Mesfin Diro Chaka

Data analysis and Visualization

Participant Manual

Data Analysis and Visualization with R

Participant Training Manual

Mada Walabu University

Bale Goba, Ethiopia
May 24, 2022

Contents

Preface

Acknowledgments

Mada Walabu University

Software information and conventions

This training manual was created using the **knitr** package[1] and the **bookdown** package[2]. The following is the R session information:

```
xfun::session_info()
```

```
## R version 4.2.0 (2022-04-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Catalina 10.15.7
##
## Locale: en_US.UTF-8 / en_US.UTF-8 / en_US.UTF-8 / C / en_US.UTF-8 / en_US.UTF-8
##
## Package version:
##   abind_1.4-5             acs_2.1.4               agridat_1.20
##   askpass_1.1             assertthat_0.2.1        backports_1.4.1
##   base64enc_0.1-3         bit_4.0.4               bit64_4.0.5
##   bitops_1.0-7            blob_1.2.3              bookdown_0.26
```

```
##   boot_1.3.28              brio_1.1.3               broom_0.8.0
##   bslib_0.3.1              cachem_1.0.6             callr_3.7.0
##   car_3.0-13               carData_3.0-5            cellranger_1.1.0
##   checkmate_2.1.0          choroplethr_3.7.0        choroplethrMaps_1.0.1
##   class_7.3-20             classInt_0.4-3           cli_3.3.0
##   clipr_0.8.0              cluster_2.1.3            codetools_0.2-18
##   colorspace_2.0-3         colourpicker_1.1.1       commonmark_1.8.0
##   compiler_4.2.0           corrplot_0.92            cowplot_1.1.1
##   cpp11_0.4.2              crayon_1.5.1             crosstalk_1.2.0
##   curl_4.3.2               data.table_1.14.2        DBI_1.1.2
##   dbplyr_2.1.1             desc_1.4.1               dichromat_2.0-0.1
##   diffobj_0.3.5            digest_0.6.29            dplyr_1.0.9
##   dtplyr_1.2.1             e1071_1.7-9              ellipsis_0.3.2
##   evaluate_0.15            fansi_1.0.3              farver_2.1.0
##   fastmap_1.1.0            fontawesome_0.2.2        forcats_0.5.1
##   foreign_0.8-82           Formula_1.2-4            fs_1.5.2
##   gargle_1.2.0             generics_0.1.2           geojsonsf_2.0.2
##   geometries_0.2.0         ggExtra_0.10.0           ggmap_3.0.0
##   ggplot2_3.3.6            ggpubr_0.4.0             ggrepel_0.9.1
##   ggsci_2.9                ggsignif_0.6.3           ggspatial_1.1.5
##   ggthemes_4.2.4           glue_1.6.2               googledrive_2.0.0
##   googlesheets4_1.0.0      graphics_4.2.0           grDevices_4.2.0
##   grid_4.2.0               gridExtra_2.3            gtable_0.3.0
##   haven_2.5.0              highr_0.9                Hmisc_4.7-0
##   hms_1.1.1                htmlTable_2.4.0          htmltools_0.5.2
##   htmlwidgets_1.5.4        httpuv_1.6.5             httr_1.4.3
##   ids_1.0.1                isoband_0.2.5            jpeg_0.1-9
##   jquerylib_0.1.4          jsonify_1.2.1            jsonlite_1.8.0
##   kableExtra_1.3.4         KernSmooth_2.23-20       knitr_1.39
##   labeling_0.4.2           later_1.3.0              lattice_0.20-45
```

```
##   latticeExtra_0.6-29      lazyeval_0.2.2           leafem_0.2.0
##   leaflet_2.1.1            leaflet.providers_1.9.0  leafsync_0.1.0
##   lifecycle_1.0.1          lme4_1.1.29              lubridate_1.8.0
##   lwgeom_0.2-8             magrittr_2.0.3           maptools_1.1-4
##   markdown_1.1             MASS_7.3.56              Matrix_1.4-1
##   MatrixModels_0.5.0       methods_4.2.0            mgcv_1.8-40
##   mime_0.12                miniUI_0.1.1.1           minqa_1.2.4
##   modelr_0.1.8             munsell_0.5.0            nasaweather_0.1
##   ndjson_0.8.0             nlme_3.1-157             nloptr_2.0.1
##   nnet_7.3-17              numDeriv_2016.8.1.1      openssl_2.0.1
##   parallel_4.2.0           pbkrtest_0.5.1           pillar_1.7.0
##   pkgconfig_2.0.3          pkgload_1.2.4            plyr_1.8.7
##   png_0.1-7                polynom_1.4.1            praise_1.0.0
##   prettymapr_0.2.3         prettyunits_1.1.1        processx_3.5.3
##   progress_1.2.2           promises_1.2.0.1         proxy_0.4-26
##   ps_1.7.0                 purrr_0.3.4              quantreg_5.93
##   R6_2.5.1                 rapidjsonr_1.2.0         rappdirs_0.3.3
##   raster_3.5-15            RColorBrewer_1.1-3       Rcpp_1.0.8.3
##   RcppEigen_0.3.3.9.2      RCurl_1.98-1.6           readr_2.1.2
##   readxl_1.4.0             rematch_1.0.1            rematch2_2.1.2
##   reprex_2.0.1             reshape2_1.4.4           rgdal_1.5-32
##   RgoogleMaps_1.4.5.3      rjson_0.2.21             RJSONIO_1.3-1.6
##   rlang_1.0.2              rmarkdown_2.14           rnaturalearth_0.1.0
##   rnaturalearthdata_0.1.0  rnaturalearthhires_0.2.0 rosm_0.2.5
##   rpart_4.1.16             rprojroot_2.0.3          rstatix_0.7.0
##   rstudioapi_0.13          rvest_1.0.2              s2_1.0.7
##   sass_0.4.1               scales_1.2.0             selectr_0.4.2
##   sf_1.0-7                 sfheaders_0.4.0          shiny_1.7.1
##   shinyjs_2.1.0            sourcetools_0.1.7        sp_1.4-7
##   SparseM_1.81             splines_4.2.0            stars_0.5-5
```

```
##   stats_4.2.0              streamR_0.4.5            stringi_1.7.6
##   stringr_1.4.0            survival_3.3-1           svglite_2.1.0
##   sys_3.4                  systemfonts_1.0.4        terra_1.5-21
##   testthat_3.1.4           tibble_3.1.7             tidycensus_1.2.1
##   tidyr_1.2.0              tidyselect_1.1.2         tidyverse_1.3.1
##   tigris_1.6               tinytex_0.39             tmap_3.3-3
##   tmaptools_3.1-1          tools_4.2.0              tzdb_0.3.0
##   units_0.8-0              utf8_1.2.2               utils_4.2.0
##   uuid_1.1-0               vctrs_0.4.1              viridis_0.6.2
##   viridisLite_0.4.0        visreg_2.7.0             vroom_1.5.7
##   waldo_0.4.0              WDI_2.7.6                webshot_0.5.3
##   widgetframe_0.3.1        withr_2.5.0              wk_0.6.0
##   xfun_0.31                XML_3.99-0.9             xml2_1.3.3
##   xtable_1.8-4             yaml_2.3.5
```

Part I

Part 1

1

Data Organization with Spreadsheet

- Any research project's cornerstone is good data organization. Because most researchers keep their data in spreadsheets, many research initiatives begin there.
- In spreadsheets, we organize data in the ways that we wish to work with it as humans, while computers demand that data be organized in specific ways. We must organize our data in the way that computers require it in order to employ tools that make computation more efficient, such as programming languages like R or Python. Since most research projects begin here, we'd like to begin here as well!
- However, you will not learn about data analysis with spreadsheets in this lesson. As a researcher, you'll spend a lot of time organizing data so you can do good analysis afterwards. It isn't the most enjoyable task, but it is important. This session will teach you how to think about data organization and some data wrangling best practices. This method allows you to properly prepare existing data and organize new data collection, resulting in less data wrangling.

1.1 Spreadsheet Programs

- Good data organization is the foundation of your research project. Most researchers have data or do data entry in spreadsheets. Spreadsheet programs

are very useful graphical interfaces for designing data tables and handling very basic data quality control functions.

- Spreadsheet programs encompass a lot of the things we need to be able to do as researchers. We can use them for:
  - Data entry
  - Organizing data
  - Subsetting and sorting data
  - Statistics
  - Plotting

1.2 Formatting data tables

- The most common mistake is using spreadsheet tools as lab notebooks, relying on context, margin notes, and data and field spatial layout to convey information. We can (usually) interpret these things as humans, but computers don't see information in the same way, and unless we explain everything to the computer (which can be difficult!), it won't be able to see how our data fits together.
- We can handle and analyze data much more effectively and quickly with the help of computers, but we must first prepare our data so that the computer can interpret it (and computers are very literal).

Why aren't we teaching data analysis in spreadsheets:

- Data analysis in spreadsheets normally requires a significant amount of human labor. You normally have to redo everything by hand if you want to alter a parameter or run an analysis with a different dataset. (We know you can make macros; nevertheless, see the next point.)

1.2.1 Problems with Spreadsheets

Spreadsheets are good for data entry, but we use them for much more than that. Use them to create data tables for publications, generate statistics and make figures. **Spreadsheets can be used with caution if you want to replicate your steps in another person's work**.

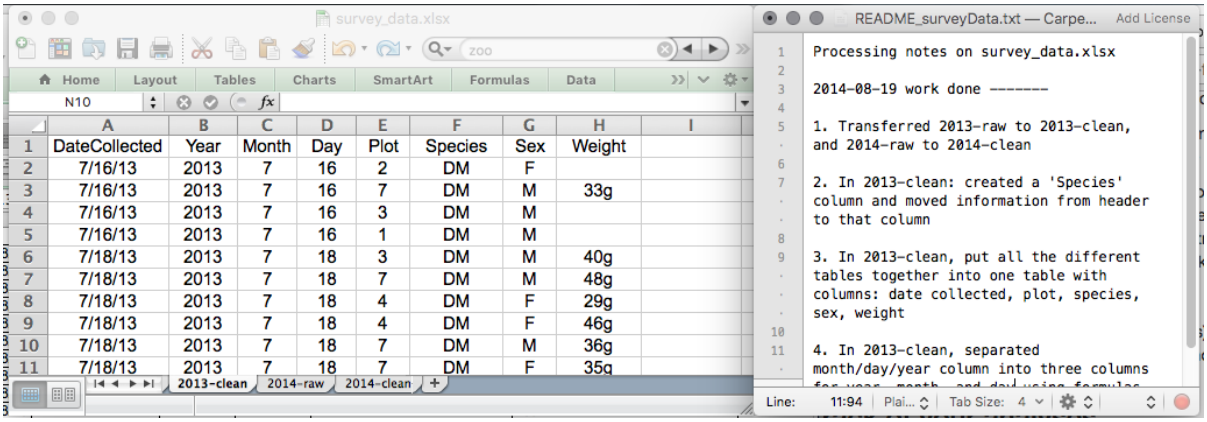
1.2.2 Use of Spreadsheets for Data Entery and Cleaning

- However, there are times when you'll want to use a spreadsheet tool to make "quick and dirty" calculations or figures, and data cleaning will make it easier to do so. Data cleaning also improves the format of your data before it is imported into a statistical analysis program. We'll show you how to use various spreadsheet tools to double-check your data quality and get preliminary summary statistics.
- In this lesson, we will assume that you are most likely using Excel as your primary spreadsheet program - there are others (gnumeric, Calc from OpenOffice), and their functionality is similar, but Excel seems to be the program most used by everyone.

1.2.3 Keeping track of your analyses

It's very simple to end up with a spreadsheet that looks nothing like the one you started with when dealing with spreadsheets for data cleaning or analytics. If a reviewer or instructor requests a new analysis, you should be able to recreate your analyses or figure out what you performed.

- Make a new file containing the data you've cleaned or examined. If you change the original dataset, you'll lose track of where you started!
- Keep a record of the steps you did to clean up or analyze your data. These stages should be tracked just like any other phase in an experiment. This should be done in a plain text file located in the same folder as the data file.



	A	B	C	D	E	F	G	H
1	DateCollected	Year	Month	Day	Plot	Species	Sex	Weight
2	7/16/13	2013	7	16	2	DM	F	
3	7/16/13	2013	7	16	7	DM	M	33g
4	7/16/13	2013	7	16	3	DM	M	
5	7/16/13	2013	7	16	1	DM	M	
6	7/18/13	2013	7	18	3	DM	M	40g
7	7/18/13	2013	7	18	7	DM	M	48g
8	7/18/13	2013	7	18	4	DM	F	29g
9	7/18/13	2013	7	18	4	DM	F	46g
10	7/18/13	2013	7	18	7	DM	M	36g
11	7/18/13	2013	7	18	7	DM	F	35g

Figure 1.1 Keeping track of your analyses

1.2.4 Structuring data in spreadsheets

The cardinal rules of using spreadsheet programs for data:

1. Put all your variables in columns - the thing you're measuring, like 'weight' or 'temperature'.
2. Put each observation in its own row.
3. Don't combine multiple pieces of information in one cell. Sometimes it just seems like one thing, but think if that's the only way you'll want to be able to use or sort that data.
4. Leave the raw data raw - don't change it!
5. Export the cleaned data to a text-based format like CSV (comma-separated values) format. This ensures that anyone can use the data, and is required by most data repositories.

Exercise:

What is the issue with the following data entry according to spreadsheet data structure rules above?

1.2.5 Columns for variables and rows for observations

The rule of thumb, when setting up a datasheet, is columns = variables, rows = observations, cells = data (values).

So, instead we should have:

Date collected	Plot	Species-Sex	Weight
1/9/78	1	DM-M	40
1/9/78	1	DM-F	36
1/9/78	1	DS-F	135
1/20/78	1	DM-F	39
1/20/78	2	DM-M	43
1/20/78	2	DS-F	144
3/13/78	2	DM-F	51
3/13/78	2	DM-F	44
3/13/78	2	DS-F	146

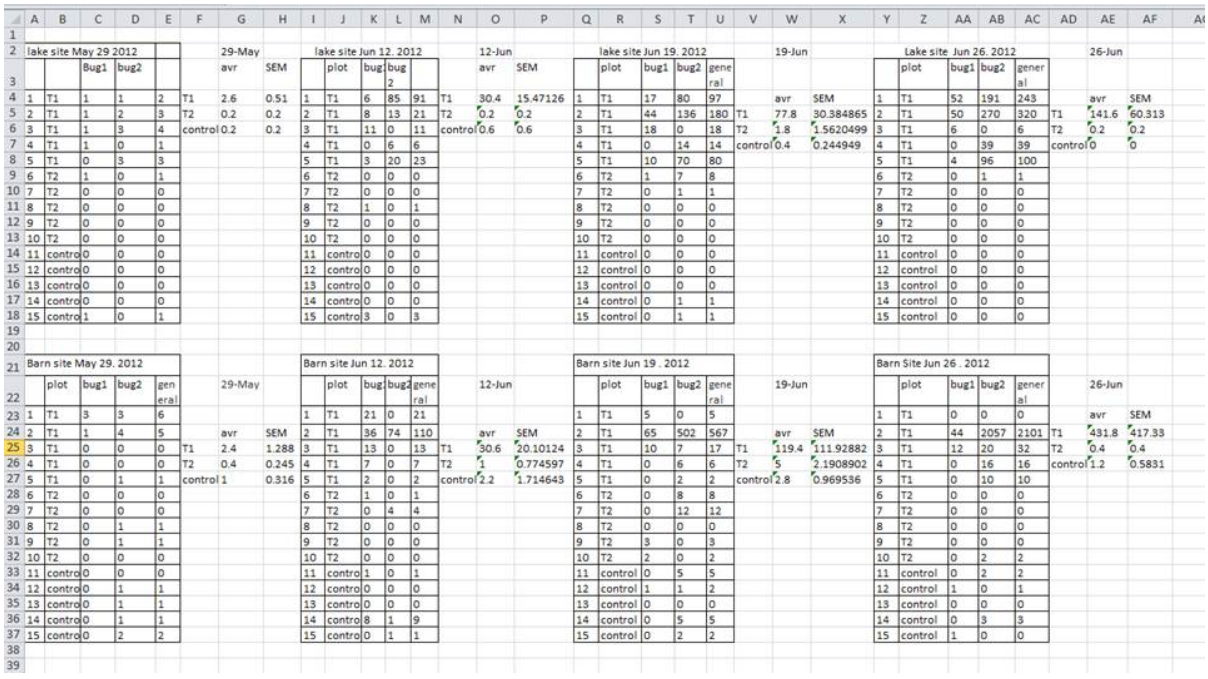
Figure 1.2 multiple-info example

1.3 Common Spreadsheet Errors

- Don't create multiple data tables within a single spreadsheet. This creates false associations between things for the computer, and makes it harder to sort and sort your data in the right order.

Date collected	Plot	Species	Sex	Weight
1/9/78	1	DM	M	40
1/9/78	1	DM	F	36
1/9/78	1	DS	F	135
1/20/78	1	DM	F	39
1/20/78	2	DM	M	43
1/20/78	2	DS	F	144
3/13/78	2	DM	F	51
3/13/78	2	DM	F	44
3/13/78	2	DS	F	146

Figure 1.3 single-info example



In the table below, each row in A-F represents four distinct samples. This means that the columns A-AF do not all refer to the same sample.

1.3.1 Using multiple tabs

- When you create extra tabs, you fail to allow the computer to see connections in the data that are there (you have to introduce spreadsheet application-specific functions or scripting).

1.3.2 Not filling in zeros

There's a difference between a zero and a blank cell in a spreadsheet. To the computer, a zero is actually data. By not entering the value of an observation, you are telling your computer to represent that data as unknown(otherwise known as a null value). This can cause problems with subsequent calculations or analyses.

1.3.3 Using problematic null values

There are a few reasons why null values get represented differently within a dataset. Sometimes confusing null values are automatically recorded from the measuring device. It's a problem if unknown or missing data is recorded as -999, 999, or 0.

Many statistical applications won't detect them as missing (null) values. The tools you select to evaluate your data will determine how these values are interpreted. It is critical to utilize a null indicator that is clearly defined and consistent. As a result, Blanks (for most applications) and NA (for R) are excellent options.

1.4 Date as data

- It is much safer to store dates with [MONTH, DAY and YEAR] in separate columns
- Excel is unable to parse dates from before 1899-12-31, and will thus leave these untouched.
- If you're mixing historic data from before and after this date, Excel will translate only the post-1900 dates into its internal format

- Ideally, data should be as unambiguous as possible.

	A	B	C	D	E	F	G	
1	**What I typed in**	**day-month**	**DOW, month, day, year**	**month-year**	**Initial-year**	**M/D/YYYY**	**DD/MM/YYYY**	**DD**
2	2-jul	2-Jul	Wednesday, July 02, 2014	Jul-14	J-14	7/2/2014	02/07/2014	
3	Jul-14	14-Jul	Monday, July 14, 2014	Jul-14	J-14	7/14/2014	14/07/2014	
4	1-jan-1900	1-Jan	Sunday, January 01, 1900	Jan-00	J-00	1/1/1900	01/01/1900	
5								

Figure 1.4 Many formats, many ambiguities

2

Data Cleaning with OpenRefine

Cleaning data is required before it can be analyzed. To provide consistent data, data cleaning identifies mistakes and corrects formatting. This procedure must be done with extreme caution since without clean data, analysis results may be inaccurate and non-reproducible.

OpenRefine is a robust free and open source tool for dealing with untidy data, such as cleaning it and converting it to a different format.

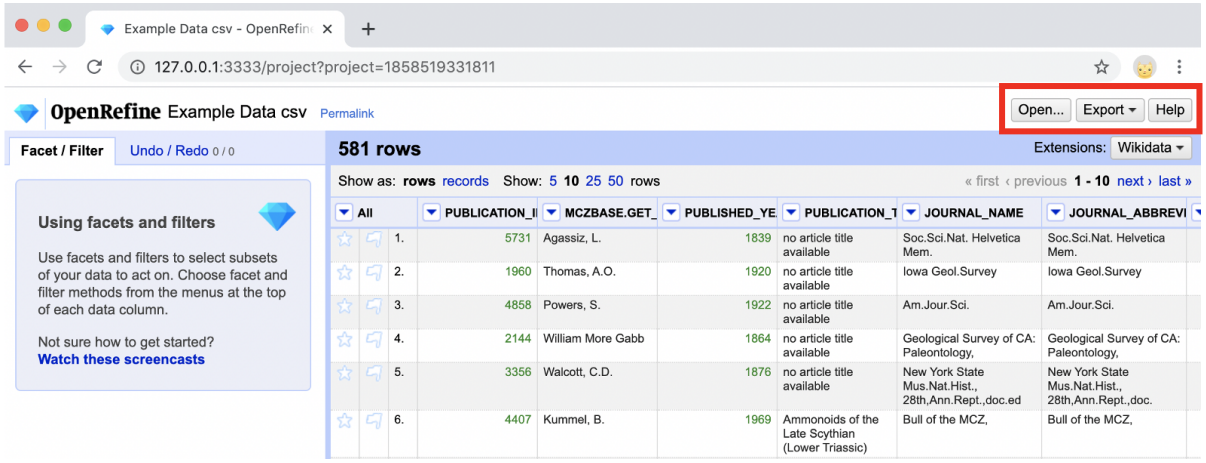
This course will show you how to use OpenRefine to successfully clean and format data while keeping track of any changes you make. Many users say that using this program saves them months of time compared to doing these modifications by hand.

