object exists within R (supply the object name in quotes).

Note that some **base** R packages use generic object names like “data” behind the scenes, that will appear as TRUE unless inherit = FALSE is specified. This is one reason to not name your dataset “data”.

If you are writing a function, you should use missing() from **base** R to check if an argument is present or not, instead of exists().

### Directories

To check whether a directory exists, provide the file path (and file name) to is\_dir() from **fs**. Scroll to the right to see that TRUE is printed.

An alternative is file.exists() from **base** R.

### Files

To check if a specific file exists, use is\_file() from **fs**. Scroll to the right to see that TRUE is printed.

A **base** R alternative is file.exists().

## Create

### Directories

To create a new directory (folder) you can use dir\_create() from **fs**. If the directory already exists, it will not be overwritten and no error will be returned.

An alternative is dir.create() from **base** R, which will show an error if the directory already exists. In contrast, dir\_create() in this scenario will be silent.

### Files

You can create an (empty) file with file\_create() from **fs**. If the file already exists, it will not be over-written or changed.

A **base** R alternative is file.create(). But if the file already exists, this option will truncate it. If you use file\_create() the file will be left unchanged.

### Create if does not exists

UNDER CONSTRUCTION

## Delete

### R objects

Use rm() from **base** R to remove an R object.

### Directories

Use dir\_delete() from **fs**.

### Files

You can delete files with file\_delete() from **fs**.

## Running other files

### source()

To run one R script from another R script, you can use the source() command (from **base** R).

This is equivalent to viewing the above R script and clicking the “Source” button in the upper-right of the script. This will execute the script but will do it silently (no output to the R console) unless specifically intended. See the page on [Interactive console] for examples of using source() to interact with a user via the R console in question-and-answer mode.

### render()

render() is a variation on source() most often used for R markdown scripts. You provide the input = which is the R markdown file, and also the output\_format = (typically either “html\_document”, “pdf\_document”, “word\_document”, "")

See the page on [Reports with R Markdown](#reports-with-r-markdown) for more details. Also see the documentation for render() [here](https://rmarkdown.rstudio.com/docs/reference/render.html) or by entering ?render.

### Run files in a directory

You can create a for loop and use it to source() every file in a directory, as identified with dir().

If you only want to run certain scripts, you can identify them by name like this:

Here is a [comparison](https://cran.r-project.org/web/packages/fs/vignettes/function-comparisons.html) of the **fs** and **base** R functions.

### Import files in a directory

See the page on [Import and export](#import-and-export) for importing and exporting individual files.

Also see the [Import and export](#import-and-export) page for methods to automatically import the most recent file, based on a date in the file name or by looking at the file meta-data.

See the page on [Iteration, loops, and lists](#iteration-loops-and-lists) for an example with the package **purrr** demonstrating:

* Splitting a data frame and saving it out as multiple CSV files
* Splitting a data frame and saving each part as a separate sheet within one Excel workbook
* Importing multiple CSV files and combining them into one dataframe
* Importing an Excel workbook with multiple sheets and combining them into one dataframe

## ****base**** R

See below the functions list.files() and dir(), which perform the same operation of listing files within a specified directory. You can specify ignore.case = or a specific pattern to look for.

If a file is currently “open”, it will display in your folder with a tilde in front, like “~$hospital\_linelists.xlsx”.

## Resources

<https://cran.r-project.org/web/packages/fs/vignettes/function-comparisons.html>

# Version control and collaboration with Git and Github

This chapter presents an overview of using Git to collaborate with others. More extensive tutorials can be found at the bottom in the Resources section.

## What is Git?

Git is a **version control** software that allows tracking changes in a folder. It can be used like the “track change” option in Word, LibreOffice or Google docs, but for all types of files. It is one of the most powerful and most used options for version control.

**Why have I never heard of it? -** While people with a developer background routinely learn to use version control software (Git, Mercurial, Subversion or others), few of us from quantitative disciplines are taught these skills. Consequently, most epidemiologists never hear of it during their studies, and have to learn it on the fly.

**Wait, I heard of Github, is it the same?** - Not exactly, but you often use them together, and we will show you how to. In short:

* **Git** is the version control system, a piece of software. You can use it locally on your computer or to synchronize a folder with a host **website**. By default, one uses a terminal to give Git instructions in command-line.
* You can use a **Git client/interface** to avoid the command-line and perform the same actions (at least for the simple, super common ones).
* If you want to store your folder in a **host website** to collaborate with others, you may create an account at Github, Gitlab, Bitbucket or others.

So you could use the client/interface **Github Desktop**, which uses **Git** in the background to manage your files, both locally on your computer, and remotely on a **Github** server.

## Why use the combo Git and Github?

Using **Git** facilitates:

1. Archiving documented versions with incremental changes so that you can easily revert backwards to any previous state
2. Having parallel branches, i.e. developing/“working” versions with structured ways to integrate the changes after review

This can be done locally on your computer, even if you don’t collaborate with other people. Have you ever:

* regretted having deleted a section of code, only to realize two months later that you actually needed it?
* come back on a project that had been on pause and attempted to remember whether you had made that tricky modification in one of the models?
* had a file model\_1.R and another file model\_1\_test.R and a file model\_1\_not\_working.R to try things out?
* had a file report.Rmd, a file report\_full.Rmd, a file report\_true\_final.Rmd, a file report\_final\_20210304.Rmd, a file report\_final\_20210402.Rmd and cursed your archiving skills?

Git will help with all that, and is worth to learn for that alone.

However, it becomes even more powerful when used with a online repository such as Github to support **collaborative projects**. This facilitates:

* Collaboration: others can review, comment on, and accept/decline changes
* Sharing your code, data, and outputs, and invite feedback from the public (or privately, with your team)

and avoids:

* “Oops, I forgot to send the last version and now you need to redo two days worth of work on this new file”
* Mina, Henry and Oumar all worked at the same time on one script and need to manually merge their changes
* Two people try to modify the same file on Dropbox and Sharepoint and this creates a synchronization error.

### This sounds complicated, I am not a programmer

It can be. Examples of advanced uses can be quite scary. However, much like R, or even Excel, you don’t need to become an expert to reap the benefits of the tool. Learning a small number of functions and notions lets you track your changes, synchronize your files on a online repository and collaborate with your colleagues in a very short amount of time.

Due to the learning curve, emergency context may not be the best of time to learn these tools. But learning can be achieved by steps. Once you acquire a couple of notions, your workflow can be quite efficient and fast. If you are not working on a project where collaborating with people through Git is a necessity, **it is actually a good time to get confident using it** in solo before diving in collaboration.