2.1 Layout

Once you have imported your data, it is important to familiarize yourself with OpenRefine's layout.

1. In the top right corner there are three buttons:

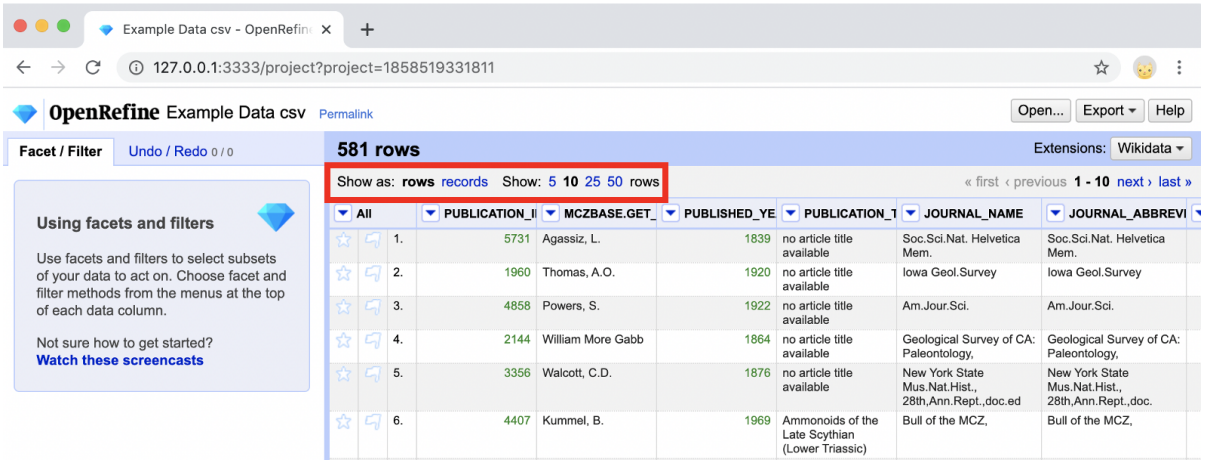
a. "Open..." returns you to the home screen where you can select projects.
b. "Export" opens a dropdown menu of options to export your data.
c. "Help" opens the OpenRefine User Documentation in a new tab in your browser.



2. Below the bolded header stating how many rows/records there are two options:

a. “Show as” allows you to change the grid view between rows and records. For more information on the difference between rows and records, see the explanation of Records and Rows below.

b. “Show” allows you to change the number of rows/records visible in the grid view.

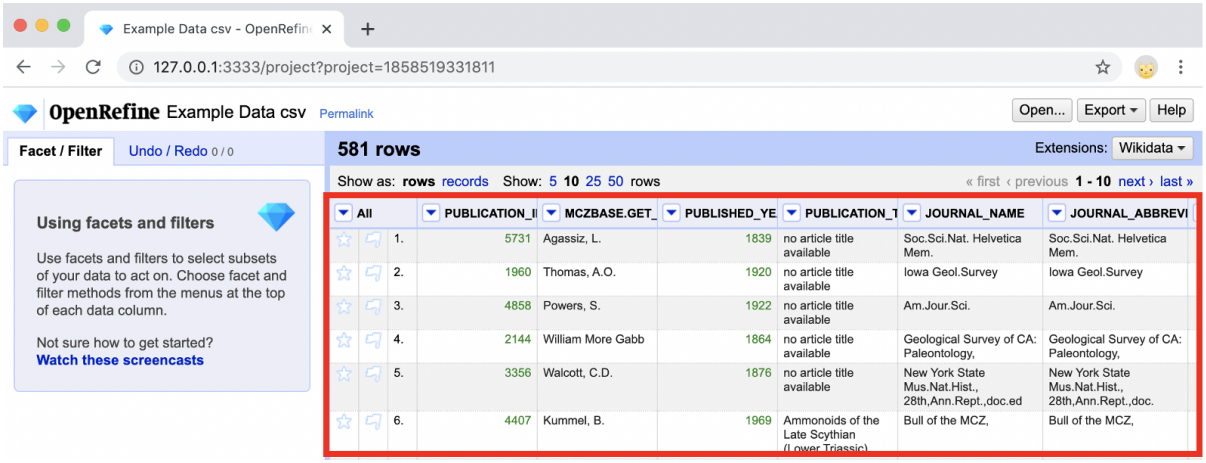


3. In the center of the page is your data in the grid view, which looks similar to Excel. Features of the grid view include:

a. Column headings with dropdown arrows for chosing functions

b. Row/Record numbers and alternate row/record shading

c. Selectable flags and stars



4. On the left, there is a pane with two tabs:

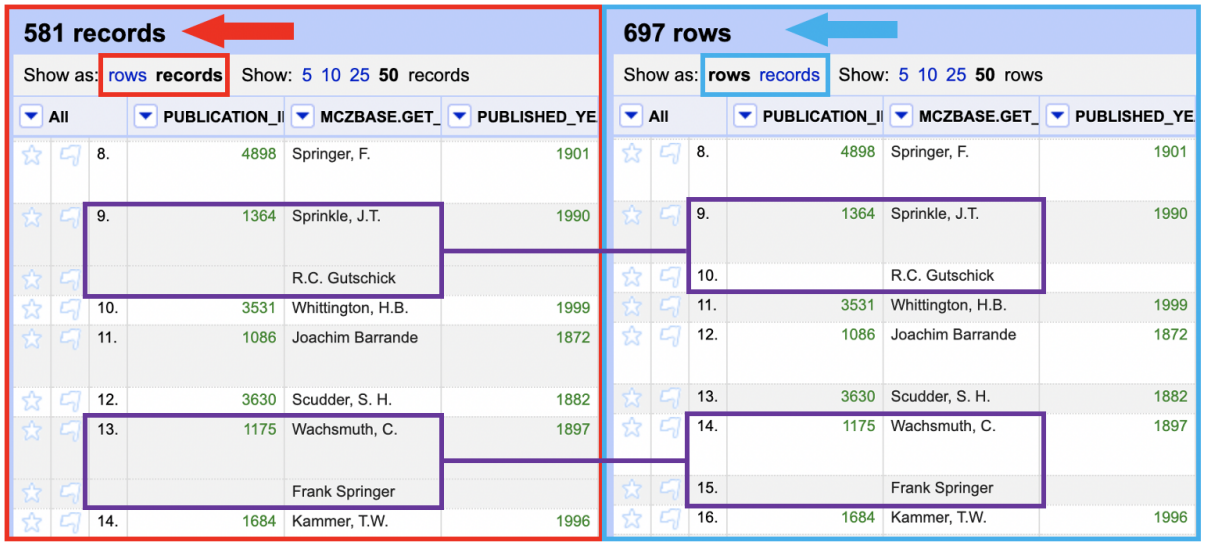
a. b"Facet/Filter" allows you to work on selected sections of your data, including faceting, clustering, and filtering.

b. "Undo/Redo" tracks and stores your history, allows you to undo or redo transformations, and export a JSON file of your transformations.

2.2 Records and Rows

There are two settings for the grid view in OpenRefine: rows or records.

The difference between rows and records is that "rows" display your data in individual lines, each numbered separately, while "records" display your data in multi-line groupings depending on the relationships between the data in those lines. For example:



This data has been transformed using "split multi-valued cells" on the author field to separate different authors into their own lines. On the left, the data is displayed as "records," showing the different lines with the multiple authors grouped together. On the right, the data is displayed as "rows," showing each of the multiple authors as a separate line.

NOTE: Take caution when permanently renumbering rows or records and be aware of what setting you are viewing your data under.

2.3 Changing the Grid View

You can adjust the grid display in numerous ways to make it easier to compare, analyse, and associate data columns. Some adjustments can be permanent or non-permanent, and they can help you organize your data better overall.

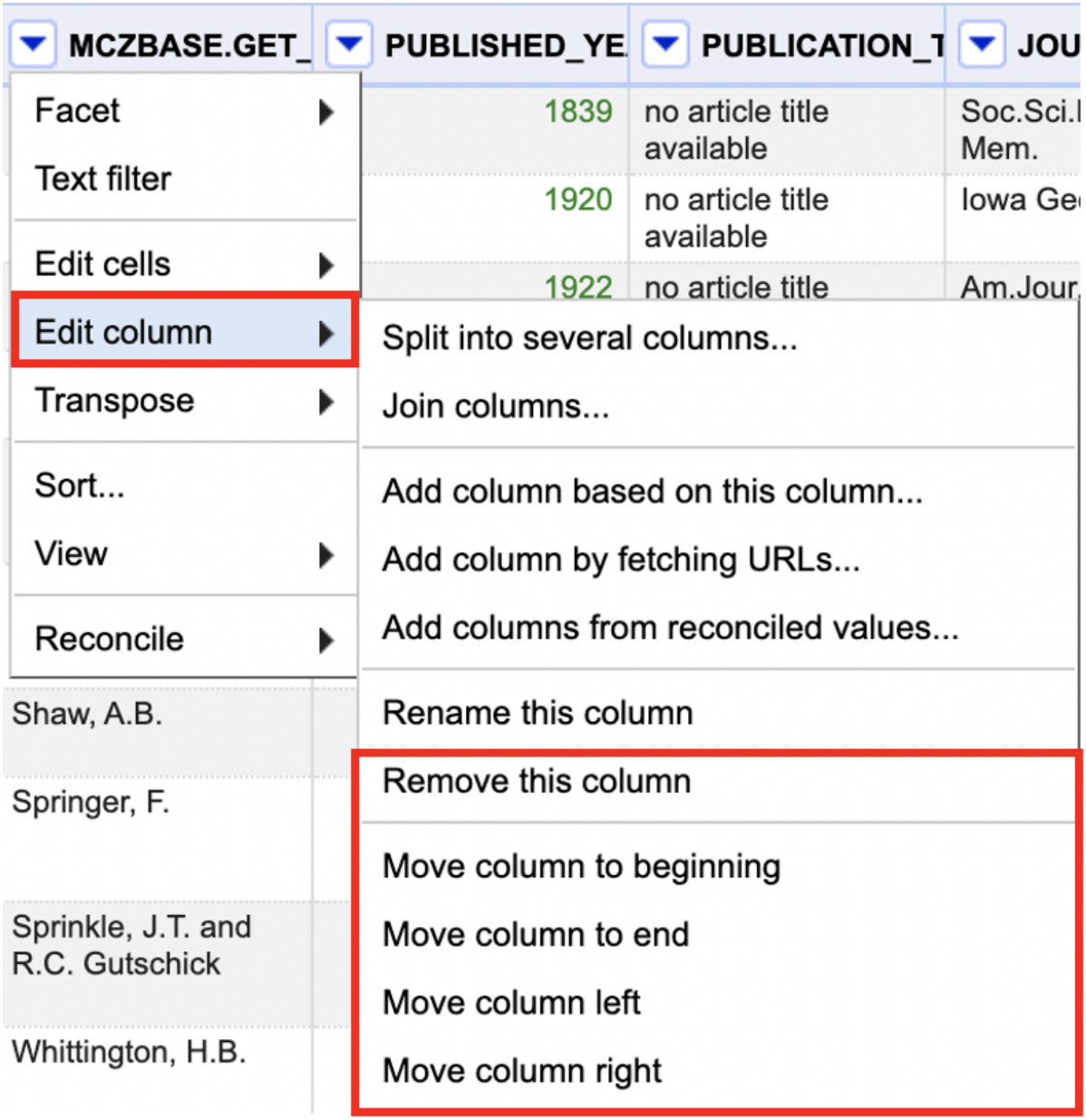
2.3.1 Permanent Changes

Rearranging columns, eliminating columns, and/or renaming columns are all permanent modifications to the grid view.

Although the instructions in Reordering Columns can be used to reorder and remove columns, this method produces the same effects for modest, rapid changes.

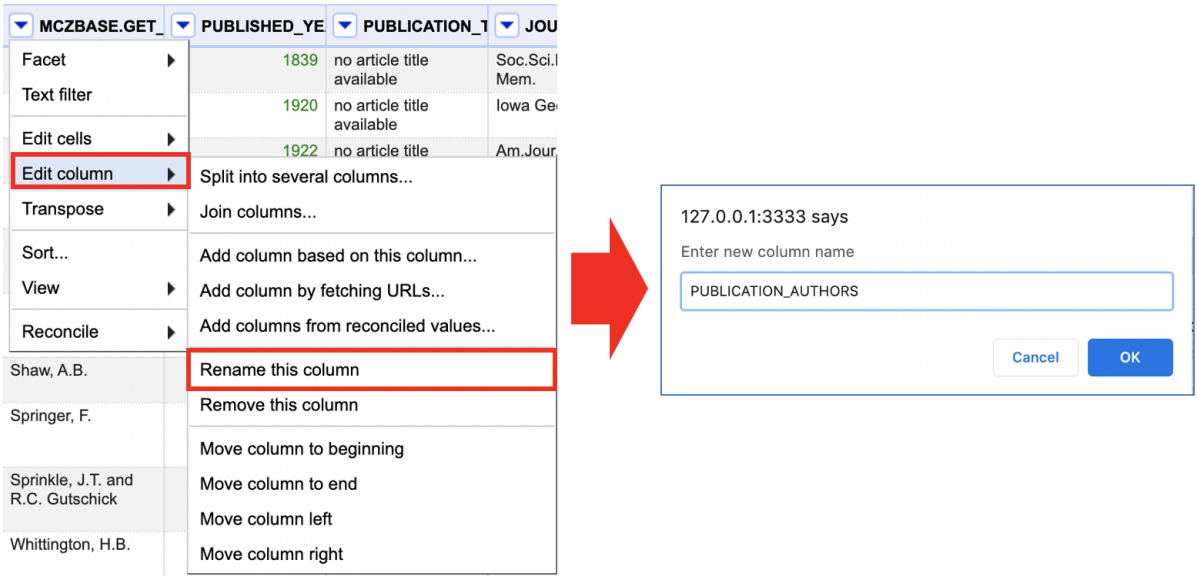
1. Go to the column you would like to rearrange or remove and click the arrow button on the column header.
2. Select the “Edit columns” option and choose the option that best fits from the dropdown:

a. Remove this column
b. Move column to beginning
c. Move column to end
d. Move column left
e. Move column right



Changes to the column name can be done in a few simple steps:

1. Go to the column you would like to rename and click the arrow button on the column header.
2. Select the "Edit columns" option and click on "Rename this column."
3. Change the column name in the pop-up window and click "OK."



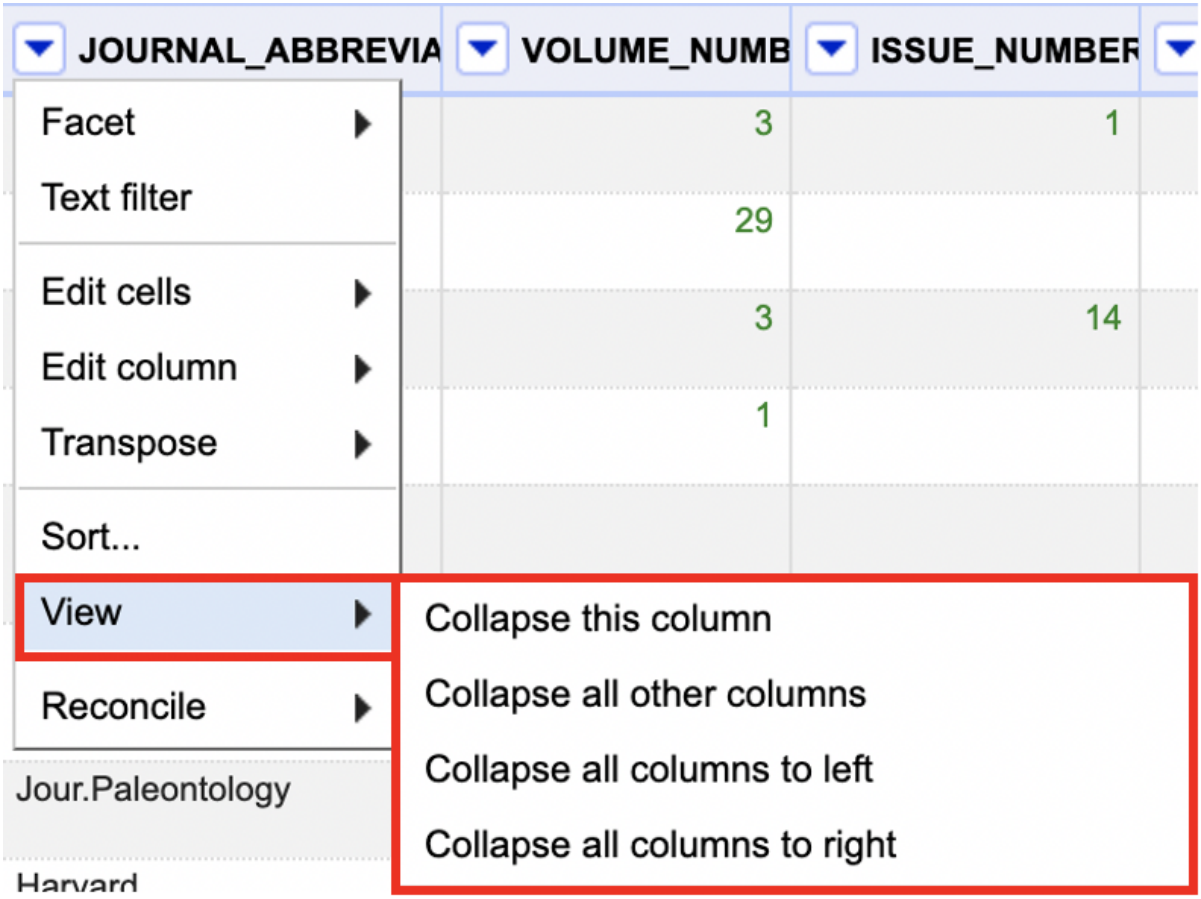
NOTE: In your tracked history, changes to column order or name, as well as the removal of columns, are logged as transformations. Follow the instructions in History and Undo/Redo to undo moving a column.

2.3.2 Non-Permanent Changes

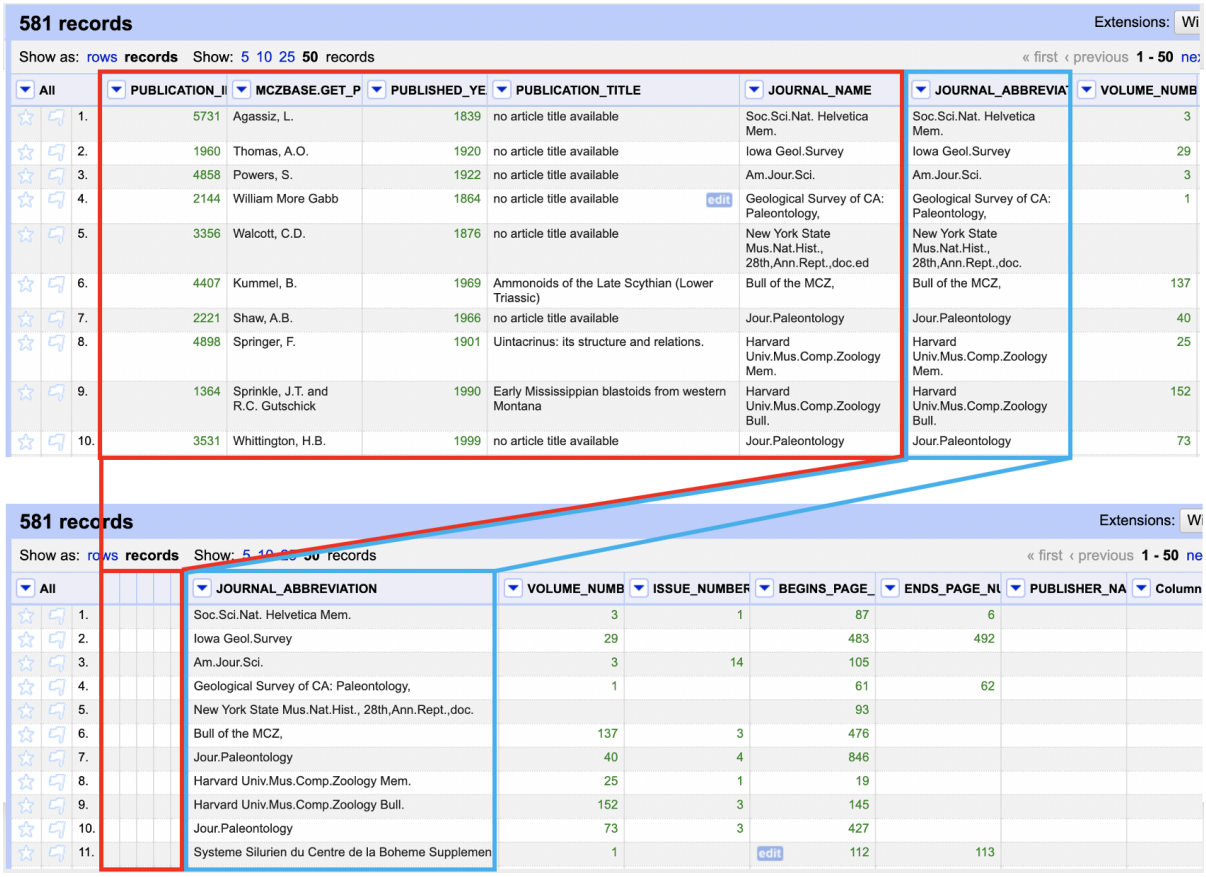
Collapsing columns in one of four ways is a non-permanent alteration to the grid view: collapsing the current column, all other columns, all columns to the left of the current column, or all columns to the right of the current column.