## Setup

### Install Git

Git is the engine behind the scenes on your computer, which tracks changes, branches (versions), merges, and reverting. **You must first install Git from** [**https://git-scm.com/downloads**](https://git-scm.com/downloads)**.**

### Install an interface (optional but recommended)

Git has its own language of commands, which can be typed into a command line terminal. However, there are many clients/interfaces and as non-developpers, in your day-to-day use, you will rarely need to interact with Git directly and interface usually provide nice visualisation tools for file modifications or branches.

Many options exist, on all OS, from beginner friendly to more complex ones. Good options for beginners include the RStudio Git pane and [Github Desktop](https://desktop.github.com/), which we will showcase in this chapter. Intermediate (more powerfull, but more complex) options include Source Tree, Gitkracken, Smart Git and others.

Quick explanation on [Git clients](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\-%09https:\happygitwithr.com\git-client.html#git-client).

Note: since interfaces actually all use Git internally, you can try several of them, switch from one to another on a given project, use the console punctually for an action your interface does not support, or even perform any number of actions online on Github.

As noted below, you may occasionally have to write Git commands into a terminal such as the RStudio terminal pane (a tab adjacent to the R Console) or the Git Bash terminal.

### Github account

Sign-up for a free account at [github.com](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\github.com).

You may be offered to set-up two-factor authentication with an app on your phone. Read more in the Github [help documents](https://docs.github.com/en/github/authenticating-to-github/securing-your-account-with-two-factor-authentication-2fa).

If you use Github Desktop, you can enter your Gitub credentials after installation following these [steps](https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/authenticating-to-github). If you don’t do it know, credentials will be asked later when you try to clone a project from Github.

## Vocabulary, concepts and basic functions

As when learning R, there is a bit of vocabulary to remember to understand Git. Here are the [basics to get you going](https://www.freecodecamp.org/news/an-introduction-to-git-for-absolute-beginners-86fa1d32ff71/) / [interactive tutorial](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\learngitbranching.js.org). In the next sections, we will show how to use interfaces, but it is good to have the vocabulary and concepts in mind, to build your mental model, and as you’ll need them when using interfaces anyway.

### Repository

A Git repository (“repo”) is a folder that contains all the sub-folders and files for your project (data, code, images, etc.) and their revision histories. When you begin tracking changes in the repository with it, Git will create a hidden folder that contains all tracking information. A typical Git repository is your R Project folder (see handbook page on [R projects](#r-projects)).

We will show how to create (initialize) a Git repository from Github, Github Desktop or Rstudio in the next sections.

### Commits

A commit is a **snapshot** of the project at a given time. When you make a change to the project, you will make a new commit to track the changes (the delta) made to your files. For example, perhaps you edited some lines of code and updated a related dataset. Once your changes are saved, you can bundle these changes together into one “commit”.

Each commit has a unique ID (a hash). For version control purposes, you can revert your project back in time based on commits, so it is best to keep them relatively small and coherent. You will also attach a brief description of the changes called the “commit message”.

Staged changes? To stage changes is to add them to the staging area in preparation for the next commit. The idea is that you can finely decide which changes to include in a given commit. For example, if you worked on model specification in one script, and later on a figure in another script, it would make sense to have two different commits (it would be easier in case you wanted to revert the changes on the figure but not the model).

### Branches

A branch represents an independent line of changes in your repo, a parallel, alternate version of your project files.

Branches are useful to test changes before they are incorporated into the main branch, which is usually the primary/final/“live” version of your project. When you are done experimenting on a branch, you can bring the changes into your main branch, by merging it, or delete it, if the changes were not so successful.

Note: you do not have to collaborate with other people to use branches, nor need to have a remote online repository.

### Local and remote repositories

To clone is to create a copy of a Git repository in another place.

For example, you can clone a online repository from Github locally on your computer, or begin with a local repository and clone it online to Github.

When you have cloned a repository, the project files exist in two places:

* the LOCAL repository on your physical computer. This is where you make the actual changes to the files/code.
* the REMOTE, online repository: the versions of your project files in the Github repository (or on any other web host).

To synchronize these repositories, we will use more functions. Indeed, unlike Sharepoint, Dropbox or other synchronizing software, Git does not automatically update your local repository based or what’s online, or vice-versa. You get to choose when and how to synchronize.

* git fetch downloads the new changes from the remote repository but does not change your local repository. Think of it as checking the state of the remote repository.
* git pull downloads the new changes from the remote repositories and update your local repository.
* When you have made one or several commits locally, you can git push the commits to the remote repository. This sends your changes on Github so that other people can see and pull them if they want to.

## Get started: create a new repository

There are many ways to create new repositories. You can do it from the console, from Github, from an interface.

Two general approaches to set-up are:

* Create a new R Project from an existing or new Github repository (preferred for beginners), or
* Create a Github repository for an existing R project

### Start-up files

When you create a new repository, you can optionally create all of the below files, or you can add them to your repository at a later stage. They would typically live in the “root” folder of the repository.

* A README file is a file that someone can read to understand why your project exists and what else they should know to use it. It will be empty at first, but you should complete it later.
* A .gitignore file is a text file where each line would contain folders or files that Git should ignore (not track changes). Read more about it and see examples [here](https://www.freecodecamp.org/news/gitignore-what-is-it-and-how-to-add-to-repo/).
* You can choose a license for your work, so that other people know under which conditions they can use or reproduce your work. For more information, see the [Creative Commons licenses](https://creativecommons.org/licenses/).

### Create a new repository in Github

To create a new repository, log into Github and look for the green button to create a new repository. This now empty repository can be cloned locally to your computer (see next section).

You must choose if you want your repository to be **public** (visible to everyone on the internet) or **private** (only visible to those with permission). This has important implications if your data are sensitive. If your repository is private you will encounter some quotas in advanced special circumstances, such as if you are using Github actions to automatically run your code in the cloud.

### Clone from a Github repository

You can clone an existing Github repository to create a new local R project on your computer.

The Github repository could be one that already exists and contains content, or could be an empty repository that you just created. In this latter case you are essentially creating the Github repo and local R project at the same time (see instructions above).

Note: if you do not have contributing rights on a Github repository, it is possible to first fork the repository to your profile, and then proceed with the other actions. Forking is explained at the end of this chapter, but we recommend that you read the other sections first.

Step 1: Navigate in Github to the repository, click on the green “**Code**” button and copy the **HTTPS clone URL** (see image below)

The next step can be performed in any interface. We will illustrate with Rstudio and Github desktop.