1. Go to the column you would like to change and click the arrow button on the column header.
2. Select the "View" option and pick the best option from the dropdown:

a. Collapse this column
b. Collapse all other columns
c. Collapse all columns to left (see second image below)
d. Collapse all columns to right



3. The collapsed column(s) are replaced by a narrow placeholder.
4. To reopen collapsed columns, click on the placeholder.



581 records

Extensions: Wi

Show as: rows **records** Show: 5 10 25 **50** records

« first ‹ previous **1 - 50** ne:

All		PUBLICATION_I	MCZBASE.GET_P	PUBLISHED_YE	PUBLICATION_TITLE	JOURNAL_NAME	JOURNAL_ABBREVIA	VOLUME_NUMB
1.		5731	Agassiz, L.	1839	no article title available	Soc.Sci.Nat. Helvetica Mem.	Soc.Sci.Nat. Helvetica Mem.	3
2.		1960	Thomas, A.O.	1920	no article title available	Iowa Geol.Survey	Iowa Geol.Survey	29
3.		4858	Powers, S.	1922	no article title available	Am.Jour.Sci.	Am.Jour.Sci.	3
4.		2144	William More Gabb	1864	no article title available edit	Geological Survey of CA: Paleontology,	Geological Survey of CA: Paleontology,	1
5.		3356	Walcott, C.D.	1876	no article title available	New York State Mus.Nat.Hist., 28th,Ann.Rept.,doc.ed	New York State Mus.Nat.Hist., 28th,Ann.Rept.,doc.	
6.		4407	Kummel, B.	1969	Ammonoids of the Late Scythian (Lower Triassic)	Bull of the MCZ,	Bull of the MCZ,	137
7.		2221	Shaw, A.B.	1966	no article title available	Jour.Paleontology	Jour.Paleontology	40
8.		4898	Springer, F.	1901	Uintacrinus: its structure and relations.	Harvard Univ.Mus.Comp.Zoology Mem.	Harvard Univ.Mus.Comp.Zoology Mem.	25
9.		1364	Sprinkle, J.T. and R.C. Gutschick	1990	Early Mississippian blastoids from western Montana	Harvard Univ.Mus.Comp.Zoology Bull.	Harvard Univ.Mus.Comp.Zoology Bull.	152
10.		3531	Whittington, H.B.	1999	no article title available	Jour.Paleontology	Jour.Paleontology	73

581 records

Extensions: W

Show as: rows **records** Show: 5 10 25 50 records

« first ‹ previous **1 - 50** ne

All		JOURNAL_ABBREVIATION	VOLUME_NUMB	ISSUE_NUMBER	BEGINS_PAGE_	ENDS_PAGE_NU	PUBLISHER_NA	Column
1.		Soc.Sci.Nat. Helvetica Mem.	3	1	87	6		
2.		Iowa Geol.Survey	29		483	492		
3.		Am.Jour.Sci.	3	14	105			
4.		Geological Survey of CA: Paleontology,	1		61	62		
5.		New York State Mus.Nat.Hist., 28th,Ann.Rept.,doc.			93			
6.		Bull of the MCZ,	137	3	476			
7.		Jour.Paleontology	40	4	846			
8.		Harvard Univ.Mus.Comp.Zoology Mem.	25	1	19			
9.		Harvard Univ.Mus.Comp.Zoology Bull.	152	3	145			
10.		Jour.Paleontology	73	3	427			
11.		Systeme Silurien du Centre de la Boheme Supplemen	1		edit 112	113		

NOTE: Collapsing columns does not permanently delete columns from your data, it only hides the selected columns from view. To permanently delete columns, follow the instructions in Reordering Columns.

2.4 Data Mining and Discovery

OpenRefine includes a number of tools that allow you to quickly search, organize, and contextualize your data without having to go through it manually. Facets and filters based on data type (text, numeric, or date) can be used to sort, arrange, and contextualize your data for better discovery and analysis while maintaining the data's original structure.

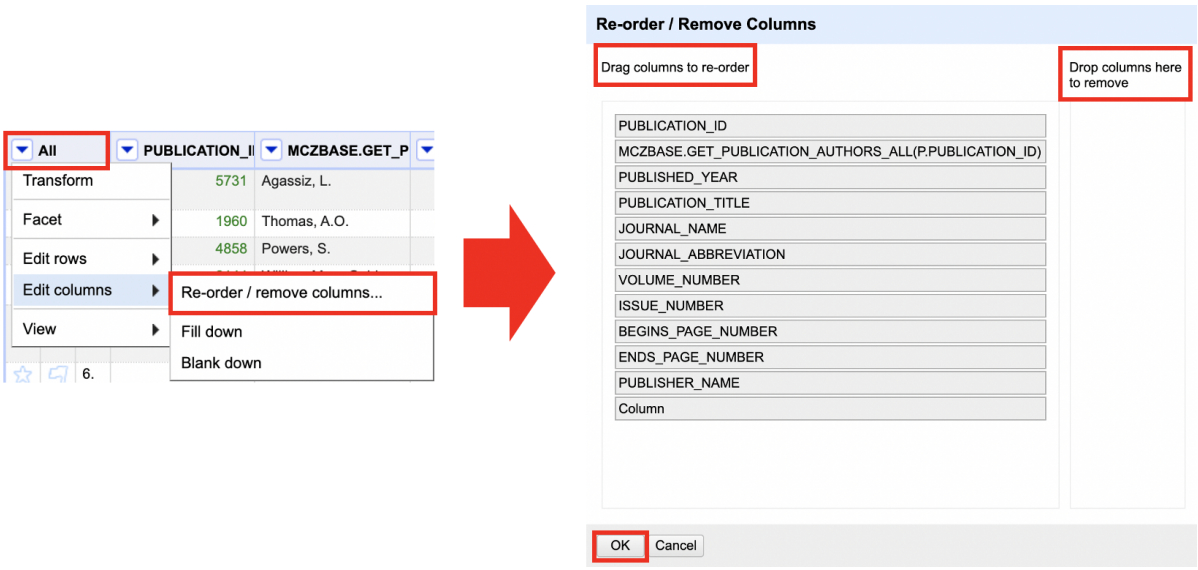
- Reordering columns Reorder your columns for better contextualization or to prepare for further transformations.

- Sorting Data Arrange your data according to defined order within one or more columns.
- Text Based Discovery Search, narrow, and isolate data based on textual information.
- Number and Date Based Discovery Search, narrow, and isolate data based on numeric and date information.

2.4.1 Reordering Columns

OpenRefine allows you to reorder your columns for better contextualization or to prepare for further transformations of your data. To reorder columns:

1. Go to the first column, labeled "All," and click the arrow button on the column header.
2. Select the option "Edit columns" and then choose "Re-order/remove columns."
3. In the pop-up window, click and drag the column titles into the order in which you would like them to be displayed or drag them to the far right to remove the columns.



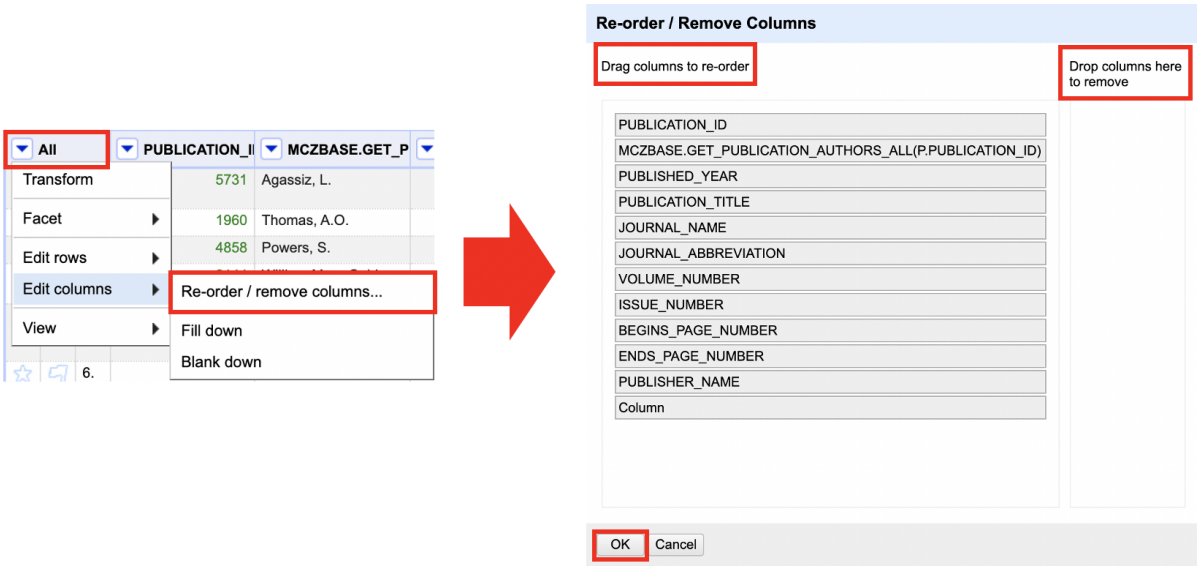
Helpful Tips:

Reordering columns can be useful when preparing to combine column data or to rearrange columns after splitting column data

2.4.2 Sorting Data

OpenRefine has a "Sort" feature that works similarly to Excel's sorting. This function allows you to arrange your data in one or more columns in a certain order. #### Sorting by One Column 1.

1. Go to the column you would like to sort by and click the arrow button on the column header.
2. Select the "Sort" option.
3. In the pop-up window, select the criteria and order you wish to sort by and select "Ok."
4. The order of rows/records will change in the grid view according to your selections

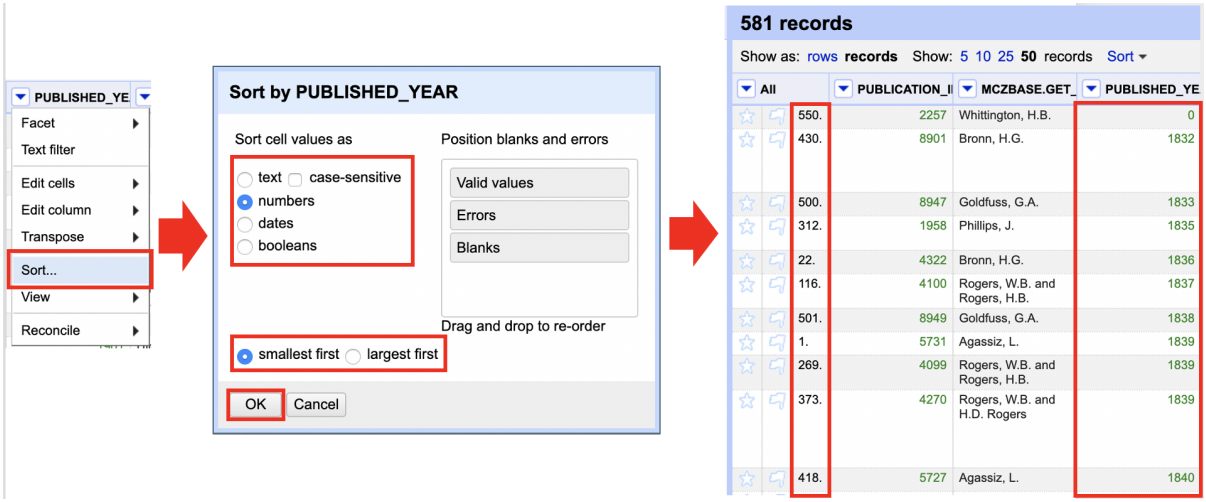


Helpful Tips:

- The "Sort" function does not permanently change the order of your rows/records, only the order you view them in. This can be checked by looking at the row/record numbers at the left of the grid view.
- The order of your rows/records can be permanently changed by selecting "Sort" at the top of the grid view and choosing "Reorder rows permanently."
- The "Sort" function can be removed by selecting "Sort" at the top of the grid view and choosing "Remove sort."

2.4.2.1 Sorting by Multiple Columns

1. Follow steps 1-4 above for your first sort.
2. Do not remove the first sort, but choose your second column and repeat steps 1-2.
3. In the pop-up window DO NOT select the "Sort by this column alone" option and select "Ok."
4. The order of rows/columns will change in the grid view according to your first sort and then within that, according to the second sort



Helpful Tips:

- If you have applied multiple sorts, you can remove each one individually or as a group. This can be done by selecting "Sort" at the top of the grid view, hovering over the sort title you wish to remove, and selecting "Remove sort."

2.4.3 Text Based Discovery

2.4.3.1 Text Facet

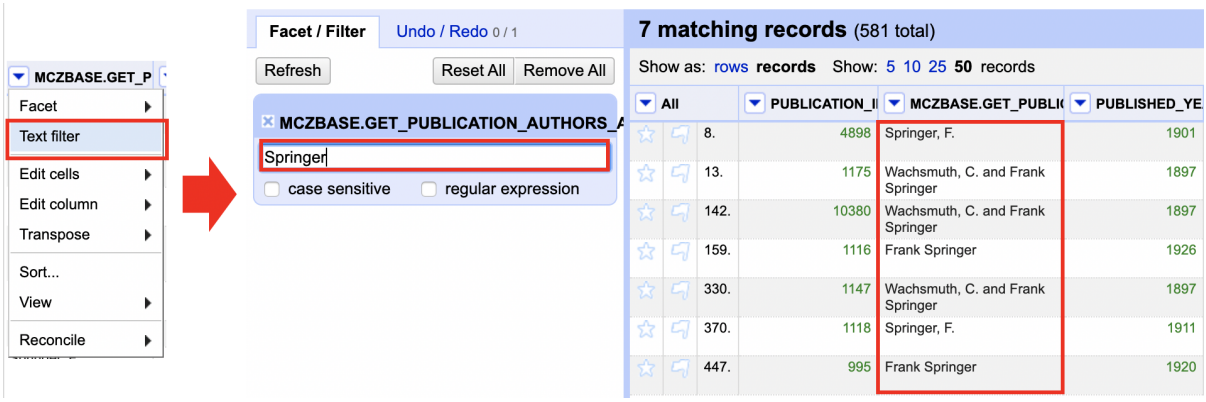
- Faceting allows you to quickly view unique values in a column, make edits to those values, and narrow your display to show results containing a specific facet.
- To display facets:

1. Go to the column you would like to analyze and click the arrow button on the column header.
2. Choose "Facet" from the drop-down menu, and then select "Text facet."
3. A facet window will appear in the pane to the left side of the grid view.

2.4.3.2 Text Filter

- The text filter option works like the "Find" function in Excel, allowing you to search a column for values containing a specific string.
- To display the text filter function:

1. Go to the column you would like to search and click the arrow button on the column header.
2. Choose "Text filter."
3. A window with a search box will appear in the pane to the left side of the grid view.



2.4.4 Number and Data Based Discovery

2.4.4.1 Changing the Cell Format

- When you import a project into `OpenRefine`, the cells will automatically be given a format: text, number, or date. To change this format:

  1. Go to the column you would like to change the format of and click the arrow button on the column header.
  2. Choose “Edit cells” and then select “Common transformations.”
  3. Under “Common transformation,” choose the desired format.

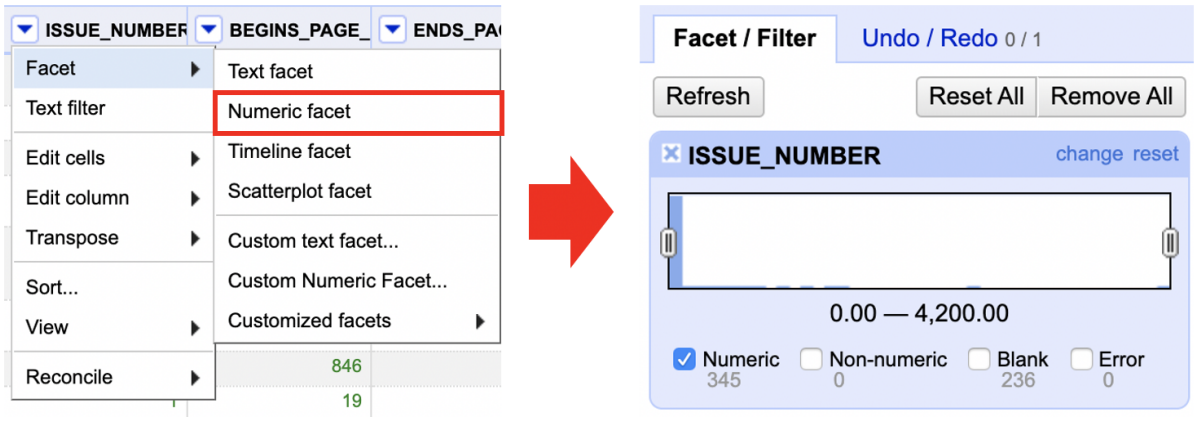
Helpful Tips:

-Columns with values in green are either in date or number format. This makes it easier to identify what types of facets and filters you can use.

2.4.4.2 Numeric Facet

- The Numeric Facet allows you to sort columns with numeric values and to use a sliding scale to adjust the range of number values displayed in the grid view. To display this facet:

1.Go to the column you would like to facet and click the arrow button on the column header. 2. Choose “Facet” and then select “Numeric facet.” 3. A window with a sliding scale will appear in the pane to the left side of the grid view.



Helpful Tips:

- The "Numeric Facet" function is useful for narrowing data to exclude outlying numeric values.
- This function can also be used to isolate high or low values.

2.4.4.3 Timeline Facet

- The Timeline Facet allows you to sort columns with date values and to use a sliding scale to adjust the range of date values to displayed in the grid view. To display this facet:

1. Go to the column you would like to facet and click the arrow button on the column header.
2. If the data is not read as "dates", it needs to covert this column to dates. See the explanation of Common transformations.
3. Choose "Facet" and then select "Timeline facet."
4. A window with a sliding scale will appear in the pane to the left side of the grid view.