#### In Rstudio

In RStudio, start a new R project by clicking File > New Project > Version Control > Git

* When prompted for the “Repository URL”, paste the HTTPS URL from Github
* Assign the R project a short, informative name
* Choose where the new R Project will be saved locally
* Check “Open in new session” and click “Create project”

You are now in a new, local, RStudio project that is a clone of the Github repository. This local project and the Github repository are now linked.

#### In Github Desktop

* Click on File > Clone a repository
* Select the URL tab
* Paste the HTTPS URL from Github in the first box
* Select the folder in which you want to have your local repository
* Click “CLONE”

### New Github repo from existing R project

An alternative setup scenario is that you have an existing R project with content, and you want to create a Github repository for it.

1. Create a new, empty Github repository for the project (see instructions above)
2. Clone this repository locally (see HTTPS instructions above)
3. Copy all the content from your pre-existing R project (codes, data, etc.) into this new empty, local, repository (e.g. use copy and paste).
4. Open your new project in RStudio, and go to the Git pane. The new files should register as file changes, now tracked by Git. Therefore, you can bundle these changes as a commit and push them up to Github. Once pushed, the repository on Github will reflect all the files.

See the Github workflow section below for details on this process.

### What does it look like now?

#### In RStudio

Once you have cloned a Github repository to a new R project, you now see in RStudio a “Git” tab. This tab appears in the same RStudio pane as your R Environment:

Please note the buttons circled in the image above, as they will be referenced later (from left to right):

* Button to commit the saved file changes to the local branch (this will open a new window)
* Blue arrow to pull (update your local version of the branch with any changes made on the remote/Github version of that branch)
* Green arrow to push (send any commits/changes for your local version of the branch to the remote/Github version of that branch)
* The Git tab in RStudio
* Button to create a NEW branch using whichever local branch is shown to the right as the base. You almost always want to branch off from the main branch (after you first pull to update the main branch)
* The branch you are currently working in
* Changes you made to code or other files will appear below

#### In Github Desktop

Github Desktop is an independent application that allows you to manage all your repositories. When you open it, the interface allows you to choose the repository you want to work on, and then to perform basic Git actions from there.

## Git + Github workflow

### Process overview

Once you have completed the setup (described above), you will have a Github repo that is connected (cloned) to a local R project. The main branch (created by default) is the so-called “live” version of all the files. When you want to make modifications, it is a good practice to create a new branch from the main branch (like “Make a Copy”). This is a typical workflow in Git because creating a branch is easy and fast.

A typical workflow is as follow:

1. Make sure that your local repository is up-to-date, update it if not
2. Go to the branch you were working on previously, or create a new branch to try out some things
3. Work on the files locally on your computer, make one or several commits to this branch
4. Update the remote version of the branch with your changes (push)
5. When you are satisfied with your branch, you can merge the online version of the working branch into the online “main” branch to transfer the changes

Other team members may be doing the same thing with their own branches, or perhaps contributing commits into your working branch as well.

We go through the above process step-by-step in more detail below. Here is a schematic we’ve developed - it’s in the format of a two-way table so it should help epidemiologists understand.

Here’s [another diagram](https://build5nines.com/introduction-to-git-version-control-workflow/).

Note: until recently, the term “master” branch was used, but it is now referred to as “main” branch.

Image [source](https://build5nines.com/introduction-to-git-version-control-workflow/)

## Create a new branch

When you select a branch to work on, **Git resets your working directory the way it was the last time you were on this branch**.

### In Rstudio Git pane

Ensure you are in the “main” branch, and then click on the purple icon to create a new branch (see image above).

* You will be prompted to name your branch with a one-word descriptive name (can use underscores if needed).
* You will see that locally, you are still in the same R project, but you are no longer working on the “main” branch.
* Once created, the new branch will also appear in the Github website as a branch.

You can visualize branches in the Git Pane in Rstudio after clicking on “History”

### In Github Desktop

The process is very much similar, you are prompted to give your branch a name. After, you will be prompted to “Publish you branch to Github” to make the new branch appear in the remote repo as well.

### In console

What is actually happening behind the scenes is that you create a new branch with git branch, then go to the branch with git checkout (i.e. tell Git that your next commits will occur there). From your git repository:

For more information about using the console, see the section on Git commands at the end.

## Commit changes

Now you can edit code, add new files, update datasets, etc.

Every one of your changes is tracked, once the respective file is saved. Changed files will appear in the RStudio Git tab, in Github Desktop, or using the command git status in the terminal (see below).

Whenever you make substantial changes (e.g. adding or updating a section of code), pause and commit those changes. Think of a commit as a “batch” of changes related to a common purpose. You can always continue to revise a file after having committed changes on it.

Advice on commits: generally, it is better to make small commits, that can be easily reverted if a problem arises, to commit together modifications related to a common purpose. To achieve this, you will find that you should commit often. At the beginning, you’ll probably forget to commit often, but then the habit kicks in.

### In Rstudio

The example below shows that, since the last commit, the R Markdown script “collaboration.Rmd” has changed, and several PNG images were added.

You might be wondering what the yellow, blue, green, and red squares next to the file names represent. Here is a snapshot from the [RStudio cheatsheet](https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf) that explains their meaning. Note that changes with yellow “?” can still be staged, committed, and pushed.

* Press the “Commit” button in the Git tab, which will open a new window (shown below)
* Click on a file name in the upper-left box
* Review the changes you made to that file (highlighted below in green or red)
* “Stage” the file, which will include those changes in the commit. Do this by checking the box next to the file name. Alternatively, you can highlight multiple file names and then click “Stage”
* Write a commit message that is short but descriptive (required)
* Press the “Commit” button. A pop-up box will appear showing success or an error message.

Now you can make more changes and more commits, as many times as you would like

### In Github Desktop

You can see the list of the files that were changed on the left. If you select a text file, you will see a summary of the modifications that were made in the right pane (the view will not work on more complex files like .docs or .xlsx).

To stage the changes, just tick the little box near file names. When you have selected the files you want to add to this commit, give the commit a name, optionally a description and then click on the **commit** button.

### In console

The two functions used behind the scenes are git add to select/stage files and git commit to actually do the commit.

### Amend a previous commit

What happens if you commit some changes, carry on working, and realize that you made changes that should “belong” to the past commit (in your opinion). Fear not! You can append these changes to your previous commit.

In Rstudio, it should be pretty obvious as there is a “Amend previous commit” box on the same line as the COMMIT button.

For some unclear reason, the functionality has not been implemented as such in Github Desktop, but there is a (conceptually awkward but easy) way around. If you have committed **but not pushed** your changes yet, an “UNDO” button appears just under the COMMIT button. Click on it and it will revert your commit (but keep your staged files and your commit message). Save your changes, add new files to the commit if necessary and commit again.

In the console:

Note: think before modifying commits that are already public and shared with your collaborators.

## Pull and push changes up to Github

“First PULL, then PUSH”

It is good practice to fetch and pull before you begin working on your project, to update the branch version on your local computer with any changes that have been made to it in the remote/Github version.

PULL often. Don’t hesitate. Always pull before pushing.

When your changes are made and committed and you are happy with the state of your project, you can push your commits up to the remote/Github version of your branch.

Rince and repeat while you are working on the repository.

**Note:** it is much easier to revert changes that were committed but not pushed (i.e. are still local) than to revert changes that were pushed to the remote repository (and perhaps already pulled by someone else), so it is better to push when you are done with introducing changes on the task that you were working on.

#### In Rstudio

PULL - First, click the “Pull” icon (downward arrow) which fetches and pulls at the same time.

PUSH - Clicking the green “Pull” icon (upward arrow). You may be asked to enter your Github username and password. The first time you are asked, you may need to enter two Git command lines into the Terminal:

* **git config –global user.email “**[**you@example.com**](mailto:you@example.com)**”** (your Github email address), and
* **git config –global user.name “Your Github username”**

To learn more about how to enter these commands, see the section below on Git commands.

**TIP:** Asked to provide your password too often? See these chapters 10 & 11 of this [tutorial](https://happygitwithr.com/credential-caching.html#credential-caching) to connect to a repository using a SSH key (more complicated)

#### In Github Desktop

Click on the “Fetch origin” button to check if there are new commits on the remote repository.

If Git finds new commits on the remote repository, the button will change into a “Pull” button. Because the same button is used to push and pull, you cannot push your changes if you don’t pull before.

You can go to the “History” tab (near the “Changes” tab) to see all commits (yours and others). This is a nice way of acquainting yourself with what your collaborators did. You can read the commit message, the description if there is one, and compare the code of the two files using the diff pane.

Once all remote changes have been pulled, and at least one local change has been committed, you can push by clicking on the same button.

#### Console

Without surprise, the commands are fetch, pull and push.

### I want to pull but I have local work

This can happen sometimes: you made some changes on your local repository, but the remote repository has commits that you didn’t pull.

Git will refuse to pull because it might overwrite your changes. There are several strategies to keep your changes, well described in [Happy Git with R](https://happygitwithr.com/pull-tricky.html), among which the two main ones are: - commit your changes, fetch remote changes, pull them in, resolve conflicts if needed (see section below), and push everything online - stash your changes, which sort of stores them aside, pull, unstash (restore), and then commit, solve any conflicts, and push.

If the files concerned by the remote changes and the files concerned by your local changes do not overlap, Git may solve conflicts automatically.

In Github Desktop, this can be done with buttons. To stash, go to Branch > Stash all changes.

## Merge branch into Main

If you have finished making changes, you can begin the process of merging those changes into the main branch. Depending on your situation, this may be fast, or you may have deliberate review and approval steps involving teammates.

### Locally in Github Desktop

One can merge branches locally using Github Desktop. First, go to (checkout) the branch that will be the recipient of the commits, in other words, the branch you want to update. Then go to the menu Branch > Merge into current branch and click. A box will allow you to select the branch you want to import from.

### In console

First move back to the branch that will be the recipient of the changes. This is usually master, but it could be another branch. Then merge your working branch into master.

[This page](https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging) shows a more advanced example of branching and explains a bit what is happening behind the scenes.

### In Github: submitting pull requests

While it is totally possible to merge two branches locally, or without informing anybody, a merge may be discussed or investigated by several people before being integrated to the master branch. To help with the process, Github offers some discussion features around the merge: the **pull request**.

A pull request (a “PR”) is a request to merge one branch into another (in other words, a request that your working branch be pulled into the “main” branch). A pull request typically involves multiple commits. A pull request usually begins a conversation and review process before it is accepted and the branch is merged. For example, you can read pull request discussions on [dplyr’s github](https://github.com/tidyverse/dplyr/pulls).

You can submit a pull request (PR) directly form the website (as illustrated bellow) or from Github Desktop.

* Go to Github repository (online)
* View the tab “Pull Requests” and click the “New pull request” button
* Select from the drop-down menu to merge your branch into main
* Write a detailed Pull Request comment and click “Create Pull Request”.

In the image below, the branch “forests” has been selected to be merged into “main”:

Now you should be able to see the pull request (example image below):

* Review the tab “Files changed” to see how the “main” branch would change if the branch were merged.
* On the right, you can request a review from members of your team by tagging their Github ID. If you like, you can set the repository settings to require one approving review in order to merge into main.
* Once the pull request is approved, a button to “Merge pull request” will become active. Click this.
* Once completed, delete your branch as explained below.

### Resolving conflicts

When two people modified the same line(s) at the same time, a merge conflict arises. Indeed, Git refuses to make a decision about which version to keep, but it helps you find where the conflict is. **DO NOT PANIC**. Most of the time, it is pretty straightforward to resolve.

For example, on Github:

After the merge raised a conflict, open the file in your favorite editor. The conflict will be indicated by series of characters:

The text between <<<<<<< HEAD and ======= comes from your local repository, and the one between ======= and >>>>>>> from the the other branch (which may be origin, master or any branch of your choice).

You need to decide which version of the code you prefer (or even write a third, including changes from both sides if pertinent), delete the rest and remove all the marks that Git added (<<<<<<< HEAD, =======, >>>>>>> origin/master/your\_branch\_name).

Then, save the file, stage it and commit it : this is the commit that makes the merged version “official”. Do not forget to push afterwards.

The more often you and your collaborators pull and push, the smaller the conflicts will be.

Note: If you feel at ease with the console, there are more [*advanced merging options*](https://git-scm.com/book/en/v2/Git-Tools-Advanced-Merging) (e.g. ignoring whitespace, giving a collaborator priority etc.).

### Delete your branch

Once a branch was merged into master and is no longer needed, you can delete it.

#### Github + Rstudio

Go to the repository on Github and click the button to view all the branches (next to the drop-down to select branches). Now find your branch and click the trash icon next to it. Read more detail on deleting a branch [here](https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests/creating-and-deleting-branches-within-your-repository#deleting-a-branch).