Helpful Tips:

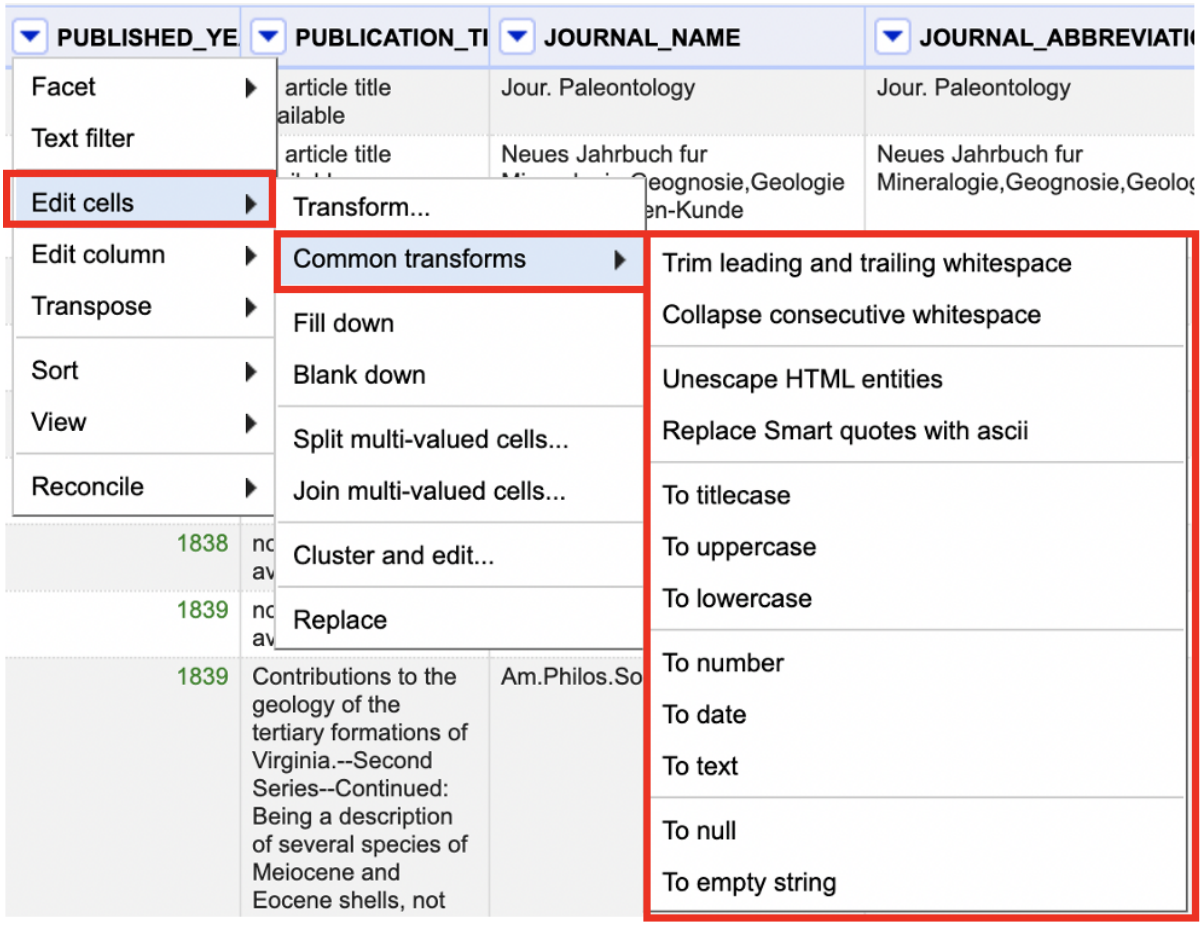
The "Timeline facet" function is useful for narrowing data to exclude outlying date values. This function can also be used to isolate specific dates, times, or date ranges.

2.5 Data Preparation and Normalization

2.5.1 Common Transformations

OpenRefine has a number of functions for performing typical data transformations. Many of these changes are for data cleansing and may be done with regular expressions as well. To locate these transformations:

1. Go to the column you would like to make edits to and click the arrow button on the column header.
2. Select the "Edit cells" and then "Common transforms" options.



2.5.2 Removing Duplicates

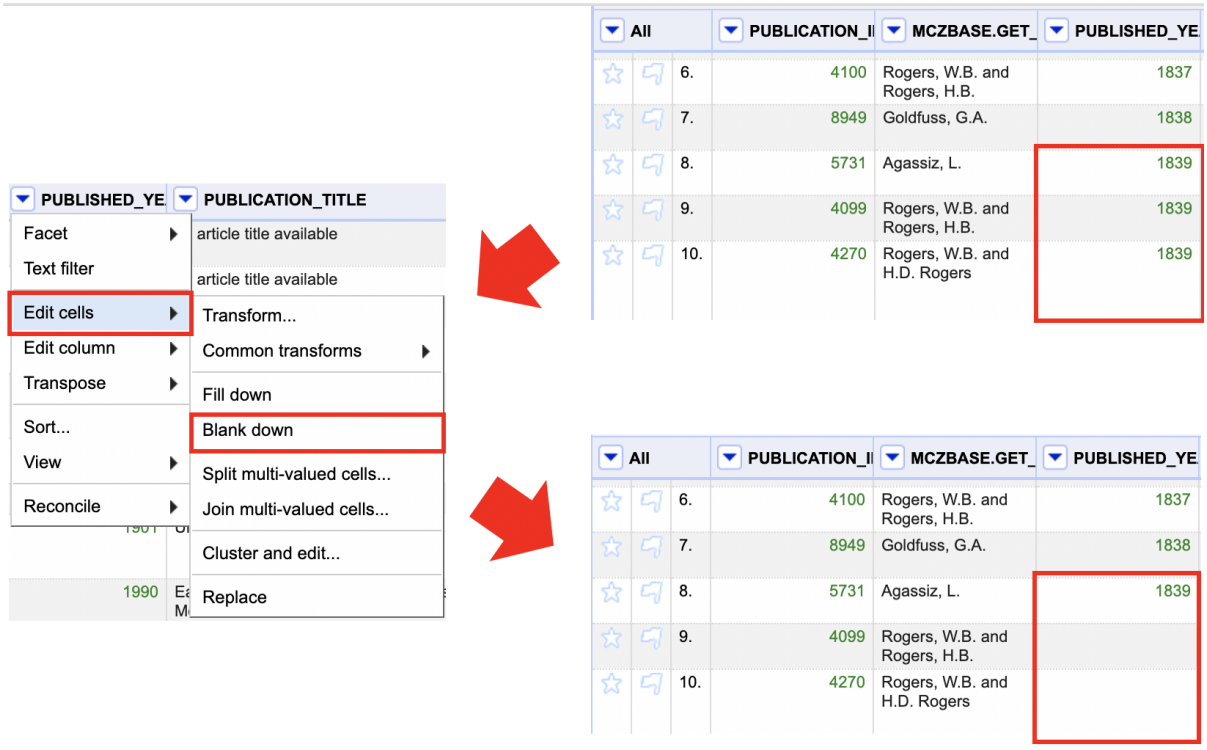
Your data may contain duplicate information that should be removed from the dataset.

NOTE: For optimal results, make sure you're in ROW mode.

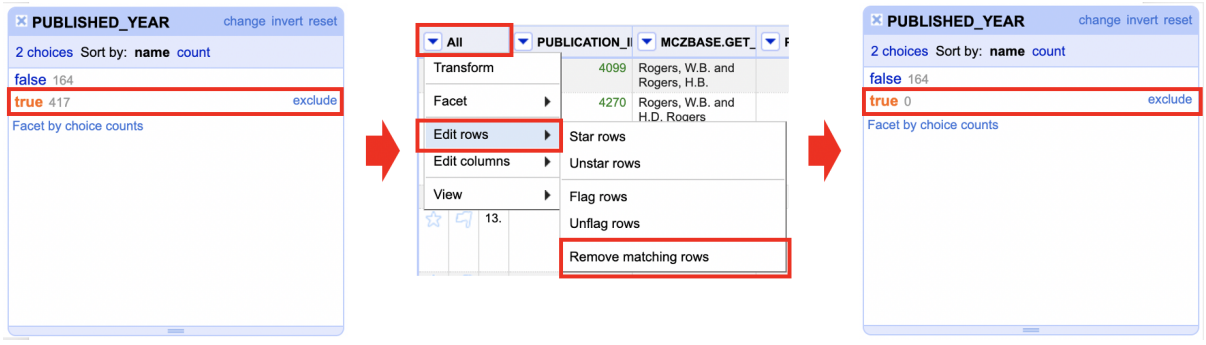
1. Sort the column with duplicates using the sort function. See the Sorting Data section for sorting instructions.
2. Choose "Sort" and then "reorder rows permanently" after you've sorted the column.
3. Select the duplicates column and click the arrow button in the column header.
4. Select "Edit cells" followed by "Blank down."

a. If two rows succeeding each other have the identical information, "Blank

down" will detect it. If they do, the cell values in the second row will be eliminated and the second row will be "blanked out."



5. After you have used the "Blank down" function, use the "Facet by blank" to identify rows with blank cell values for that column.
6. From the facet window, select the "true" option.
7. Go to the column labeled "All" and click on the arrow button, then select "Edit rows" and choose "Remove all matching rows."
8. All rows with the identified duplicates will be removed. To restore the full data view, simply reset the facets



Helpful Tips:

- It is important to make sure that all cells in the column you are applying this transformation to have values. If there are cells that were originally blank before applying the "Blank down" function, they will be deleted along with the duplicate rows.
- It is important to be aware that the entire row of values will be deleted along with the duplicate value in a given column. Be careful and check before deleting rows or applying the "Blank down" function to make sure that data meant to be kept is not accidentally deleted.

2.5.3 Splitting Cell Values

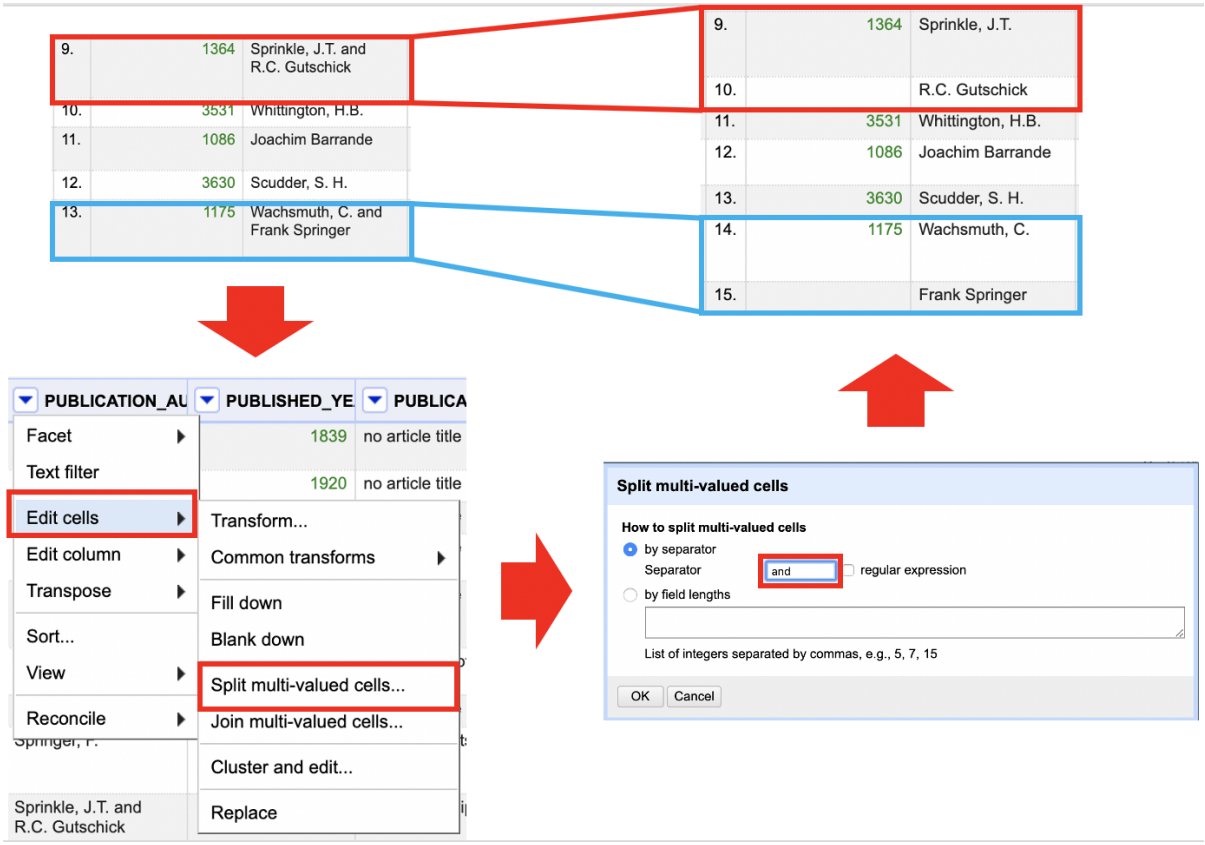
You can come across columns that have a lot of data clumped together. OpenRefine has numerous options for separating these pieces of data into more meaningful divisions, such creating new columns or creating a multi-row record.

2.5.3.1 Creating a New Row Based on a Separator

NOTE: Before starting this process, make sure to reset all facets on the column you want to work with.

1. Click the arrow button on the column header next to the column you want to split.
2. Choose "Split multi-valued cells" from the "Edit cells" menu.
3. Type the separator (usually a semicolon, comma, or other special character) in the pop-up window and click "OK."
4. Cells having separator values will be divided into different rows according to those separators.

Note: The new rows will be part of the same record as before. See the Layout section for further information on rows and records.



2.5.3.2 Creating a New Column Based on a Separator or Field Length

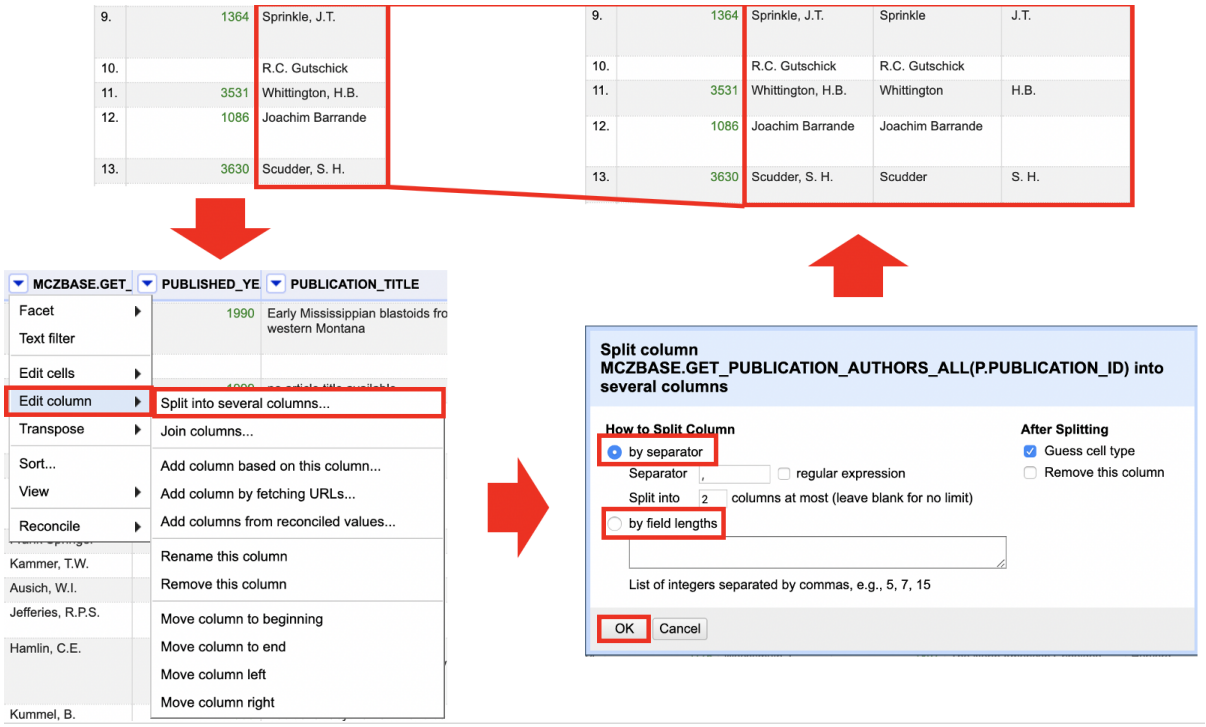
1. Go to the column you would like to split and click the arrow button on the column header.
2. Select the "Edit column" option and choose "Split into several columns."
3. In the pop-up window, select the method by which you would like to split the column:

a. "By separator" allows you to input the separator value (often a semi-colon, comma, or other special character) as well as to limit the number of additional columns are created.

b. "By field lengths" allows you to input a string of integers to indicate where in the original column value you would like OpenRefine to split the values. For example: i. A column contains dates in the YYYY/MM/DD format. You wish to divide this into three columns (Year, Month, Day). Under "by field lengths," use the list "4, 1, 2, 1, 2" indicating "year, slash, month,

slash, day." The result will be five columns, two with only slashes which you can remove, and three with the Year, Month, and Day respectively.

4. When you have selected and identified all your criteria, select "OK."



Helpful Tips:

- The new columns will be named after the original column from which they were built. You can change the column titles by going to the column header, clicking the arrow button, selecting "Edit column," and then selecting "Rename this column."
- This transformation is useful for splitting data into manageable chunks for analysis. Information such as addresses, dates, and people could be included.

2.5.4 Combining cell Values

In OpenRefine 3.3, you can merge data from different columns. You can use this technique to combine the contents of two columns, add a certain string or characters to the values of a column, or do both. There are various methods for

combining data from different columns. Introduce two fundamental functions for combining data:

- Joining columns
- Concatenation

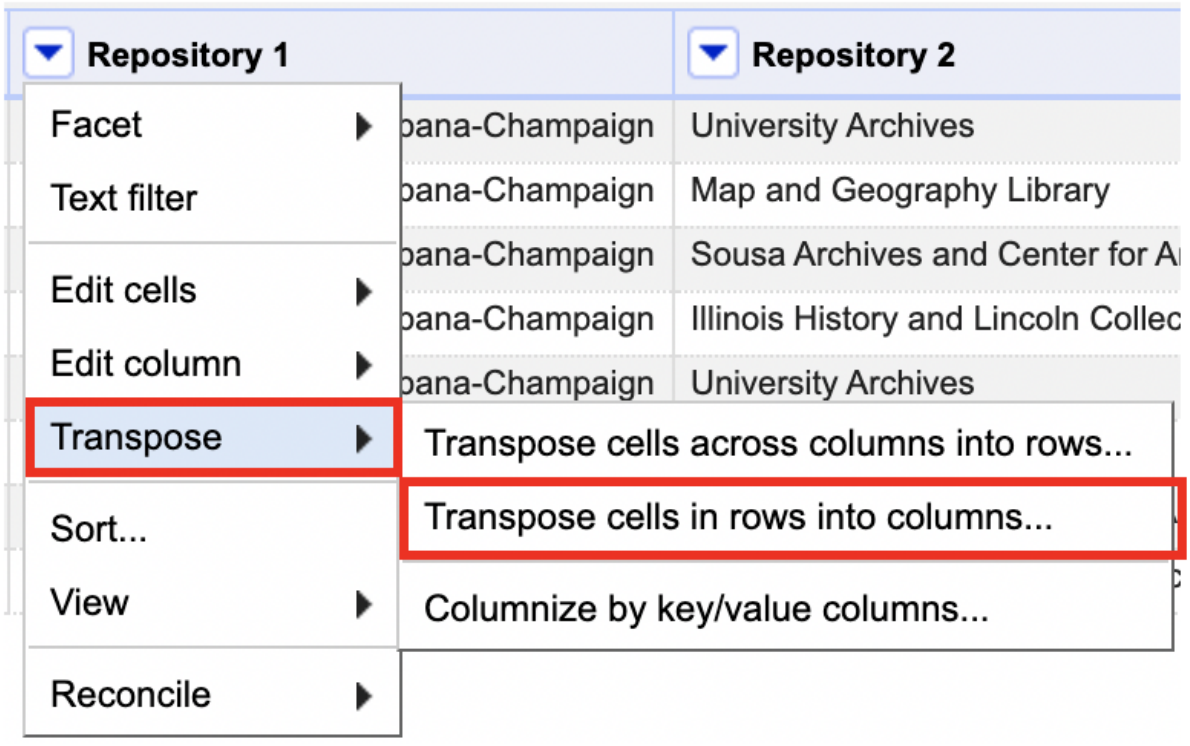
2.5.5 Data Shaping

OpenRefine has functions that allow you to reshape your data by transposing information between rows and columns in order to see it in a new way.

2.5.5.1 Rows into Columns

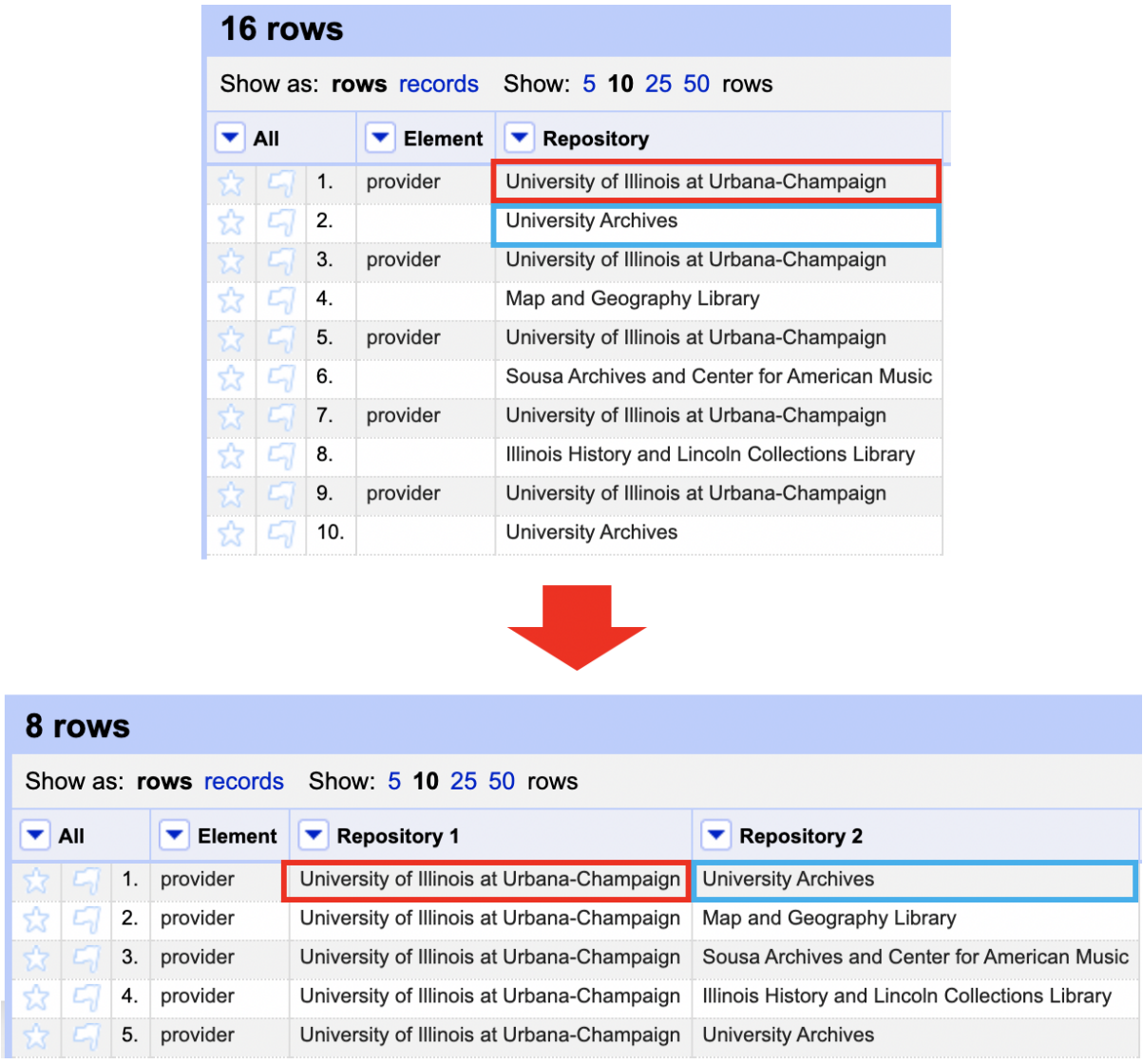
To transpose rows into columns:

1. Go to the column with the data you would like to separate into different columns.
2. Click the arrow button in the column header and select "Transpose."
3. Select "Transpose cells in rows into columns



4. In the pop-up window, enter the number of rows you would like to

transpose and click OK. NOTE: If you enter "2," OpenRefine will create two columns, pulling cell values from rows in sets of two. For example: (Row 1, Column A) and (Row 2, Column A) will become (Row 1, Column A) and (Row 1, Column B) respectively.



16 rows

Show as: **rows** records Show: 5 **10** 25 50 rows

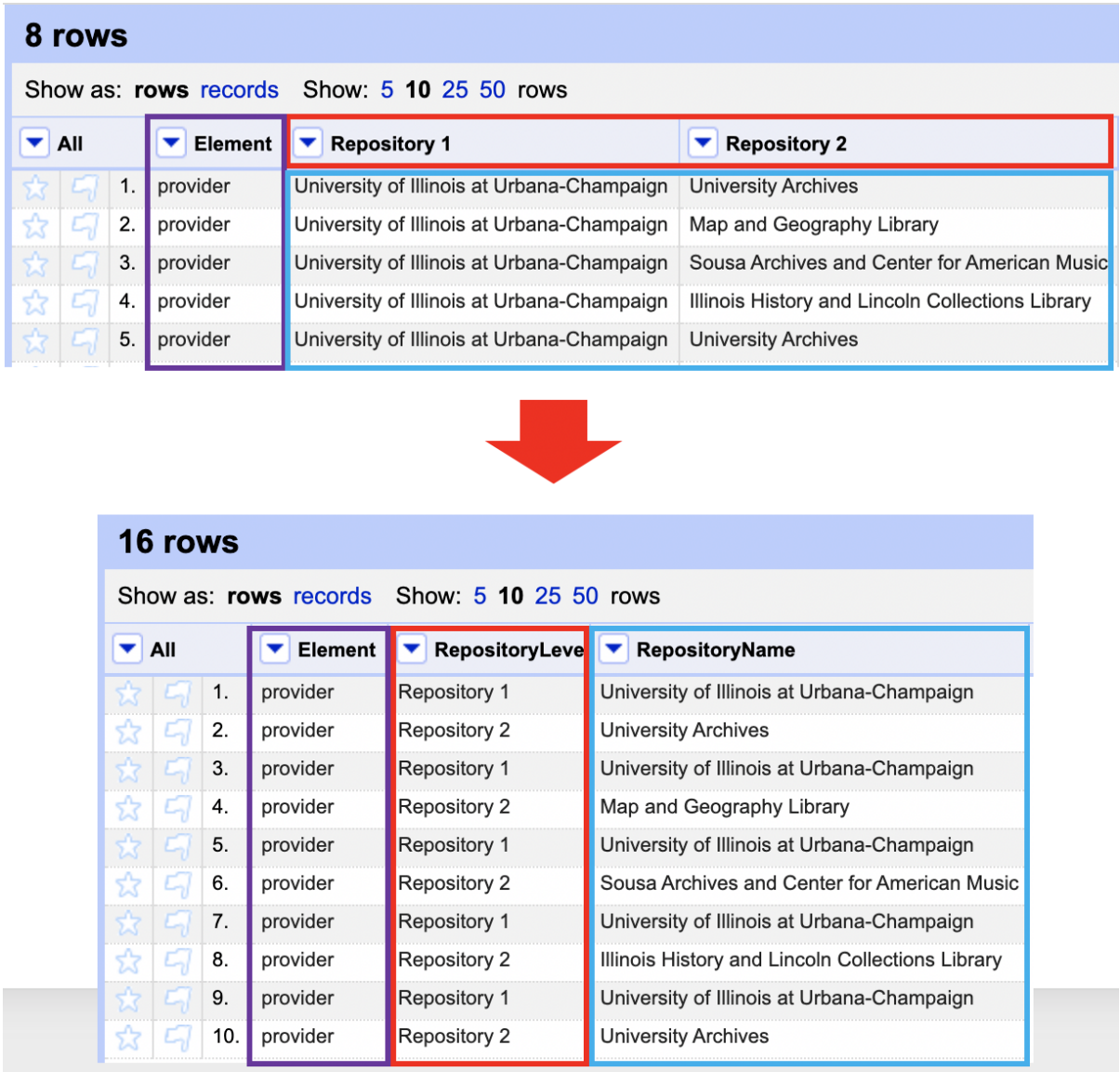
All		Element	Repository
1.	provider		University of Illinois at Urbana-Champaign
2.			University Archives
3.	provider		University of Illinois at Urbana-Champaign
4.			Map and Geography Library
5.	provider		University of Illinois at Urbana-Champaign
6.			Sousa Archives and Center for American Music
7.	provider		University of Illinois at Urbana-Champaign
8.			Illinois History and Lincoln Collections Library
9.	provider		University of Illinois at Urbana-Champaign
10.			University Archives

8 rows

Show as: **rows** records Show: 5 **10** 25 50 rows

All		Element	Repository 1	Repository 2
1.	provider		University of Illinois at Urbana-Champaign	University Archives
2.	provider		University of Illinois at Urbana-Champaign	Map and Geography Library
3.	provider		University of Illinois at Urbana-Champaign	Sousa Archives and Center for American Music
4.	provider		University of Illinois at Urbana-Champaign	Illinois History and Lincoln Collections Library
5.	provider		University of Illinois at Urbana-Champaign	University Archives

5. Click "Transpose."



8 rows

Show as: **rows** records Show: 5 **10** 25 50 rows

All	Element	Repository 1	Repository 2
1.	provider	University of Illinois at Urbana-Champaign	University Archives
2.	provider	University of Illinois at Urbana-Champaign	Map and Geography Library
3.	provider	University of Illinois at Urbana-Champaign	Sousa Archives and Center for American Music
4.	provider	University of Illinois at Urbana-Champaign	Illinois History and Lincoln Collections Library
5.	provider	University of Illinois at Urbana-Champaign	University Archives

16 rows

Show as: **rows** records Show: 5 **10** 25 50 rows

All	Element	RepositoryLeve	RepositoryName
1.	provider	Repository 1	University of Illinois at Urbana-Champaign
2.	provider	Repository 2	University Archives
3.	provider	Repository 1	University of Illinois at Urbana-Champaign
4.	provider	Repository 2	Map and Geography Library
5.	provider	Repository 1	University of Illinois at Urbana-Champaign
6.	provider	Repository 2	Sousa Archives and Center for American Music
7.	provider	Repository 1	University of Illinois at Urbana-Champaign
8.	provider	Repository 2	Illinois History and Lincoln Collections Library
9.	provider	Repository 1	University of Illinois at Urbana-Champaign
10.	provider	Repository 2	University Archives

Helpful Tips:

- If there are specific columns you would like to transpose into rows, consider moving them next to each other by following the instructions for Reordering Columns.
- Since transposing columns to rows does not preserve record associations (see Records and Rows), make sure you have a unique identifier column not being included in the transformation. This way, when you select “Fill down in other columns.” the unique identifier can be used to associate related data.
- All transformations of columns into rows are tracked in the Undo/Redo tab, so

you can experiment with transposing columns into rows until you are comfortable with the result.

2.5.6 Clustering

3

Data Analysis with R

3.1 What is R? What is RStudio?

The term `R` is used to refer to both the programming language and the software that interprets the scripts written using it.

`RStudio` is a popular tool for not just writing R scripts but also interacting with the R software. RStudio requires R to function properly, thus both must be installed on your machine.

3.2 Why learn R?

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

The learning curve may be longer than with other tools, but the outcomes of your research with R are based on a sequence of written commands rather than a series of pointing and clicking, which is a positive thing! So, if you want to redo your analysis because you gathered more data, you don't need to remember which buttons you clicked in which order to get your results; all you have to do is rerun your script.

Working using scripts clarifies the steps you took in your investigation, and the code you write can be reviewed by another person who can provide input and point out errors.

Working with scripts challenges you to think more deeply about what you're doing and makes it easier to learn and comprehend the approaches you utilize.

3.2.2 R code is great for reproducibility

When someone else (including your future self) can get the same results from the same dataset using the same analysis, it's called reproducibility.

To generate manuscripts from your code, R interfaces with different technologies. The figures and statistical tests in your publication are automatically updated if you acquire more data or correct a mistake in your dataset.

Knowing R will offer you an advantage with these requirements, as an increasing number of journals and funding agencies seek repeatable analyses.

3.2.3 R is interdisciplinary and extensible

R provides a framework that allows you to integrate statistical methodologies from various scientific disciplines to best suit the analytical framework you need to examine your data, with over 10,000+ packages that can be installed to increase its capabilities. R, for example, contains packages for image analysis, geographic information systems (GIS), time series, population genetics, and much more.

3.2.4 R works on data of all shapes and sizes

The abilities you learn with R scale quickly as your dataset grows in size. It won't make a difference to you if your dataset comprises hundreds or millions of lines.

R was created with data analysis in mind. It includes specific data structures and data types that simplify the handling of missing data and statistical elements.

R can connect to spreadsheets, databases, and a variety of other data formats, both locally and remotely.

3.2.5 R produces high-quality graphics

R's plotting capabilities are limitless, allowing you to tweak every part of your graph to best convey the message from your data.

3.2.6 R has a large and welcoming community

R is used by thousands of people every day. Many of them are prepared to help you through mailing lists and websites such as Stack Overflow, or on the RStudio community.

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

Anyone can look at the source code to learn more about R. Because of this transparency, there are less chances for errors, and you (or someone else) can report and repair faults if you find them.

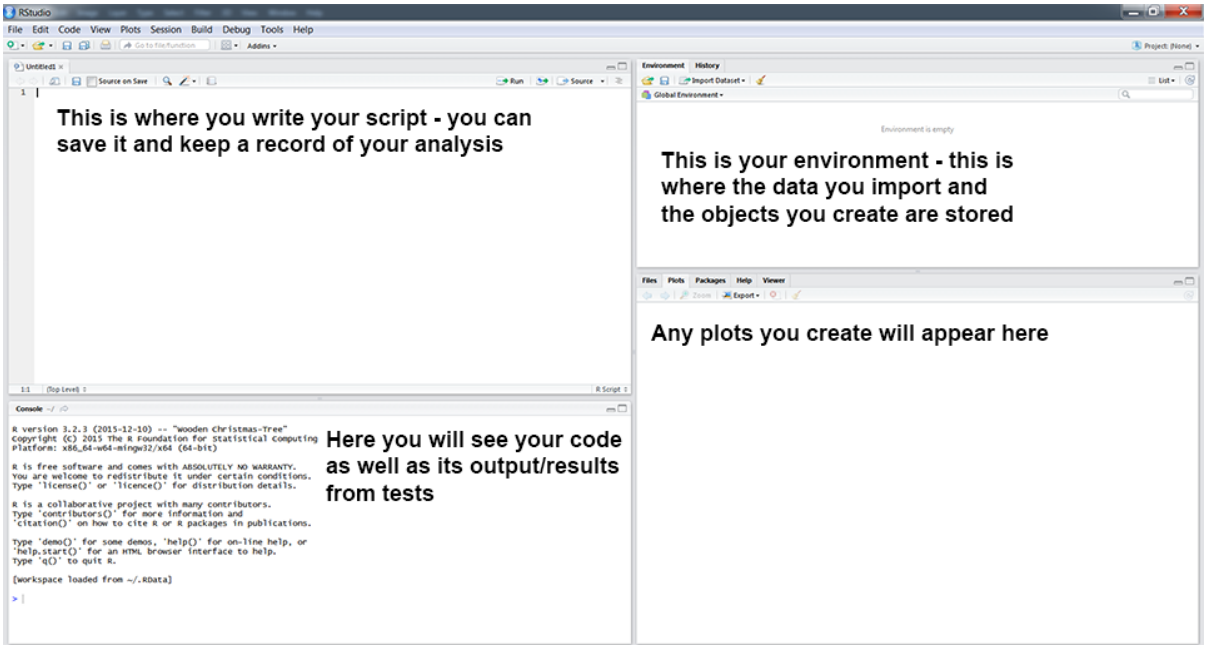


Figure 3.1 Rstudio Panels

3.2.8 Knowing your way around RStudio

Let's begin by learning about RStudio, an Integrated Development Environment (IDE) for R programming.

The Affero General Public License (AGPL) v3 covers the RStudio IDE open-source project. A commercial license and priority email assistance from RStudio, Inc. are also available for the RStudio IDE.

We'll use the RStudio IDE to write code, navigate our computer's files, analyze the variables we'll create, and visualize the plots we'll make. Other features of RStudio that we will not cover during the course include version management, package development, and writing Shiny apps.

3.3 Introduction to R

You can get output from R simply by typing math in the console:

```
5 + 6
```

```
## [1] 11
```

```
10 /3
```

```
## [1] 3.333333
```

However, we must give values to objects in order to accomplish useful and interesting things. To make an object, we must first give it a name, then use the assignment operator. `<-`, and the value we want to give it:

```
weight_kg <- 65
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of x is 3. The arrow can be read as **3 goes into x**. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing `Alt + -` (push Alt at the same time as the - key) will write `<-` in a single keystroke in a PC, while typing Option + - (push Option at the same time as the - key) does the same in a Mac.

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

```
weight_kg <- 65    # doesn't print anything
(weight_kg <- 65)  # but putting parenthesis around the call prints the value of `
```

```
## [1] 65
```

```
weight_kg          # and so does typing the name of the object
```

```
## [1] 65
```

Now that R has weight_kg in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
weight_lb <- weight_kg * 2.2
```

3.3.1 Comments

The comment character in R is `#`, and anything in a script to the right of a `#` is disregarded by *R*. Leaving notes and clarifications in your scripts is beneficial. To remark or uncomment a paragraph in RStudio, select the lines you want to comment and hit `Ctrl + Shift + C` on your keyboard at the same time. If you only want to comment out one line, place the cursor wherever on it (no need to select the entire line), then click `Ctrl + Shift + C`.

Exercise

What are the values after each statement in the following?

```
mass <- 65.5            # mass?
age  <- 55             # age?
mass <- mass * 2.0      # mass?
age  <- age - 20        # age?
mass_index <- mass/age  # mass_index?
```

3.3.2 Functions and their arguments

Functions are "canned scripts" that automate more complex sets of commands, such as operations assignments and the like. Many functions are predefined or can be accessed by installing R packages (more on that later). A function typically takes one or more arguments as input. Most (but not all) functions return a value. The function `sqrt()` is an excellent example. The input (argument) must be an integer, and the output (return value) is the square root of that number. Calling a function ('running it') is the process of executing it. A function call might look like this:

```
ten <- sqrt(weight_kg)
```

The **sqrt()** method is given the value of weight kg, which calculates the square root and returns the value, which is then assigned to the object ten. Because it just accepts one argument, this function is incredibly basic.

Let's try a function that can take multiple arguments: `round()`

```
round(3.14159)
```

```
## [1] 3
```

We used `round()` with only one argument, `3.14159`, and it returned `3`. Because the default is to round to the nearest full number, this is the case. We can see

how to gain extra digits by gathering knowledge about the round function. To find out what arguments it requires, we can use `args(round)` or search up the documentation for this method with `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

We see that if we want a different number of digits, we can type digits = 3 or however many we want.

```
round(3.14159, digits = 3)
```

```
## [1] 3.142
```

```
# or
round(3.14159, 3)
```

```
## [1] 3.142
```

3.3.3 Data Structure

R has several core data structures, and we'll take a look at each. * Vectors * Factors * Lists * Matrices/arrays * Data frames

3.3.3.1 Vectors

Vectors form the basis of R data structures. It is the most frequent and fundamental data type in R, and it is the workhorse of the language. A vector is made up of a collection of values that can be numbers or characters. The `c()` function can be used to assign a set of values to a vector. For instance, we may make a vector of animal weights and assign it to a new object called weight_g:

```
weight_g <- c(70, 80, 75, 22)
weight_g
```

```
## [1] 70 80 75 22
```

A vector can also contain characters:

```
animals <- c("rat","mouse",  "dog")
animals
```

```
## [1] "rat"   "mouse" "dog"
```

The quotation marks surrounding "mouse," "rat," and other words are crucial. Without the quotations, `R` will presume that objects named **rat, mouse, and dog** have been created. There will be an error message since these objects do not exist in R's memory.

There are numerous functions for inspecting the content of a vector. `length()` returns the number of elements in a given vector:

```
length(weight_g)
```

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

```
length(animals)
```

```
## [1] 3
```

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(animals)
```

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

```
class(weight_g)
```

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

The function **str()** provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
```

```
##  num [1:4] 70 80 75 22
```

```
str(animals)
```

```
##  chr [1:3] "rat" "mouse" "dog"
```

You can use the `c()` function to add other elements to your vector:

```
weight_g <- c(weight_g, 10) # add to the end of the vector
weight_g <- c(20, weight_g) # add to the beginning of the vector
weight_g
```

```
## [1] 20 70 80 75 22 10
```

Elements of an atomic vector are the same type. Example types include:

- character
- numeric (double)
- integer
- logical

3.3.4 Subsetting vectors

We must specify one or more indices in square brackets if we want to extract one or more values from a vector. Consider the following:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
```

```
## [1] "rat"
```

```
animals <- c("mouse", "rat", "dog", "cat")
animals[c(3,1)]
```

```
## [1] "dog"   "mouse"
```

We can also repeat the indices to create an object with more elements than the original one:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[c(3,1, 2, 1, 3, 1,4)]
```

```
## [1] "dog"   "mouse" "rat"   "mouse" "dog"   "mouse" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

3.3.5 Conditional subsetting

Another common way of subsetting is by using a logical vector. TRUE will select the element with the same index, while FALSE will not:

```
weight_g <- c(20, 10, 39, 54, 65)
weight_g[c(TRUE, FALSE, FALSE, TRUE, TRUE)]
```

```
## [1] 20 54 65
```

```
weight_g > 40
```

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

```
weight_g[weight_g > 40]
```

```
## [1] 54 65
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
weight_g[weight_g < 20 | weight_g > 60]
```

```
## [1] 10 65
```

```
weight_g[weight_g < 20 & weight_g == 60]
```

```
## numeric(0)
```

3.3.6 Missing Data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as NA.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm = TRUE` to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
```

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

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
```

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

```
min(heights)
```

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

```
max(heights, na.rm = TRUE)
```

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

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
# Extract those elements which are not missing values.
heights[!is.na(heights)]
```

```
## [1] 2 4 4 6
```

```
# Returns the object with incomplete cases removed. The returned object is an atom
na.omit(heights)
```

```
## [1] 2 4 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

```
# Extract those elements which are complete cases. The returned object is an atomi
heights[complete.cases(heights)]
```

```
## [1] 2 4 4 6
```

3.3.6.1 Factors

An important type of vector is a factor. Factors are used to represent categorical data structures. Although not exactly precise, one can think of factors as integers with labels. For example, the underlying representation of a variable for sex is `1:2` with labels "Male" and "Female". They are a special class with attributes, or metadata, that contains the information about the levels.