Be sure to also delete the branch locally on your computer. This will not happen automatically.

* From RStudio, make sure you are in the Main branch
* Switch to typing Git commands in the RStudio “Terminal” (the tab adjacent to the R console), and type: **git branch -d branch\_name**, where “branch\_name” is the name of your branch to be deleted
* Refresh your Git tab and the branch should be gone

#### In Github Desktop

Just checkout the branch you want to delete, and go to the menu Branch > Delete.

### Forking

You can fork a project if you would like to contribute to it but do not have the rights to do so, or if you just want to modify it for your personal use. A short description of forking can be found [here](https://guides.github.com/activities/forking/).

On Github, click on the “Fork” button:

This will clone the original repository, but in your own profile. So now, there are two versions of the repository **on Github**: the original one, that you cannot modify, and the cloned version in your profile.

Then, you can proceed to clone your version of the online repository locally on your computer, using any of the methods described in previous sections. Then, you can create a new branch, make changes, commit and push them to your remote repository.

Once you are happy with the result you can create a Pull Request from Github or Github Desktop to begin the conversation with the owners/maintainers of the original repository.

**What if you need some newer commits from the official repository?**

Imagine that someone makes a critical modification to the official repository, which you want to include to your cloned version. It is possible to synchronize your fork with the official repository. It involves using the terminal, but it is not too complicated. You mostly need to remember that: - upstream = the official repository, the one that you could not modify - origin = your version of the repository on your Github profile

You can read [this tutorial](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork) or follow along below:

First, type in your Git terminal (inside your repo):

If you have not yet configured the upstream repository you should see two lines, beginning by origin. They show the remote repo that fetch and push point to. Remember, origin is the conventional nickname for your own version of the repository on Github. For example:

Now, add a new remote repository:

Here the address is the address that Github generates when you clone a repository (see section on cloning). Now you will have four remote pointers:

Now that the setup is done, whenever you want to get the changes from the original (upstream) repository, you just have to go (checkout) to the branch you want to update and type:

If there are conflicts, you will have to solve them, as explained in the Resolving conflicts section.

**Summary**: forking is cloning, but on the Github server side. The rest of the actions are typical collaboration workflow actions (clone, push, pull, commit, merge, submit pull requests…).

Note: while forking is a concept, not a Git command, it also exist on other Web hosts, like [*Bitbucket*](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow).

## What we learned

You have learned how to:

* setup Git to track modifications in your folders,
* connect your local repository to a remote online repository,
* commit changes,
* synchronize your local and remote repositories.

All this should get you going and be enough for most of your needs as epidemiologists. We usually do not have as advanced usage as developers.

However, know that should you want (or need) to go further, Git offers more power to simplify commit histories, revert one or several commits, cherry-pick commits, etc. Some of it may sound like pure wizardry, but now that you have the basics, it is easier to build on it.

Note that while the Git pane in Rstudio and Github Desktop are good for beginners / day-to-day usage in our line of work, they do not offer an interface to some of the intermediate / advanced Git functions. Some more complete interfaces allows you to do more with point-and-click (usually at the cost of a more complex layout).

Remember that since you can use any tool at any point to track your repository, you can very easily install an interface to try it out sometimes, or to perform some less common complex task occasionally, while preferring a simplified interface for the rest of time (e.g. using Github Desktop most of the time, and switching to SourceTree or Gitbash for some specific tasks).

## Git commands

### Recommended learning

To learn Git commands in an interactive tutorial, see [this website](https://learngitbranching.js.org/).

### Where to enter commands

You enter commands in a Git shell.

Option 1 You can open a new Terminal in RStudio. This tab is next to the R Console. If you cannot type any text in it, click on the drop-down menu below “Terminal” and select “New terminal”. Type the commands at the blinking space in front of the dollar sign “$”.

Option 2 You can also open a shell (a terminal to enter commands) by clicking the blue “gears” icon in the Git tab (near the RStudio Environment). Select “Shell” from the drop-down menu. A new window will open where you can type the commands after the dollar sign “$”.

Option 3 Right click to open “Git Bash here” which will open the same sort of terminal, or open Git Bash form your application list. [More beginner-friendly informations on Git Bash](https://happygitwithr.com/shell.html), how to find it and some bash commands you will need.

### Sample commands

Below we present a few common git commands. When you use them, keep in mind which branch is active (checked-out), as that will change the action!

In the commands below, represents a branch name. represents the hash ID of a specific commit. represents a number. Do not type the < or > symbols.

| **Git command** | **Action** |
| --- | --- |
| git branch <name> | Create a new branch with the name |
| git checkout <name> | Switch current branch to |
| git checkout -b <name> | Shortcut to create new branch and switch to it |
| git status | See untracked changes |
| git add <file> | Stage a file |
| git commit -m <message> | Commit currently staged changes to current branch with message |
| git fetch | Fetch commits from remote repository |
| git pull | Pull commits from remote repository in current branch |
| git push | Push local commits to remote directory |
| git switch | An alternative to git checkout that is being phased in to Git |
| git merge <name> | Merge branch into current branch |
| git rebase <name> | Append commits from current branch on to branch |

## Resources

Much of this page was informed by [this “Happy Git with R” website](https://happygitwithr.com/) by Jenny Bryan. There is a very helpful section of this website that helps you troubleshoot common Git and R-related errors.

The [Github.com documentation and start guide](https://docs.github.com/en/github).

The RStudio [“IDE” cheatsheet](https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf) which includes tips on Git with RStudio.

<https://ohi-science.org/news/github-going-back-in-time>

**Git commands for beginners**

An [interactive tutorial](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\learngitbranching.js.org) to learn Git commands.

<https://www.freecodecamp.org/news/an-introduction-to-git-for-absolute-beginners-86fa1d32ff71/>: good for learning the absolute basics to track changes in one folder on you own computer.

Nice schematics to understand branches: <https://speakerdeck.com/alicebartlett/git-for-humans>

**Tutorials covering both basic and more advanced subjects**

<https://tutorialzine.com/2016/06/learn-git-in-30-minutes>

<https://dzone.com/articles/git-tutorial-commands-and-operations-in-git> <https://swcarpentry.github.io/git-novice/> (short course) <https://rsjakob.gitbooks.io/git/content/chapter1.html>

The [Pro Git book](https://git-scm.com/book/en/v2) is considered an official reference. While some chapters are ok, it is usually a bit technical. It is probably a good resource once you have used Git a bit and want to learn a bit more precisely what happens and how to go further.