```
x = factor(c("Male","Female","Female","Male","Male"))
x
```

```
## [1] Male   Female Female Male   Male
## Levels: Female Male
```

```
attributes(x)
```

```
## $levels
## [1] "Female" "Male"
##
## $class
## [1] "factor"
```

The underlying representation is numeric, but it is important to remember that factors are categorical. Thus, they can't be used as numbers would be, as the following demonstrates.

```
x_num = as.numeric(x)  # convert to a numeric object
sum(x_num)
```

```
## [1] 8
```

```
Error in Summary.factor(c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 2L,  :
  'sum' not meaningful for factors
 Error in base::try(heights, silent = TRUE) : object 'heights' not found
```

3.3.6.2 Logicals

Logical scalar/vectors are those that take on one of two values: TRUE or FALSE. They are especially useful in flagging whether to run certain parts of code, and indexing certain parts of data structures (e.g. taking rows that correspond to TRUE). We'll talk about the latter usage later.

Here is a logical vector.

```
my_logic= c(TRUE, FALSE, TRUE, FALSE, TRUE, TRUE)
my_logic
```

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

Note also that logicals are also treated as binary 0:1, and so, for example, taking the mean will provide the proportion of TRUE values.

```
!my_logic
```

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

```
as.numeric(my_logic)
```

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

```
mean(my_logic)
```

```
## [1] 0.6666667
```

3.3.6.3 Numeric and Integer

The most common type of data structure you'll deal with are integer and numeric vectors.

```
ints <- -3:3   # integer sequences are easily constructed with the colon operator
class(ints)
```

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

```
x <- rnorm(5)  # 5 random values from the standard normal distribution
x
```

```
## [1] -2.4129509 -0.4187672  0.4456555 -0.7738910  1.0146025
```

```
typeof(x)
```

```
## [1] "double"
```

```
class(x)
```

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

The main difference between the two is that integers regard whole numbers only and are otherwise smaller in size in memory, but practically speaking you typically won't distinguish them for most of your data science needs.

3.3.6.4 Dates

The common data structure you'll deal with is a date variable. Typically dates require special treatment and to work as intended, but they can be stored as character strings or factors if desired. The following shows some of the base R functionality for this.

```
Sys.Date()
```

```
## [1] "2022-05-24"
```

```
x = as.Date(c(Sys.Date(), '2022-05-25', Sys.Date()+2))
x
```

```
## [1] "2022-05-24" "2022-05-25" "2022-05-26"
```

In almost every case however, a package like `lubridate` will make processing them much easier. The following shows how to strip out certain aspects of a date using it.

```
library(lubridate)
month(Sys.Date())
```

```
## [1] 5
```

```
library(lubridate)
day(Sys.Date())
```

```
## [1] 24
```

```
library(lubridate)
wday(Sys.Date())  # week start day ( 1-Monday, 7-sunday)
```

```
## [1] 3
```

```
quarter(Sys.Date())
```

```
## [1] 2
```

```
as_date('2022-01-01') + 100
```

```
## [1] "2022-04-11"
```

In general though, dates are treated as numeric variables, with consistent (but arbitrary) starting point. If you use these in analysis, you'll probably want to make zero a useful value (e.g. the starting date)

```
as.numeric(Sys.Date())
```

```
## [1] 19136
```

3.3.7 Matrices

With multiple dimensions, we are dealing with arrays. Matrices are two dimensional (2-d) arrays, and extremely commonly used for scientific computing. The vectors making up a matrix must all be of the same type. For example, all values in a matrix might be numeric, or all character strings.

3.3.7.1 Creating a matrix

Creating a matrix can be done in a variety of ways.

```
# create vectors
x <- 1:4
y <-  5:8
z <-  9:12


rbind(x, y, z)   # row bind
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y    5    6    7    8
## z    9   10   11   12
```

```
cbind(x, y, z)   # row bind
```

```
##      x y  z
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
```

```
matrix(
  c(x, y, z),
  nrow = 3,
  ncol = 4,
  byrow = TRUE
)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
matrix(
  c(x, y, z),
  nrow = 3,
  ncol = 4,
  byrow = FALSE
)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

3.3.8 Data Frames

Data frames are a very commonly used data structure, and are essentially a representation of data in a table format with rows and columns. Elements of a data frame can be different types, and this is because the data.frame class is actually just a list. As such, everything about lists applies to them. But they can also be indexed by row or column as well, just like matrices. There are other very common types of object classes associated with packages that are both a data.frame and some other type of structure (e.g. tibbles in the tidyverse).

Usually your data frame will come directly from import or manipulation of other R objects (e.g. matrices). However, you should know how to create one from scratch.

3.3.8.1 Creating a data frame

The following will create a data frame with two columns, a and b.

```
a <- c(1, 5, 2)
b <- c(3, 8, 1)
df <- data.frame(a,b)
df
```

```
##   a b
## 1 1 3
## 2 5 8
## 3 2 1
```

Much to the disdain of the tidyverse, we can add row names also

```
rownames(df) <- paste0('row',1:3)
df
```

```
##      a b
## row1 1 3
## row2 5 8
## row3 2 1
```

Everything about lists applies to data.frames, so we can add, select, and remove elements of a data frame just like lists. However we'll visit this more in depth later, and see that we'll have much more flexibility with data frames than we would lists for common data analysis and visualization.

3.4 Starting with data

We can read and load the data table from csv format:

```
library(tidyverse)
df <- read_csv("data/combined.csv")
```

This statement doesn't produce any output because, as you might recall, assignments don't display anything. If we want to check that our data has been loaded, we can see the contents of the data frame by typing its name: `df`

Wow... that was a lot of output. At least it means the data loaded properly. Let's check the top (the first 6 lines) of this data frame using the function `head()`:

```
head(df)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1         1     7    16  1977       2 NL         M                  32     NA
## 2        72     8    19  1977       2 NL         M                  31     NA
## 3       224     9    13  1977       2 NL         <NA>               NA     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
## # ... with 4 more variables: genus <chr>, species <chr>, taxa <chr>,
## #   plot_type <chr>
```

```
str(df)
```

```
## spec_tbl_df [34,786 x 13] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ record_id      : num [1:34786] 1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : num [1:34786] 7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : num [1:34786] 16 19 13 16 12 12 10 8 18 11 ...
```

```
##  $ year           : num [1:34786] 1977 1977 1977 1977 1977 ...
##  $ plot_id        : num [1:34786] 2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr [1:34786] "NL" "NL" "NL" "NL" ...
##  $ sex            : chr [1:34786] "M" "M" NA NA ...
##  $ hindfoot_length: num [1:34786] 32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : num [1:34786] NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr [1:34786] "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr [1:34786] "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr [1:34786] "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr [1:34786] "Control" "Control" "Control" "Control" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   record_id = col_double(),
##   ..   month = col_double(),
##   ..   day = col_double(),
##   ..   year = col_double(),
##   ..   plot_id = col_double(),
##   ..   species_id = col_character(),
##   ..   sex = col_character(),
##   ..   hindfoot_length = col_double(),
##   ..   weight = col_double(),
##   ..   genus = col_character(),
##   ..   species = col_character(),
##   ..   taxa = col_character(),
##   ..   plot_type = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

3.4.1 Inspecting Data Frame Objects

We already saw how the functions `head()` and `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

- Size:
  - `dim(df)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow(df)` - returns the number of rows
  - `ncol(df)` - returns the number of columns
- Content:
  - `head(df)` - shows the first 6 rows
  - `tail(df)` - shows the last 6 rows
- Names:
  - `names(df)` - returns the column names (synonym of colnames() for data.frame objects)
  - `rownames(df)` - returns the row names Summary:
  - `str(df)` - structure of the object and information about the class, length and content of each column
  - `summary(df)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides data.frame.

3.5 Indexing and subsetting data frames

Our survey data frame has rows and columns (it has 2 dimensions), if we want to extract some specific data from it, we need to specify the "coordinates" we want

from it. Row numbers come first, followed by column numbers. However, note that different ways of specifying these coordinates lead to results with different classes.

```
# first element in the first column of the data frame (as a vector)
df[1, 1]
```

```
## # A tibble: 1 x 1
##   record_id
##       <dbl>
## 1         1
```

```
# first element in the 6th column (as a vector)
df[1, 6]
```

```
## # A tibble: 1 x 1
##   species_id
##   <chr>
## 1 NL
```

```
# first column of the data frame (as a vector)
df[, 1]
```

```
## # A tibble: 34,786 x 1
##    record_id
##        <dbl>
##  1         1
##  2        72
##  3       224
```

```
##  4       266
##  5       349
##  6       363
##  7       435
##  8       506
##  9       588
## 10       661
## # ... with 34,776 more rows
```

```
# first column of the data frame (as a data.frame)
df[1]
```

```
## # A tibble: 34,786 x 1
##    record_id
##        <dbl>
##  1         1
##  2        72
##  3       224
##  4       266
##  5       349
##  6       363
##  7       435
##  8       506
##  9       588
## 10       661
## # ... with 34,776 more rows
```

```
# first three elements in the 7th column (as a vector)
df[1:3, 7]
```

```
## # A tibble: 3 x 1
##   sex
##   <chr>
## 1 M
## 2 M
## 3 <NA>
```

```
# the 3rd row of the data frame (as a data.frame)
df[3, ]
```

```
## # A tibble: 1 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1       224     9    13  1977       2 NL         <NA>               NA     NA
## # ... with 4 more variables: genus <chr>, species <chr>, taxa <chr>,
## #   plot_type <chr>
```

```
# equivalent to head_df <- head(df)
head_df <- df[1:6, ]
```

: is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance.

You can also exclude certain indices of a data frame using the "`-`" sign:

```
df[, -1]          # The whole data frame, except the first column
```

```
## # A tibble: 34,786 x 12
##    month   day  year plot_id species_id sex   hindfoot_length weight genus
##    <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl> <chr>
##  1     7    16  1977       2 NL         M                  32     NA Neotoma
##  2     8    19  1977       2 NL         M                  31     NA Neotoma
##  3     9    13  1977       2 NL         <NA>               NA     NA Neotoma
##  4    10    16  1977       2 NL         <NA>               NA     NA Neotoma
##  5    11    12  1977       2 NL         <NA>               NA     NA Neotoma
##  6    11    12  1977       2 NL         <NA>               NA     NA Neotoma
##  7    12    10  1977       2 NL         <NA>               NA     NA Neotoma
##  8     1     8  1978       2 NL         <NA>               NA     NA Neotoma
##  9     2    18  1978       2 NL         M                  NA    218 Neotoma
## 10     3    11  1978       2 NL         <NA>               NA     NA Neotoma
## # ... with 34,776 more rows, and 3 more variables: species <chr>, taxa <chr>,
## #   plot_type <chr>
```

```
df[-c(7:34786), ] # Equivalent to head(df)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1         1     7    16  1977       2 NL         M                  32     NA
## 2        72     8    19  1977       2 NL         M                  31     NA
## 3       224     9    13  1977       2 NL         <NA>               NA     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
```

```
## # ... with 4 more variables: genus <chr>, species <chr>, taxa <chr>,
## #   plot_type <chr>
```

Data frames can be subset by calling indices (as shown previously), but also by calling their column names directly:

```
df["species_id"]       # Result is a data.frame
df[, "species_id"]     # Result is a vector
df[["species_id"]]     # Result is a vector
df$species_id          # Result is a vector
```

In RStudio, you can use the autocompletion feature to get the full and correct names of the columns.

3.6 Factors

When we did `str(df)` we saw that several of the columns consist of integers. The columns `genus`, `species`, `sex`, `plot_type`, ... however, are of a special class called `factor`. Factors are very useful and actually contribute to making R particularly well suited to working with data. So we are going to spend a little time introducing them.

Factors represent categorical data. They are stored as integers associated with labels and they can be ordered or unordered. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings.

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts levels in alphabetical order. For instance, if you have a factor with 2 levels:

```
sex <- factor(c("male", "female", "female", "male"))
```

R will assign `1` to the level `"female"` and `2` to the level `"male"` (because `f` comes before `m`, even though the first element in this vector is `"male"`). You can see this by using the function `levels()` and you can find the number of levels using `nlevels()`:

```
levels(sex)
```

```
## [1] "female" "male"
```

```
nlevels(sex)
```

```
## [1] 2
```

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "medium", "high"), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the `sex` vector would be:

```
sex # current order
```

```
## [1] male   female female male
## Levels: female male
```

```
sex <- factor(sex, levels = c("male", "female"))
sex # after re-ordering
```

```
## [1] male   female female male
## Levels: male female
```

In R's memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self describing: `"female"`, `"male"` is more descriptive than `1`, `2`. Which one is "male"? You wouldn't be able to tell just from the integer data. Factors, on the other hand, have this information built in. It is particularly helpful when there are many levels (like the species names in our example dataset).

3.6.1 Converting factors

If you need to convert a factor to a character vector, you use `as.character(x)`.

```
as.character(sex)
```

```
## [1] "male"   "female" "female" "male"
```

In some cases, you may have to convert factors where the levels appear as numbers (such as concentration levels or years) to a numeric vector. For instance, in one part of your analysis the years might need to be encoded as factors (e.g., comparing average weights across years) but in another part of your analysis they may need to be stored as numeric values (e.g., doing math operations on the years). This conversion from factor to numeric is a little trickier. The `as.numeric()` function returns the index values of the factor, not its levels, so it will result in an entirely new (and unwanted in this case) set of numbers. One method to avoid this is to convert factors to characters, and then to numbers.

Another method is to use the `levels()` function. Compare:

```
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(year_fct)               # Wrong! And there is no warning...
```

```
## [1] 3 2 1 4 3
```

```
as.numeric(as.character(year_fct)) # Works...
```

```
## [1] 1990 1983 1977 1998 1990
```

```
as.numeric(levels(year_fct))[year_fct]    # The recommended way.
```

```
## [1] 1990 1983 1977 1998 1990
```

Notice that in the `levels()` approach, three important steps occur:

- We obtain all the factor levels using `levels(year_fct)`
- We convert these levels to numeric values using `as.numeric(levels(year_fct))`
- We then access these numeric values using the underlying integers of the vector `year_fct` inside the square brackets

3.6.2 Renaming factors

In addition to males and females, there are about 1700 individuals for which the sex information hasn't been recorded. Additionally, for these individuals, there is no label to indicate that the information is missing or undetermined. Let's rename this label to something more meaningful. Before doing that, we're going to pull out the data on sex and work with that data, so we're not modifying the working copy of the data frame:

```
sex <- df$sex
head(sex)
```

```
## [1] "M" "M" NA  NA  NA  NA
```

```
levels(sex)
```

```
## NULL
```

```
levels(sex)[1] <- "undetermined"
levels(sex)
```

```
## [1] "undetermined"
```

```
head(sex)
```

```
## [1] "M" "M" NA  NA  NA  NA
```

3.6.3 Using `stringsAsFactors=FALSE`

By default, when building or importing a data frame, the columns that contain characters (i.e. text) are coerced (= converted) into factors. Depending on what you want to do with the data, you may want to keep these columns as `character`. To do so, `read.csv()` and `read.table()` have an argument called `stringsAsFactors` which can be set to `FALSE`.

In most cases, it is preferable to set `stringsAsFactors = FALSE` when importing data and to convert as a factor only the columns that require this data type.

```
## Compare the difference between our data read as `factor` vs `character`.
df <- read.csv("data/combined.csv", stringsAsFactors = TRUE)
str(df)
df <- read.csv("data/combined.csv", stringsAsFactors = FALSE)
```

```
str(df)
## Convert the column "plot_type" into a factor
df$plot_type <- factor(df$plot_type)
```

3.7 Formatting Dates

One of the most common issues that new (and experienced!) R users have is converting date and time information into a variable that is appropriate and usable during analyses. As a reminder from earlier in this lesson, the best practice for dealing with date data is to ensure that each component of your date is stored as a separate variable. Using `str()`, We can confirm that our data frame has a separate column for day, month, and year, and that each contains integer values.

```
str(df)
```

We are going to use the `ymd()` function from the package **`lubridate`** (which belongs to the **`tidyverse`**; learn more here). . **`lubridate`** gets installed as part as the **`tidyverse`** installation. When you load the **`tidyverse`** (`library(tidyverse)`), the core packages (the packages used in most data analyses) get loaded. **`lubridate`** however does not belong to the core tidyverse, so you have to load it explicitly with `library(lubridate)`

Start by loading the required package:

```
library(lubridate)
```

`ymd()` takes a vector representing year, month, and day, and converts it to a `Date` vector. `Date` is a class of data recognized by R as being a date and can be

manipulated as such. The argument that the function requires is flexible, but, as a best practice, is a character vector formatted as "YYYY-MM-DD".

Let's create a date object and inspect the structure:

```
my_date <- ymd("2015-01-01")
str(my_date)
```

```
##  Date[1:1], format: "2015-01-01"
```

Now let's paste the year, month, and day separately - we get the same result:

```
# sep indicates the character to use to separate each component
my_date <- ymd(paste("2015", "1", "1", sep = "-"))
str(my_date)
```

```
##  Date[1:1], format: "2015-01-01"
```

Now we apply this function to the surveys dataset. Create a character vector from the `year`, `month`, and `day` columns of `df` using `paste()`:

```
date <- paste(df$year, df$month, df$day, sep = "-")
head(date)
```

```
## [1] "1977-7-16"  "1977-8-19"  "1977-9-13"  "1977-10-16" "1977-11-12"
## [6] "1977-11-12"
```

This character vector can be used as the argument for `ymd()`:

```
date2 <- ymd(paste(df$year, df$month, df$day, sep = "-"))
```

The resulting `Date` vector can be added to `df` as a new column called `date`:

```
df$date <- ymd(paste(df$year, df$month, df$day, sep = "-"))
str(df) # notice the new column, with 'date' as the class
```

```
## spec_tbl_df [34,786 x 14] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ record_id      : num [1:34786] 1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : num [1:34786] 7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : num [1:34786] 16 19 13 16 12 12 10 8 18 11 ...
##  $ year           : num [1:34786] 1977 1977 1977 1977 1977 ...
##  $ plot_id        : num [1:34786] 2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr [1:34786] "NL" "NL" "NL" "NL" ...
##  $ sex            : chr [1:34786] "M" "M" NA NA ...
##  $ hindfoot_length: num [1:34786] 32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : num [1:34786] NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr [1:34786] "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr [1:34786] "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr [1:34786] "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr [1:34786] "Control" "Control" "Control" "Control" ...
##  $ date           : Date[1:34786], format: "1977-07-16" "1977-08-19" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   record_id = col_double(),
##   ..   month = col_double(),
##   ..   day = col_double(),
##   ..   year = col_double(),
##   ..   plot_id = col_double(),
##   ..   species_id = col_character(),
##   ..   sex = col_character(),
##   ..   hindfoot_length = col_double(),
```

```
##   ..   weight = col_double(),
##   ..   genus = col_character(),
##   ..   species = col_character(),
##   ..   taxa = col_character(),
##   ..   plot_type = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

Let's make sure everything worked correctly. One way to inspect the new column is to use `summary()`:

```
summary(df$date)
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
## "1977-07-16" "1984-03-12" "1990-07-22" "1990-12-15" "1997-07-29" "2002-12-31"
##         NA's
##        "129"
```

Something went wrong: some dates have missing values. Let's investigate where they are coming from.

We can use the functions we saw previously to deal with missing data to identify the rows in our data frame that are failing. If we combine them with what we learned about subsetting data frames earlier, we can extract the columns "year,"month", "day" from the records that have `NA` in our new column `date`. We will also use `head()` so we don't clutter the output:

```
missing_dates <- df[is.na(df$date), c("year", "month", "day")]

head(missing_dates)
```

```
## # A tibble: 6 x 3
##    year month   day
##   <dbl> <dbl> <dbl>
## 1  2000     9    31
## 2  2000     4    31
## 3  2000     4    31
## 4  2000     4    31
## 5  2000     4    31
## 6  2000     9    31
```

3.8 manipulating and analyzing data with tidyverse

3.8.1 Data Manipulation using `dplyr` and `tidyr`

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Enter **`dplyr`**. **`dplyr`** is a package for making tabular data manipulation easier. It pairs nicely with **`tidyr`** which enables you to swiftly convert between different data formats for plotting and analysis.

Packages in R are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it. You should already have installed the **`tidyverse`** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **`tidyr`**, **`dplyr`**, **`ggplot2`**, **`tibble`**, etc.

The **`tidyverse`** package tries to address 3 common issues that arise when doing data analysis with some of the functions that come with R:

1. The results from a base R function sometimes depend on the type of data.

2. Using R expressions in a non standard way, which can be confusing for new learners.
3. Hidden arguments, having default operations that new learners are not aware of.

We have seen in our previous lesson that when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. We had to set **`stringsAsFactors`** to **`FALSE`** to avoid this hidden argument to convert our data type.

This time we will use the **`tidyverse`** package to read the data and avoid having to set **`stringsAsFactors`** to **`FALSE`**

To load the package type:

```
## load the tidyverse packages, incl. dplyr
library("tidyverse")
```

3.8.1.1 What are `dplyr` and `tidyr`?

The package **`dplyr`** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can connect to a database of many hundreds of GB, conduct queries on it directly, and pull back into R only what you need for analysis.

The package **`tidyr`** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data

sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this one about **tidyr**.

We'll read in our data using the `read_csv()` function, from the tidyverse package **readr**, instead of `read.csv()`.

```
df <- read_csv("data/combined.csv")


## inspect the data
str(df)
```

Notice that the class of the data is now `tbl_df`

This is referred to as a "tibble". Tibbles tweak some of the behaviors of the data frame objects we introduced in the previous episode. The data structure is very similar to a data frame. For our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

We're going to learn some of the most common **dplyr** functions:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarize()`: create summary statisitcs on grouped data

- `arrange()`: sort results
- `count()`: count discrete values

3.8.1.2 Selecting columns and filtering rows

To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`df`), and the subsequent arguments are the columns to keep.

```
library(dplyr)
dplyr::select(df, plot_id, species_id, weight)
```

```
## # A tibble: 34,786 x 3
##    plot_id species_id weight
##      <dbl> <chr>       <dbl>
##  1       2 NL             NA
##  2       2 NL             NA
##  3       2 NL             NA
##  4       2 NL             NA
##  5       2 NL             NA
##  6       2 NL             NA
##  7       2 NL             NA
##  8       2 NL             NA
##  9       2 NL            218
## 10       2 NL             NA
## # ... with 34,776 more rows
```

To select all columns *except* certain ones, put a "-" in front of the variable to exclude it.

```
dplyr::select(df, -record_id, -species_id)
```

```
## # A tibble: 34,786 x 11
##    month   day  year plot_id sex   hindfoot_length weight genus   species  taxa
##    <dbl> <dbl> <dbl>   <dbl> <chr>           <dbl>  <dbl> <chr>   <chr>    <chr>
##  1     7    16  1977       2 M                  32     NA Neotoma albigula Rode~
##  2     8    19  1977       2 M                  31     NA Neotoma albigula Rode~
##  3     9    13  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  4    10    16  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  5    11    12  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  6    11    12  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  7    12    10  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  8     1     8  1978       2 <NA>               NA     NA Neotoma albigula Rode~
##  9     2    18  1978       2 M                  NA    218 Neotoma albigula Rode~
## 10     3    11  1978       2 <NA>               NA     NA Neotoma albigula Rode~
## # ... with 34,776 more rows, and 1 more variable: plot_type <chr>
```

This will select all the variables in `df` except `record_id` and `species_id`.

To choose rows based on a specific criteria, use `filter()`:

```
dplyr::filter(df, year == 1995)
```

```
## # A tibble: 1,180 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     22314     6     7  1995       2 NL         M                  34     NA
##  2     22728     9    23  1995       2 NL         F                  32    165
##  3     22899    10    28  1995       2 NL         F                  32    171
##  4     23032    12     2  1995       2 NL         F                  33     NA
##  5     22003     1    11  1995       2 DM         M                  37     41
```

```
##  6     22042     2     4  1995       2 DM         F                 36     45
##  7     22044     2     4  1995       2 DM         M                 37     46
##  8     22105     3     4  1995       2 DM         F                 37     49
##  9     22109     3     4  1995       2 DM         M                 37     46
## 10     22168     4     1  1995       2 DM         M                 36     48
## # ... with 1,170 more rows, and 4 more variables: genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>
```

3.8.1.3 Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you create a temporary data frame and use that as input to the next function, like this:

```
df2 <- dplyr::filter(df, weight < 5)
df2_sml <- dplyr::select(df2, species_id, sex, weight)
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually. With multiple steps, that can be hard to keep track of.

You can also nest functions (i.e. one function inside of another), like this:

```
df_sml <- dplyr::select(filter(df, weight < 5), species_id, sex, weight)
```

This is handy, but can be difficult to read if too many functions are nested, as R evaluates the expression from the inside out (in this case, filtering, then selecting).

The last option, *pipes*, are a recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made

available via the **magrittr** package, installed automatically with **dplyr**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
df %>%
  dplyr::filter(weight < 5) %>%
  dplyr::select(species_id, sex, weight)
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

In the above code, we use the pipe to send the `df` dataset first through `filter()`

to keep rows where `weight` is less than 5, then through `select()` to keep only the `species_id`, `sex`, and `weight` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include the data frame as an argument to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word "then". For instance, in the above example, we took the data frame `df`, *then* we `filter`ed for rows with `weight < 5`, *then* we `select`ed columns `species_id`, `sex`, and `weight`. The **`dplyr`** functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex manipulations of data frames.

If we want to create a new object with this smaller version of the data, we can assign it a new name:

```
df_sml <- df %>%
  dplyr::filter(weight < 5) %>%
  dplyr::select(species_id, sex, weight)

df_sml
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
```

```
##  8 RM         M           4
##  9 RM         M           4
## 10 RM         M           4
## 11 PF         M           4
## 12 PF         F           4
## 13 RM         M           4
## 14 RM         M           4
## 15 RM         F           4
## 16 RM         M           4
## 17 RM         M           4
```

Note that the final data frame is the leftmost part of this expression.

3.8.1.3.1 Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of weight in kg:

```
df %>%
  mutate(weight_kg = weight / 1000)
```

```
## # A tibble: 34,786 x 14
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1         1     7    16  1977       2 NL         M                  32     NA
##  2        72     8    19  1977       2 NL         M                  31     NA
##  3       224     9    13  1977       2 NL         <NA>               NA     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
```

```
##  7       435    12    10  1977       2 NL         <NA>                NA     NA
##  8       506     1     8  1978       2 NL         <NA>                NA     NA
##  9       588     2    18  1978       2 NL         M                   NA    218
## 10       661     3    11  1978       2 NL         <NA>                NA     NA
## # ... with 34,776 more rows, and 5 more variables: genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>, weight_kg <dbl>
```

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
df %>%
  mutate(weight_kg = weight / 1000,
         weight_kg2 = weight_kg * 2)
```

```
## # A tibble: 34,786 x 15
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1         1     7    16  1977       2 NL         M                  32     NA
##  2        72     8    19  1977       2 NL         M                  31     NA
##  3       224     9    13  1977       2 NL         <NA>               NA     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
##  7       435    12    10  1977       2 NL         <NA>               NA     NA
##  8       506     1     8  1978       2 NL         <NA>               NA     NA
##  9       588     2    18  1978       2 NL         M                  NA    218
## 10       661     3    11  1978       2 NL         <NA>               NA     NA
## # ... with 34,776 more rows, and 6 more variables: genus <chr>, species <chr>,
## #   taxa <chr>, plot_type <chr>, weight_kg <dbl>, weight_kg2 <dbl>
```

If this runs off your screen and you just want to see the first few rows, you can

use a pipe to view the `head()` of the data. (Pipes work with non-**dplyr** functions, too, as long as the **dplyr** or `magrittr` package is loaded).

```
df %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1         1     7    16  1977       2 NL         M                  32     NA
## 2        72     8    19  1977       2 NL         M                  31     NA
## 3       224     9    13  1977       2 NL         <NA>               NA     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
## # ... with 5 more variables: genus <chr>, species <chr>, taxa <chr>,
## #   plot_type <chr>, weight_kg <dbl>
```

The first few rows of the output are full of `NA`s, so if we wanted to remove those we could insert a `filter()` in the chain:

```
df %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
```

```
## 1       588     2    18  1978       2 NL         M                   NA    218
## 2       845     5     6  1978       2 NL         M                   32    204
## 3       990     6     9  1978       2 NL         M                   NA    200
## 4      1164     8     5  1978       2 NL         M                   34    199
## 5      1261     9     4  1978       2 NL         M                   32    197
## 6      1453    11     5  1978       2 NL         M                   NA    218
## # ... with 5 more variables: genus <chr>, species <chr>, taxa <chr>,
## #   plot_type <chr>, weight_kg <dbl>
```

`is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for every row where weight *is not* an `NA`.

3.8.1.3.2 Split-apply-combine data analysis and the `summarize()` function

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **`dplyr`** makes this very easy through the use of the `group_by()` function.

3.8.1.3.2.1 The `summarize()` function

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to compute the mean `weight` by sex:

```
df %>%
  group_by(sex) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   sex   mean_weight
##   <chr>       <dbl>
```

```
## 1 F            42.2
## 2 M            43.0
## 3 <NA>         64.7
```

You may also have noticed that the output from these calls doesn't run off the screen anymore. It's one of the advantages of `tbl_df` over data frame.

You can also group by multiple columns:

```
df %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

```
## # A tibble: 92 x 3
## # Groups:   sex [3]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## # ... with 82 more rows
```

When grouping both by `sex` and `species_id`, the last few rows are for animals that escaped before their sex and body weights could be determined. You may notice that the last column does not contain `NA` but `NaN` (which refers to "Not a

Number"). To avoid this, we can remove the missing values for weight before we attempt to calculate the summary statistics on weight. Because the missing values are removed first, we can omit `na.rm = TRUE` when computing the mean:

```
df %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight))
```

```
## # A tibble: 64 x 3
## # Groups:   sex [3]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## # ... with 54 more rows
```

Here, again, the output from these calls doesn't run off the screen anymore. If you want to display more data, you can use the `print()` function at the end of your chain with the argument `n` specifying the number of rows to display:

```
df %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight)) %>%
  print(n = 15)
```

```
## # A tibble: 64 x 3
## # Groups:   sex [3]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## 11 F     PF                7.97
## 12 F     PH               30.8
## 13 F     PL               19.3
## 14 F     PM               22.1
## 15 F     PP               17.2
## # ... with 49 more rows
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum weight for each species for each sex:

```
df %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight))
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     BA                9.16          6
##  2 F     DM               41.6          10
##  3 F     DO               48.5          12
##  4 F     DS              118.           45
##  5 F     NL              154.           32
##  6 F     OL               31.1          10
##  7 F     OT               24.8           5
##  8 F     OX               21            20
##  9 F     PB               30.2          12
## 10 F     PE               22.8          11
## # ... with 54 more rows
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_weight` to put the lighter species first:

```
df %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
```

```
            min_weight = min(weight)) %>%
  arrange(min_weight)
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     PF                7.97          4
##  2 F     RM               11.1           4
##  3 M     PF                7.89          4
##  4 M     PP               17.2           4
##  5 M     RM               10.1           4
##  6 <NA>  PF                6             4
##  7 F     OT               24.8           5
##  8 F     PP               17.2           5
##  9 F     BA                9.16          6
## 10 M     BA                7.36          6
## # ... with 54 more rows
```

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing order of mean weight:

```
df %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(mean_weight))
```

```
## # A tibble: 64 x 4
```

```
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 <NA>  NL               168.          83
##  2 M     NL               166.          30
##  3 F     NL               154.          32
##  4 M     SS               130          130
##  5 <NA>  SH               130          130
##  6 M     DS               122.          12
##  7 <NA>  DS               120           78
##  8 F     DS               118.          45
##  9 F     SH                78.8         30
## 10 F     SF                69           46
## # ... with 54 more rows
```

3.8.1.3.2.2 Counting

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, **dplyr** provides `count()`. For example, if we wanted to count the number of rows of data for each sex, we would do:

```
df %>%
    count(sex)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group. In other words, `df %>% count()` is equivalent to:

```
df %>%
    group_by(sex) %>%
    summarise(count = n())
```

```
## # A tibble: 3 x 2
##   sex   count
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

For convenience, `count()` provides the `sort` argument:

```
df %>%
    count(sex, sort = TRUE)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 M     17348
## 2 F     15690
## 3 <NA>   1748
```

Previous example shows the use of `count()` to count the number of rows/observations for *one* factor (i.e., `sex`). If we wanted to count *combination of factors*, such as `sex` and `species`, we would specify the first and the second factor as the arguments of `count()`:

```
df %>%
  count(sex, species)
```

```
## # A tibble: 81 x 3
##    sex   species         n
##    <chr> <chr>       <int>
##  1 F     albigula      675
##  2 F     baileyi      1646
##  3 F     eremicus      579
##  4 F     flavus        757
##  5 F     fulvescens     57
##  6 F     fulviventer    17
##  7 F     hispidus       99
##  8 F     leucogaster   475
##  9 F     leucopus       16
## 10 F     maniculatus   382
## # ... with 71 more rows
```

With the above code, we can proceed with `arrange()` to sort the table according to a number of criteria so that we have a better comparison. For instance, we might want to arrange the table above in (i) an alphabetical order of the levels of the species and (ii) in descending order of the count:

```
df %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

```
## # A tibble: 81 x 3
##    sex   species             n
##    <chr> <chr>           <int>
```

```
##  1 F     albigula          675
##  2 M     albigula          502
##  3 <NA>  albigula           75
##  4 <NA>  audubonii          75
##  5 F     baileyi          1646
##  6 M     baileyi          1216
##  7 <NA>  baileyi            29
##  8 <NA>  bilineata         303
##  9 <NA>  brunneicapillus    50
## 10 <NA>  chlorurus          39
## # ... with 71 more rows
```

From the table above, we may learn that, for instance, there are 75 observations of the *albigula* species that are not specified for its sex (i.e. `NA`).

3.8.1.3.3 Reshaping with gather and spread

In the spreadsheet lesson, we discussed how to structure our data leading to the four rules defining a tidy dataset:

1. Each variable has its own column
2. Each observation has its own row
3. Each value must have its own cell
4. Each type of observational unit forms a table

Here we examine the fourth rule: Each type of observational unit forms a table.

In `df` , the rows of `df` contain the values of variables associated with each record (the unit), values such the weight or sex of each animal associated with each record. What if instead of comparing records, we wanted to compare the different mean weight of each species between plots? (Ignoring `plot_type` for simplicity).

We'd need to create a new table where each row (the unit) is comprised of values of variables associated with each plot. In practical terms this means the

values of the species in `genus` would become the names of column variables and the cells would contain the values of the mean weight observed on each plot.

Having created a new table, it is therefore straightforward to explore the relationship between the weight of different species within, and between, the plots. The key point here is that we are still following a tidy data structure, but we have **reshaped** the data according to the observations of interest: average species weight per plot instead of recordings per date.

The opposite transformation would be to transform column names into values of a variable.

We can do both these of transformations with two `tidyr` functions, `spread()` and `gather()`.

3.8.1.3.3.1 Spreading

`spread()` takes three principal arguments:

1. the data
2. the *key* column variable whose values will become new column names.

3. the *value* column variable whose values will fill the new column variables.

Further arguments include `fill` which, if set, fills in missing values with the value provided.

Let's use `spread()` to transform df to find the mean weight of each species in each plot over the entire survey period. We use `filter()`, `group_by()` and `summarise()` to filter our observations and variables of interest, and create a new variable for the `mean_weight`. We use the pipe as before too.

```
df_gw <- df %>%
  filter(!is.na(weight)) %>%
  group_by(genus, plot_id) %>%
  summarize(mean_weight = mean(weight))
```

```
str(df_gw)
```

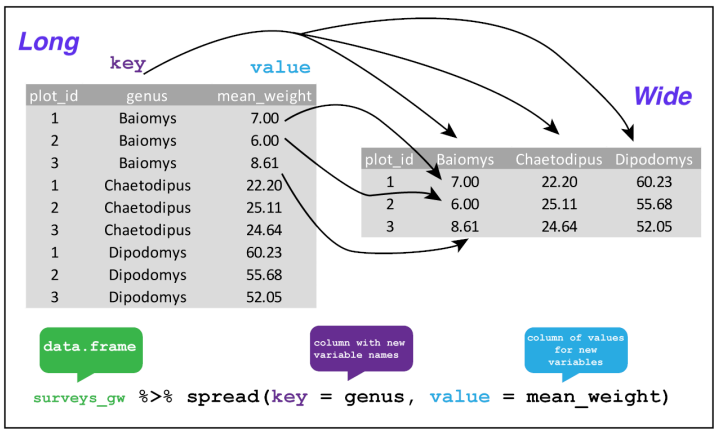
```
## grouped_df [196 x 3] (S3: grouped_df/tbl_df/tbl/data.frame)
##  $ genus      : chr [1:196] "Baiomys" "Baiomys" "Baiomys" "Baiomys" ...
##  $ plot_id    : num [1:196] 1 2 3 5 18 19 20 21 1 2 ...
##  $ mean_weight: num [1:196] 7 6 8.61 7.75 9.5 ...
##  - attr(*, "groups")= tibble [10 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ genus: chr [1:10] "Baiomys" "Chaetodipus" "Dipodomys" "Neotoma" ...
##   ..$ .rows: list<int> [1:10]
##   .. ..$ : int [1:8] 1 2 3 4 5 6 7 8
##   .. ..$ : int [1:24] 9 10 11 12 13 14 15 16 17 18 ...
##   .. ..$ : int [1:24] 33 34 35 36 37 38 39 40 41 42 ...
##   .. ..$ : int [1:24] 57 58 59 60 61 62 63 64 65 66 ...
##   .. ..$ : int [1:24] 81 82 83 84 85 86 87 88 89 90 ...
##   .. ..$ : int [1:23] 105 106 107 108 109 110 111 112 113 114 ...
##   .. ..$ : int [1:24] 128 129 130 131 132 133 134 135 136 137 ...
##   .. ..$ : int [1:24] 152 153 154 155 156 157 158 159 160 161 ...
##   .. ..$ : int [1:19] 176 177 178 179 180 181 182 183 184 185 ...
##   .. ..$ : int [1:2] 195 196
##   .. ..@ ptype: int(0)
##   ..- attr(*, ".drop")= logi TRUE
```

This yields `df_gw` where the observations for each plot are spread across multiple rows, 196 observations of 3 variables. Using `spread()` to key on `genus` with values from `mean_weight` this becomes 24 observations of 11 variables, one row for each plot. We again use pipes:

```
df_spread <-df_gw %>%
  spread(key = genus, value = mean_weight)
```

```
str(df_spread)
```

```
## tibble [24 x 11] (S3: tbl_df/tbl/data.frame)
##  $ plot_id        : num [1:24] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Baiomys        : num [1:24] 7 6 8.61 NA 7.75 ...
##  $ Chaetodipus    : num [1:24] 22.2 25.1 24.6 23 18 ...
##  $ Dipodomys      : num [1:24] 60.2 55.7 52 57.5 51.1 ...
##  $ Neotoma        : num [1:24] 156 169 158 164 190 ...
##  $ Onychomys      : num [1:24] 27.7 26.9 26 28.1 27 ...
##  $ Perognathus    : num [1:24] 9.62 6.95 7.51 7.82 8.66 ...
##  $ Peromyscus     : num [1:24] 22.2 22.3 21.4 22.6 21.2 ...
##  $ Reithrodontomys: num [1:24] 11.4 10.7 10.5 10.3 11.2 ...
##  $ Sigmodon       : num [1:24] NA 70.9 65.6 82 82.7 ...
##  $ Spermophilus   : num [1:24] NA NA NA NA NA NA NA NA NA NA ...
```



3.9 Exploratory data analysis

In his 1977 book "Exploratory Data Analysis," statistician John Tukey popularized exploratory data analysis (EDA). The overall goal of EDA is to assist us in

formulating and refining hypotheses that will lead to useful analyses or additional data collecting. EDA's main goals are as follows:

- to suggest hypotheses about the causes of observed phenomena,
- to guide the selection of appropriate statistical tools and techniques,
- to assess the assumptions on which statistical analysis will be based,
- to provide a foundation for further data collection.

EDA employs a combination of numerical and visual analytic techniques. Statistical approaches are occasionally used to enhance EDA, although their primary goal is to aid comprehension before moving on to formal statistical modeling.

3.9.1 Statistical variables and data

Statistical variables can be described in a variety of ways depending on how they might be analyzed, quantified, or presented. While it may not be the most fascinating topic, knowing what kind of variables we're dealing with is critical since it defines how we should visualize the data and, later, how we might analyze it statistically. Statisticians have given considerable effort to how variables should be classified. Although the distinctions can be subtle, we'll only look at two simple classification approaches.

3.9.1.1 Numeric vs. categorical variables

The values of **numeric variables** are numbers that describe a measurable amount, such as 'how many' or 'how much.' Quantitative variables are also known as numeric variables, and the data obtained with numeric variables is referred to as quantitative data. Numeric variables can also be classified as continuous or discrete:

- **Continuous numeric variable**: Observations can take any value between a certain set of real numbers, i.e. numbers represented with decimals. This set is typically either "every possible number".

- Discrete numeric variable: Observations can take a value based on a count from a set of whole values; e.g. 1, 2, 3, 4, 5, and so on. A discrete variable cannot take the value of a fraction between one value and the next closest value.