# Common errors

This page includes a running list of common errors and suggests solutions for troubleshooting them.

## Interpreting error messages

R errors can be cryptic at times, so Google is your friend. Search the error message with “R” and look for recent posts in [StackExchange.com](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\StackExchange.com), [stackoverflow.com](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\stackoverflow.com), [community.rstudio.com](file:///C:\Users\Neale\OneDrive%20-%20Neale%20Batra\Documents\Analytic%20Software\R\Projects\R%20handbook\Epi_R_handbook\community.rstudio.com), twitter (#rstats), and other forums used by programmers to filed questions and answers. Try to find recent posts that have solved similar problems.

If after much searching you cannot find an answer to your problem, consider creating a reproducible example (“reprex”) and posting the question yourself. See the page on [Getting help](#getting-help) for tips on how to create and post a reproducible example to forums.

## Common errors

Below, we list some common errors and potential explanations/solutions. Some of these are borrowed from Noam Ross who analyzed the most common forum posts on Stack Overflow about R error messages (see analysis [here](https://github.com/noamross/zero-dependency-problems/blob/master/misc/stack-overflow-common-r-errors.md))

### Typo errors

Error: unexpected symbol in:

" geom\_histogram(stat = "identity")+

tidyquant::geom\_ma(n=7, size = 2, color = "red" lty"

If you see “unexpected symbol”, check for missing commas

### Package errors

could not find function "x"...

This likely means that you typed the function name incorrectly, or forgot to install or load a package.

Error in select(data, var) : unused argument (var)

You think you are using dplyr::select() but the select() function has been masked by MASS::select() - specify dplyr:: or re-order your package loading so that dplyr is after all the others.

Other common masking errors stem from: plyr::summarise() and stats::filter(). Consider using the [**conflicted** package](https://www.tidyverse.org/blog/2018/06/conflicted/).

Error in install.packages : ERROR: failed to lock directory ‘C:\Users\Name\Documents\R\win-library\4.0’ for modifying

Try removing ‘C:\Users\Name\Documents\R\win-library\4.0/00LOCK’

If you get an error saying you need to remove an “00LOCK” file, go to your “R” library in your computer directory (e.g. R/win-library/) and look for a folder called “00LOCK”. Delete this manually, and try installing the package again. A previous install process was probably interrupted, which led to this.

### Object errors

No such file or directory:

If you see an error like this when you try to export or import: Check the spelling of the file and filepath, and if the path contains slashes make sure they are forward / and not backward \. Also make sure you used the correct file extension (e.g. .csv, .xlsx).

object 'x' not found

This means that an object you are referencing does not exist. Perhaps code above did not run properly?

Error in 'x': subscript out of bounds

This means you tried to access something (an element of a vector or a list) that was not there.

### Function syntax errors

# ran recode without re-stating the x variable in mutate(x = recode(x, OLD = NEW)

Error: Problem with `mutate()` input `hospital`.

x argument ".x" is missing, with no default

i Input `hospital` is `recode(...)`.

This error above (argument .x is missing, with no default) is common in mutate() if you are supplying a function like recode() or replace\_na() where it expects you to provide the column name as the first argument. This is easy to forget.

### Logic errors

Error in if

This likely means an if statement was applied to something that was not TRUE or FALSE.

### Factor errors

#Tried to add a value ("Missing") to a factor (with replace\_na operating on a factor)

Problem with `mutate()` input `age\_cat`.

i invalid factor level, NA generated

i Input `age\_cat` is `replace\_na(age\_cat, "Missing")`.invalid factor level, NA generated

If you see this error about invalid factor levels, you likely have a column of class Factor (which contains pre-defined levels) and tried to add a new value to it. Convert it to class Character before adding a new value.

### Plotting errors

Error: Insufficient values in manual scale. 3 needed but only 2 provided. ggplot() scale\_fill\_manual() values = c(“orange”, “purple”) … insufficient for number of factor levels … consider whether NA is now a factor level…

Can't add x object

You probably have an extra + at the end of a ggplot command that you need to delete.

### R Markdown errors

If the error message contains something like Error in options[[sprintf("fig.%s", i)]], check that your knitr options at the top of each chunk correctly use the out.width = or out.height = and not fig.width= and fig.height=.

### Miscellaneous

Consider whether you re-arranged piped **dplyr** verbs and didn’t replace a pipe in the middle, or didn’t remove a pipe from the end after re-arranging.

## Resources

This is another blog post that lists common [R programming errors faced by beginners](https://www.r-bloggers.com/2016/06/common-r-programming-errors-faced-by-beginners/)

# Getting help

This page covers how to get help by posting a Github issue or by posting a reproducible example (“reprex”) to an online forum.

## Github issues

Many R packages and projects have their code hosted on the website Github.com. You can communicate directly with authors via this website by posting an “Issue”.

Read more about how to store your work on Github in the page [Collaboration and Github].

On Github, each project is contained within a repository. Each repository contains code, data, outputs, help documentation, etc. There is also a vehicle to communicate with the authors called “Issues”.

See below the Github page for the **incidence2** package (used to make epidemic curves). You can see the “Issues” tab highlighted in yellow. You can see that there are 5 open issues.

Once in the Issues tab, you can see the open issues. Review them to ensure your problem is not already being addressed. You can open a new issue by clicking the green button on the right. You will need a Github account to do this.

In your issue, follow the instructions below to provide a minimal, reproducible example. And please be courteous! Most people developing R packages and projects are doing so in their spare time (like this handbook!).

To read more advanced materials about handling issues in your own Github repository, check out the Github [documentation on Issues](https://guides.github.com/features/issues/).

## Reproducible example

Providing a reproducible example (“reprex”) is key to getting help when posting in a forum or in a Github issue. People want to help you, but you have to give them an example that they can work with on their own computer. The example should:

* Demonstrate the problem you encountered
* Be minimal, in that it includes only the data and code required to reproduce your problem
* Be reproducible, such that all objects (e.g. data), package calls (e.g. library() or p\_load()) are included

Also, be sure you do not post any sensitive data with the reprex! You can create example data frames, or use one of the data frames built into R (enter data() to open a list of these datasets).

### The ****reprex**** package

The **reprex** package can assist you with making a reproducible example:

1. **reprex** is installed with **tidyverse**, so load either package
2. Begin an R script that creates your problem, step-by-step, starting from loading packages and data.

Copy all the code to your clipboard, and run the following command:

You will see an HTML output appear in the RStudio Viewer pane. It will contain all your code and any warnings, errors, or plot outputs. This output is also copied to your clipboard, so you can post it directly into a Github issue or a forum post.

* If you set session\_info = TRUE the output of sessioninfo::session\_info() with your R and R package versions will be included
* You can provide a working directory to wd =
* You can read more about the arguments and possible variations at the documentation or by entering ?reprex

In the example above, the ggplot() command did not run because the arguemnt date\_format = is not correct - it should be date\_labels =.

### Minimal data

The helpers need to be able to use your data - ideally they need to be able to create it with code.

To create a minumal dataset, consider anonymising and using only a subset of the observations.

UNDER CONSTRUCTION - you can also use the function dput() to create minimal dataset.

## Posting to a forum

Read lots of forum posts. Get an understanding for which posts are well-written, and which ones are not.

1. First, decide whether to ask the question at all. Have you thoroughly reviewed the forum website, trying various search terms, to see if your question has already been asked?
2. Give your question an informative title (not “Help! this isn’t working”).
3. Write your question:

* Introduce your situation and problem
* Link to posts of similar issues and explain how they do not answer your question
* Include any relevant information to help someone who does not know the context of your work
* Give a minimal reproducible example with your R session information
* Use proper spelling, grammar, punctuation, and break your question into paragraphs so that it is easier to read

1. Monitor your question once posted to respond to any requests for clarification. Be courteous and gracious - often the people answering are volunteering their time to help you. If you have a follow-up question consider whether it should be a separate posted question.
2. Mark the question as answered, if you get an answer that meets the original request. This helps others later quickly recognize the solution.

Read these posts about [how to ask a good question](https://stackoverflow.com/help/how-to-ask) the [Stack overflow code of conduct](https://stackoverflow.com/conduct).

## Resources

Tidyverse page on how to [get help!](https://www.tidyverse.org/help/#:~:text=When%20you%20want%20to%20make,to%20load%20the%20reprex%20package.&text=Enter%20reprex()%20in%20the,preview%20of%20your%20rendered%20reprex.)

Tips on [producing a minimal dataset](https://xiangxing98.github.io/R_Learning/R_Reproducible.nb.html#producing-a-minimal-dataset)

Documentation for the [dput function](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/dput)

# R on network drives

## Overview

Using R on network or “company” shared drives can present additional challenges. This page contains approaches, common errors, and suggestions on troubleshooting gained from our experience working through these issues. These include tips for the particularly delicate situations involving R Markdown.

**Using R on Network Drives: Overarching principles**

1. You must get administrator access for your computer. Setup RStudio specifically to run as administrator.
2. Save packages to a library on a lettered drive (e.g. “C:”) when possible. Use a package library whose path begins with "\" as little as possible.
3. the **rmarkdown** package must **not** be in a "\" package library, as then it can’t connect to TinyTex or Pandoc.

## RStudio as administrator

When you click the RStudio icon to open RStudio, do so with a right-click. Depending on your machine, you may see an option to “Run as Administrator”. Otherwise, you may see an option to select Properties (then there should appear a window with the option “Compatibility”, and you can select a checkbox “Run as Administrator”).

## Useful commands

Below are some useful commands when trying to troubleshoot issues using R on network drives.

You can return the path(s) to package libraries that R is using. They will be listed in the order that R is using to install/load/search for packages. Thus, if you want R to use a different default library, you can switch the order of these paths (see below).

You may want to switch the order of the package libraries used by R. For example if R is picking up a library location that begins with “\" and one that begins with a letter e.g. ”D:". You can adjust the order of .libPaths() with the following code.

If you are having difficulties with R Markdown connecting to Pandoc, begin with this code to find out where RStudio thinks your Pandoc installation is.

If you want to see which library a package is loading from, try the below code:

## Troubleshooting common errors

**“Failed to compile…tex in rmarkdown”**

* Check the installation of TinyTex, or install TinyTex to C: location. See the [R basics](#r-basics) page on how to install TinyTex.

**Internet routines cannot be loaded**

For example, Error in tools::startDynamicHelp() : internet routines cannot be loaded

* Try selecting 32-bit version from RStudio via Tools/Global Options.
  + note: if 32-bit version does not appear in menu, make sure you are not using RStudio v1.2.
* Alternatively, try uninstalling R and re-installing with different bit version (32 instead of 64)

**C: library does not appear as an option when I try to install packages manually**

* Run RStudio as an administrator, then this option will appear.
* To set-up RStudio to always run as administrator (advantageous when using an Rproject where you don’t click RStudio icon to open)… right-click the Rstudio icon

The image below shows how you can manually select the library to install a package to. This window appears when you open the Packages RStudio pane and click “Install”.

**Pandoc 1 error**

If you are getting “pandoc error 1” when knitting R Markdowns scripts on network drives:

* Of multiple library locations, have the one with a lettered drive listed first (see codes above)
* The above solution worked when knitting on local drive but while on a networked internet connection
* See more tips here: <https://ciser.cornell.edu/rmarkdown-knit-to-html-word-pdf/>

**Pandoc Error 83**

The error will look something like this: can't find file...rmarkdown...lua.... This means that it was unable to find this file.

See <https://stackoverflow.com/questions/58830927/rmarkdown-unable-to-locate-lua-filter-when-knitting-to-word>

Possibilities:

1. Rmarkdown package is not installed
2. Rmarkdown package is not findable
3. An admin rights issue.

It is possible that R is not able to find the **rmarkdown** package file, so check which library the **rmarkdown** package lives (see code above). If the package is installed to a library that in inaccessible (e.g. starts with "\") consider manually moving it to C: or other named drive library. Be aware that the **rmarkdown** package has to be able to connect to TinyTex installation, so can not live in a library on a network drive.

**Pandoc Error 61**

For example: Error: pandoc document conversion failed with error 61 or Could not fetch...

* Try running RStudio as administrator (right click icon, select run as admin, see above instructions)
* Also see if the specific package that was unable to be reached can be moved to C: library.

**LaTex error (see below)**

An error like: ! Package pdftex.def Error: File 'cict\_qm2\_2020-06-29\_files/figure-latex/unnamed-chunk-5-1.png' not found: using draft setting. or Error: LaTeX failed to compile file\_name.tex.

* See <https://yihui.org/tinytex/r/#debugging> for debugging tips.
* See file\_name.log for more info.

**Pandoc Error 127**

This could be a RAM (space) issue. Re-start your R session and try again.

**Mapping network drives**

Mapping a network drive can be risky. Consult with your IT department before attempting this.

A tip borrowed from this [forum discussion](https://stackoverflow.com/questions/48161177/r-markdown-openbinaryfile-does-not-exist-no-such-file-or-directory/55616529?noredirect=1#comment97966859_55616529):

How does one open a file “through a mapped network drive”?

* First, you’ll need to know the network location you’re trying to access.
* Next, in the Windows file manager, you will need to right click on “This PC” on the right hand pane, and select “Map a network drive”.
* Go through the dialogue to define the network location from earlier as a lettered drive.
* Now you have two ways to get to the file you’re opening. Using the drive-letter path should work.

**Error in install.packages()**

If you get an error that includes mention of a “lock” directory, for example: Error in install.packages : ERROR: failed to lock directory...

Look in your package library and you will see a folder whose name begins with “00LOCK”. Try the following tips:

* Manually delete the “00LOCK” folder directory from your package library. Try installing the package again.
* You can also try the command pacman::p\_unlock() (you can also put this command in the Rprofile so it runs every time project opens.). Then try installing the package again. It may take several tries.
* Try running RStudio in Administrator mode, and try installing the packages one-by-one.
* If all else fails, install the package to another library or folder (e.g. Temp) and then manually copy the package’s folder over to the desired library.

# Data Table

The handbook focusses on the **dplyr** “verb” functions and the **magrittr** pipe operator %>% as a method to clean and group data, but the **data.table** package offers an alternative method that you may encounter in your R career.

## Intro to data tables

A data table is a 2-dimensional data structure like a data frame that allows complex grouping operations to be performed. The data.table syntax is structured so that operations can be performed on rows, columns and groups.

The structure is **DT[i, j, by]**, separated by 3 parts; the **i, j** and **by** arguments. The **i** argument allows for subsetting of required rows, the **j** argument allows you to operate on columns and the **by** argument allows you operate on columns by groups.

This page will address the following topics:

* Importing data and use of fread() and fwrite()
* Selecting and filtering rows using the **i** argument
* Using helper functions %like%, %chin%, %between%
* Selecting and computing on columns using the **j** argument
* Computing by groups using the **by** argument
* Adding and updating data to data tables using :=

## Load packages and import data

### Load packages

Using the p\_load() function from **pacman**, we load (and install if necessary) packages required for this analysis.

### Import data

This page will explore some of the core functions of **data.table** using the case linelist referenced throughout the handbook.

We import the dataset of cases from a simulated Ebola epidemic. If you want to download the data to follow step-by-step, see instructions in the [Download book and data] page. The dataset is imported using the import() function from the **rio** package. See the page on [Import and export](#import-and-export) for various ways to import data. From here we use data.table() to convert the data frame to a data table.

The fread() function is used to directly import regular delimited files, such as .csv files, directly to a data table format. This function, and its counterpart, fwrite(), used for writing data.tables as regular delimited files are very fast and computationally efficient options for large databases.

The first 20 rows of linelist:

Base R commands such as dim() that are used for data frames can also be used for data tables

## The i argument: selecting and filtering rows

Recalling the **DT[i, j, by]** structure, we can filter rows using either row numbers or logical expressions. The i argument is first; therefore, the syntax **DT[i]** or **DT[i,]** can be used.

The first example retrieves the first 5 rows of the data table, the second example subsets cases are 18 years or over, and the third example subsets cases 18 years old or over but not diagnosed at the Central Hospital:

Using .N in the i argument represents the total number of rows in the data table. This can be used to subset on the row numbers:

### Using helper functions for filtering

Data table uses helper functions that make subsetting rows easy. The %like% function is used to match a pattern in a column, %chin% is used to match a specific character, and the %between% helper function is used to match numeric columns within a prespecified range.

In the following examples we: \* filter rows where the hospital variable contains “Hospital” \* filter rows where the outcome is “Recover” or “Death” \* filter rows in the age range 40-60

## The j argument: selecting and computing on columns

Using the **DT[i, j, by]** structure, we can select columns using numbers or names. The **j** argument is second; therefore, the syntax **DT[, j]** is used. To facilitate computations on the **j** argument, the column is wrapped using either list() or .().

### Selecting columns

The first example retrieves the first, third and fifth columns of the data table, the second example selects all columns except the height, weight and gender columns. The third example uses the .() wrap to select the **case\_id** and **outcome** columns.

### Computing on columns

By combining the **i** and **j** arguments it is possible to filter rows and compute on the columns. Using **.N** in the **j** argument also represents the total number of rows in the data table and can be useful to return the number of rows after row filtering.

In the following examples we: \* Count the number of cases that stayed over 7 days in hospital \* Calculate the mean age of the cases that died at the military hospital \* Calculate the standard deviation, median, mean age of the cases that recovered at the central hospital

Remember using the .() wrap in the j argument facilitates computation, returns a data table and allows for column naming.

## The by argument: computing by groups

The **by** argument is the third argument in the **DT[i, j, by]** structure. The **by** argument accepts both a character vector and the list() or .() syntax. Using the .() syntax in the **by** argument allows column renaming on the fly.

In the following examples we:  
\* group the number of cases by hospital \* in cases 18 years old or over, calculate the mean height and weight of cases according to gender and whether they recovered or died \* in admissions that lasted over 7 days, count the number of cases according to the month they were admitted and the hospital they were admitted to

Data.table also allows the chaining expressions as follows:

In these examples we are following the assumption that a row in the data table is equal to a new case, and so we can use the **.N** to represent the number of rows in the data table. Another useful function to represent the number of unique cases is uniqueN(), which returns the number of unique values in a given input. This is illustrated here:

The answer is 3, as the unique values in the gender column are m, f and N/A. Compare with the base R function unique(), which returns all the unique values in a given input:

To find the number of unique cases in a given month we would write the following:

## Adding and updating to data tables

The := operator is used to add or update data in a data table. Adding columns to your data table can be done in the following ways:

Further complex aggregations are beyond the scope of this introductory chapter, but the idea is to provide a popular and viable alternative to **dplyr** for grouping and cleaning data. The **data.table** package is a great package that allows for neat and readable code.

## Resources

Here are some useful resources for more information: \* <https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html> \* <https://github.com/Rdatatable/data.table> \* <https://s3.amazonaws.com/assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf> \* <https://www.machinelearningplus.com/data-manipulation/datatable-in-r-complete-guide/> \* <https://www.datacamp.com/community/tutorials/data-table-r-tutorial>

You can perform any summary function on grouped data; see the Cheat Sheet here for more info: <https://s3.amazonaws.com/assets.datacamp.com/blog_assets/datatable_Cheat_Sheet_R.pdf>