Categorical variables have values that describe a data unit's characteristic, such as 'what type' or 'which category.' Mutually exclusive (in one category or another) and exhaustive (containing all conceivable alternatives) categories exist for categorical variables. As a result, categorical variables are qualitative variables with a non-numeric representation. The information gathered for a category variable is qualitative. Ordinal and nominal variables are two types of categorical variables:

- **Ordinal variable**: Observations can take a value that can be logically ordered or ranked. The categories associated with ordinal variables can be ranked higher or lower than another, but do not necessarily establish a numeric difference between each category.
- **Nominal variable**: Observations can take a value that is not able to be organised in a logical sequence. Examples of nominal categorical variables include sex, business type, eye colour, religion and brand.

3.9.2 Understanding numerical variables

To explain the key concepts, we'll use `wind` and `pressure` variables using the storms data from the `nasaweather` package (remember to load and attach the package), we'll review some basic descriptive statistics and visualisations that are appropriate for numeric variables.

Atmospheric pressure and wind speed are also numerical variables. We have a little more to say. Because zero truly is zero, they are both numeric variables that are measured on a ratio scale: it makes sense to state that 20 mph is twice as fast as 10 mph, and 1000 mbar exerts twice as much pressure on objects as 500 mbar. Are these variables continuous or discrete? Consider the range of possibilities for wind speed and air pressure. Wind speeds of 40.52 mph and atmospheric pressure

of 1000.23 mbar are both completely realistic numbers, therefore these are basically continuous variables.

```
# first 100 values of atmospheric pressure
storms$pressure[1:100]
```

```
##   [1] 1005 1004 1003 1001  997  995  987  988  988  990  990  993  993  994  995
##  [16]  995  992  990  988  984  982  984  989  993  995  996  997 1000  997  990
##  [31]  992  992  993 1019 1019 1018 1017 1016 1013 1011 1009 1007 1004 1001  997
##  [46]  997  997  997  996  995  993  991  990  989 1012 1012 1012 1011 1011 1011
##  [61] 1010 1010 1006 1008 1009 1010 1009 1006 1006 1005 1004  999  999  997  991
##  [76]  995  997  995  994  994  995  996  997  997  998  998  999  999 1000 1000
##  [91] 1001 1002 1003 1005 1005 1009 1008 1008 1008 1007
```

```
# first 100 values of atmospheric pressure
storms$wind[1:100]
```

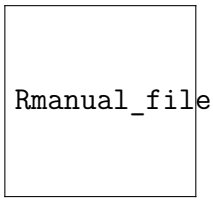
```
##   [1] 30 30 35 40 50 60 65 65 65 60 60 45 30 35 35 40 40 45 45 45 50 50 50 45 40
##  [26] 40 40 40 40 40 35 35 20 20 20 25 25 30 30 30 35 40 60 60 55 50 50 50 50
##  [51] 50 50 45 40 25 25 25 30 30 30 30 30 35 35 35 40 40 45 45 45 45 50 50 55 60
##  [76] 60 60 55 55 55 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 25 30 30 30 30
```

Notice that even though pressure is continuous variables it looks like a discrete variable because it has only been measured to the nearest whole millibar. Similarly, wind is only measured to the nearest 5 mph. These differences reflect the limitations of the methodology used to measure each variable, e.g. measuring wind speed is hard because it varies so much in space and time.

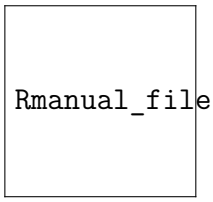
We set the properties of the geom_histogram to tweak this kind of thing—the binwidth argument adjusts the width of the bins used. Let's construct the

histogram again with 7 mbar wide bins, as well as adjust the colour scheme and axis labels a bit:

```
library(nasaweather)
ggplot(storms, aes(x = pressure)) +
  geom_histogram(binwidth = 7, fill = "steelblue", colour="darkgrey", alpha = 0.8) +
  xlab("Atmospheric Pressure (mbar)") + ylab("Count")
```



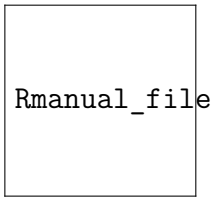

```
ggplot(storms, aes(x = wind)) +
  geom_histogram(binwidth = 10, fill = "steelblue", colour="darkgrey", alpha = 0.8) +
  xlab("Wind Speed (mph)") + ylab("Count")
```



The only things that changed in this example were the aesthetic mapping and the bin width, which we set to 10. It reveals that the wind speed during a storm tends to be about 40 mph, though the range of wind speeds is about 100 mph and the shape of the distribution is asymmetric.

We have to choose the bin widths carefully. Remember that wind speed is measured to the nearest 5 mph. This means we should choose a bin width that is a multiple of 5 to produce a meaningful histogram. Look what happens if we set the bin width to 3:

```
ggplot(storms, aes(x = wind)) +
  geom_histogram(binwidth = 3, fill = "steelblue", colour="darkgrey", alpha = 0.8) +
  xlab("Wind Speed (mph)") + ylab("Count")
```



We end up with gaps in the histogram because some intervals do not include multiples of 5. This is not a good histogram because it fails to reliably summarise the distribution. Similar problems would occur if we chose a bin width that is greater than, but not a multiple of 5, because different bins would cover a different number of values that make up the wind variable. The take home message is that we have to know our data in order to produce meaningful summaries of it.

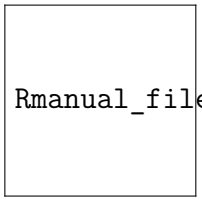
Histograms are good for visualising sample distributions when we have a reasonable sample size (at least dozens, and ideally, hundreds of observations). They aren't very effective when the sample is quite small. In this 'small data' situation it's better to use something called a dot plot

Let's use dplyr to extract a small(ish) subset of the storms data:

```
storms_small <-
  storms %>%
  filter(year == 1998, type == "Hurricane")
```

This just extracts the subset of hurricane observations from 1998. The ggplot2 code to make a dot plot with these data is very similar to the histogram case:

```
ggplot(storms_small, aes(x = pressure)) +
  geom_dotplot(binwidth = 2) +
  xlab("Atmospheric Pressure (mbar)") + ylab("Count")
```



Here, each observation in the data adds one dot, and dots that fall into the same bin are stacked up on top of one another. The resulting plot displays the same information about a sample distribution as a histogram, but it tends to be more informative when there are relatively few observations.

3.9.3 Understanding categorical variables

Exploring categorical variables is generally simpler than working with numeric variables because we have fewer options, or at least life is simpler if we only require basic summaries. We'll work with the year and type variables in storms to illustrate the key ideas.

Which kind of categorical variable is `type`? There are four storm categories in type. We can use the unique function to print these for us:

```
unique(storms$type)
```

```
## [1] "Tropical Depression" "Tropical Storm"      "Hurricane"
## [4] "Extratropical"
```

When we calculate summaries of categorical variables we are aiming to describe the sample distribution of the variable, just as with numeric variables. The general question we need to address is, 'what are the relative frequencies of different categories?' We need to understand which categories are common and which are rare. Since a categorical variable takes a finite number of possible values, the simplest thing to do is tabulate the number of occurrences of each type. We've seen how the `table` function is used to do this:

```
table(storms$type)
```

```
##
##       Extratropical           Hurricane Tropical Depression      Tropical Storm
##                 412                 896                 513                 926
```

This shows that the number of observations associated with hurricanes and tropical storms are about equal, that the number of observations associated with extratropical and tropical systems is similar, and the former pair of categories are more common than the latter. This indicates that in general, storm systems in Central America spend relatively more time in the more severe classes.

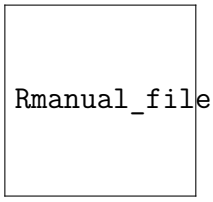
Raw frequencies give us information about the rates of occurance of different categories in a dataset. However, it's difficult to compare raw counts across different data sets if the sample sizes vary (which they usually do). This is why we often convert counts to proportions. To do this, we have to divide each count by the total count across all categories. This is easy to do in R because it's vectorised:

```
type_counts <- table(storms$type)
type_counts / sum(type_counts)
```

```
##
##       Extratropical           Hurricane Tropical Depression      Tropical Storm
##           0.1499818           0.3261740           0.1867492           0.3370950
```

So about 2/3 of observations are associated with hurricanes and tropical storms, with a roughly equal split, and the remaining 1/3 associated with less severe storms.

```
ggplot(storms, aes(x = type)) +
  geom_bar(fill = "orange", width = 0.7) +
  xlab("Storm Type") + ylab("Number of Observations")
```

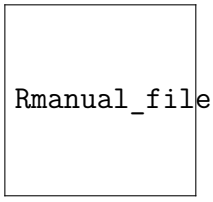


Rmanual_files/figure-latex/unnamed-chunk-55-1.pdf

This is the same summary information we produced using the table function, only now it's presented in graphical form. We can customized the bar graph with functions like `xlab` and `ylab`, and by setting various properties inside `geom_bar`.

The categories of type have quite long names, meaning the axis labels are all bunched together. One way to fix this is to make the labels smaller or rotate them via the 'themes' system. Here's an alternative solution: just flip the x and y axes to make a horizontal bar chart. We can do this with the coord_flip function (this is new):

```
ords <- c("Tropical Depression", "Extratropical", "Tropical Storm", "Hurricane")
ggplot(storms, aes(x = type)) +
  geom_bar(fill = "orange", width = 0.7) +
  scale_x_discrete(limits = ords) +
  coord_flip() +
  xlab("Storm Type") + ylab("Number of Observations")
```



Rmanual_files/figure-latex/unnamed-chunk-56-1.pdf

3.10 Regression with R

3.10.1 Linear Regression

Linear regression allows us to explore the relationship between a quantitative response variable and an explanatory variable while other variables are held constant.

In a linear model, given a vector of inputs, $X^T = (X_1, X_2, ..., X_p)$, we predict the output Y via the model:

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

The term $\hat{\beta}_0$ is the intercept, also known as the *bias* in machine learning. Often it is convenient to include the constant variable 1 in X, include β_0 in the vector of coefficients $\hat{\beta}$, and then write the linear model in vector form as an inner product,

$$\hat{Y} = X^T \hat{\beta},$$

where X^T denotes the transpose of the design matrix.

Ordinary Least Squares (OLS)

There are many different methods to fitting a linear model, but the most simple and popular method is Ordinary Least Squares (OLS). The OLS method minimizes the residual sum of squares (RSS), and leads to a closed-form expression for the estimated value of the unknown parameter β.

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - x_i^T \beta)^2$$

$RSS(\beta)$ is a quadradic function of the parameters, and hence its minimum always exists, but may not be unique. The solution is easiest to characterize in matrix notation:

$$RSS(\beta) = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(\boldsymbol{y} - \boldsymbol{X}\beta)$$

where $\boldsymbol{X}$ is an $n \times p$ matrix with each row an input vector, and $\boldsymbol{y}$ is a vector of length n representing the response in the training set. Differentiating with respect to β, we get the *normal equations*,

$$\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\beta) = 0$$

Consider predicting property values based on lot size (square feet), age (years), land value (thousands of dollars), living area (square feet), number of bedrooms and bathrooms, and whether or not the home is on the waterfront.

The basic method of performing a linear regression in R is to the use the `lm()` function.

To see the parameter estimates alone, you can just call the `lm()` function. But much more results are available if you save the results to a regression output object, which can then be accessed using the `summary()` function.

```
library(tidyverse)
library(readr)

houses <- read_csv("data/houses.csv")

houses_lm <- lm(price ~ lotSize + age + landValue +
                  livingArea + bedrooms + bathrooms +
                  waterfront,
                data = houses)
summary(houses_lm)
```

```
##
## Call:
```

```
## lm(formula = price ~ lotSize + age + landValue + livingArea +
##     bedrooms + bathrooms + waterfront, data = houses)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -220208  -35416   -5443   27570  464320
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)    1.915e+04  6.428e+03   2.980 0.002927 **
## lotSize        7.501e+03  2.075e+03   3.615 0.000309 ***
## age           -1.360e+02  5.416e+01  -2.512 0.012099 *
## landValue      9.093e-01  4.583e-02  19.841  < 2e-16 ***
## livingArea     7.518e+01  4.158e+00  18.080  < 2e-16 ***
## bedrooms      -5.767e+03  2.388e+03  -2.414 0.015863 *
## bathrooms      2.455e+04  3.332e+03   7.366 2.71e-13 ***
## waterfrontYes  1.207e+05  1.560e+04   7.738 1.70e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 59370 on 1720 degrees of freedom
## Multiple R-squared:  0.6378, Adjusted R-squared:  0.6363
## F-statistic: 432.6 on 7 and 1720 DF,  p-value: < 2.2e-16
```

3.10.1.1 Formatting regression output: tidyr

With the `tidy()` function from the broom package, you can easily create standard regression output tables.

```
library(broom)
tidy(houses_lm)
```

```
## # A tibble: 8 x 5
##   term            estimate  std.error statistic  p.value
##   <chr>              <dbl>      <dbl>     <dbl>    <dbl>
## 1 (Intercept)    19152.     6428.          2.98 2.93e- 3
## 2 lotSize         7501.     2075.          3.61 3.09e- 4
## 3 age             -136.       54.2        -2.51 1.21e- 2
## 4 landValue          0.909     0.0458     19.8  4.72e-79
## 5 livingArea        75.2       4.16       18.1  4.95e-67
## 6 bedrooms       -5767.     2388.         -2.41 1.59e- 2
## 7 bathrooms      24547.     3332.          7.37 2.71e-13
## 8 waterfrontYes 120727.    15601.          7.74 1.70e-14
```

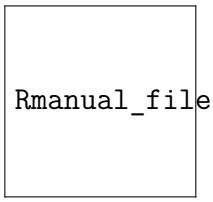
3.10.1.2 Regression Visualization

The `visreg` function takes (1) the model and (2) the variable of interest and plots the conditional relationship, controlling for the other variables. A ggplot2 graph is created with the argument $\mathbf{gg = TRUE}$.

```
# conditional plot of price vs. living area
library(ggplot2)
library(visreg)
library(readr)
houses <- read_csv("data/houses.csv")

houses_lm <- lm(price ~ lotSize + age + landValue +
                  livingArea + bedrooms + bathrooms +
                  waterfront,
                data = houses)
visreg(houses_lm, "livingArea", gg = TRUE)+
  scale_y_continuous(label = scales::dollar) +
  labs(title = "Relationship between price and location",
```

```
       subtitle = "controlling for lot size, age, land value, bedrooms and bathrooms
       caption = "source: Saratoga Housing Data (2006)",
       y = "Home Price",
       x = "Waterfront")
```



3.10.2 Generalized Linear Model(GLM)

The generalized linear model (GLM) is a variant of ordinary linear regression that allows for response variables with error distribution models other than a normal distribution, such as the Gaussian distribution.

GLMs are fit with function `glm()`. Like linear models (`lm()s`), `glm()s` have formulas and data as inputs, but also have a family input. Generalized Linear Model Syntax:

```
glm( y ~ x, data=df, family ="gaussian")
```

```
##
## Call:  glm(formula = y ~ x, family = "gaussian", data = df)
##
## Coefficients:
## (Intercept)            x
##           4            1
##
## Degrees of Freedom: 3 Total (i.e. Null);  2 Residual
## Null Deviance:       5
## Residual Deviance: 0     AIC: -Inf
```

The Gaussian family is how R refers to the normal distribution and is the default for a **glm()**. If the family is Gaussian then a GLM is the same as an LM. Non-normal errors or distributions can occur in generalized linear models.However, the possible distributions are constrained. You can, for example, use the Poisson family for count data or the binomial family for binomial data.

3.10.3 Logistic Regression

While other variables are kept constant, logistic regression can be used to investigate the relationship between a binary response variable and an explanatory variable. There are two levels to binary response variables (yes/no, lived/died, pass/fail, malignant/benign). The `visreg` package, like linear regression, may be used to illustrate these relationships.

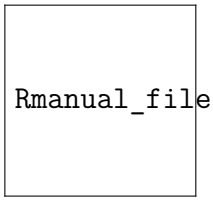
```
library(tidyverse)
library(visreg)
library(ggplot2)

marriage <- read_csv("data/marriage.csv")

# convert variable from character  to factor
marriage$sex <- factor(marriage$sex)
marriage$race <- factor(marriage$race)
marriage$sector <- factor(marriage$sector)
marriage$married <- factor(marriage$married)
#LR
married_glm <- glm(married ~ sex + age + race + sector,
                   family="binomial",
                   data=marriage)

# plot regression
```

```
visreg(married_glm, "age",
       by = "sex",
       line=list(col="red"),
       gg = TRUE,
       scale="response") +
  labs(y = "Prob(Married)",
       x = "Age",
       title = "Relationship of age and marital status",
       subtitle = "controlling for race and job sector",
       caption = "source: Current Population Survey 1985")
```



Rmanual_files/figure-latex/unnamed-chunk-59-1.pdf

3.11 Anova with R

ANOVA is one of the most often used statistical techniques in the field of ecological and environmental studies. It is also widely used in many social science areas, including as sociology, psychology, communication, and media studies.

Let's review a few key definitions in order to better understand ANOVA.

- Categorical variables have a limited number of categories or groupings, such as treatments, material type, and payment method.
- Continuous variables, such as time and height, have an endless number of values between any two values.
- An explanatory variable (also known as a **independent variable, factor,**

treatment, or predictor variable) in an experiment is a variable that is changed to examine how it impacts a response variable (also called dependent variable or outcome variable).

The explanatory variable is the CAUSE and the response variable is the EFFECT.

The analysis of variance (ANOVA) is used to determine how a continuous dependent variable (y) varies across levels of one or more categorical independent variables (x). As explained above, the latter are often referred to as "factors". Within each factor, there may be different categories called levels.

You probably guessed it by now: ANOVA = Analysis of Variance!

Part II

Part 2

4

Data Visualization with R

"The simple graph has brought more information to the data analyst's mind than any other device." — John Tukey

This chapter will show you how to use ggplot2 to visualize your data. There are various graphing systems in R, but ggplot2 is one of the most attractive and versatile. ggplot2 implements graphics language, a unified method for describing and creating graphs. By learning one system and using it in multiple locations, you can accomplish more in less time with ggplot2.

4.1 Basics of ggplot2

The tidyverse's main member, ggplot2, is the subject of this chapter. Load the tidyverse with this code to get access to the datasets, help pages, and functions:

```
library(tidyverse) # it has ggplot2 package
library(cowplot) # it allows you to save figures in .png file
```

That one line of code loads the fundamental tidyverse packages, which are used in practically every data analysis. It also shows you which tidyverse functions clash with base R functions (or from other packages you might have loaded).

If the error message "there is no package called 'tidyverse' " appears when you execute this code, you must first install it before running **library()** again.

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

A package only needs to be installed once, but it must be reloaded every time you start a new session.

4.1.1 Importing data

R can read data from a variety of sources, including text files, spreadsheets, statistical packages, and database management systems. We'll use the `mpg dataset` to demonstrate these strategies. This dataset comprises data from the US Environmental Protection Agency on 38 different car models.

4.1.1.1 Text files

The readr package provides functions for importing delimited text files into R data frames.

```
library(readr) # read dataset from file
options(readr.show_col_types = FALSE)
# CSV
mpg_csv <- read_csv("data/mpg.csv")

# TSV
mpg_tsv <- read_tsv("data/mpg.tsv")
```

These functions assume that the variable names are on the first line of data, that values are separated by commas or tabs, and that missing data is represented by blanks.

4.1.1.2 Excel spreadsheets

Data from Excel workbooks can be imported using the `readxl` program. The formats `xls` and `xlsx` are both supported.

```
library(readxl) # excel
#Excel
mpg_excel <- read_excel("data/mpg.xlsx", sheet='mpg')
head(mpg_excel)
```

```
## # A tibble: 6 x 12
##    ...1 manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
##   <dbl> <chr>        <chr> <chr> <dbl> <dbl> <chr> <chr> <dbl> <dbl> <chr> <chr>
## 1     1 audi         a4    1,8    1999     4 auto~ f        18    29 p     comp~
## 2     2 audi         a4    1,8    1999     4 manu~ f        21    29 p     comp~
## 3     3 audi         a4    2,0    2008     4 manu~ f        20    31 p     comp~
## 4     4 audi         a4    2,0    2008     4 auto~ f        21    30 p     comp~
## 5     5 audi         a4    2,8    1999     6 auto~ f        16    26 p     comp~
## 6     6 audi         a4    2,8    1999     6 manu~ f        18    26 p     comp~
```

4.1.1.3 Statistical packages

The `haven` package contains functions for importing data from many statistical programs such as SAS, SPSS and Stata.

```
library(haven) # excel
# SAS
mtcars_sas <- read_sas("data/mtcars.sas7bdat")
# SPSS
mtcars_spss <- read_sav("data/mtcars.sav")
# Stata
mtcars_stata <- read_dta("data/mtcars.dta")
head(mtcars_spss)
```

```
## # A tibble: 6 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
```

```
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## 4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
## 6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

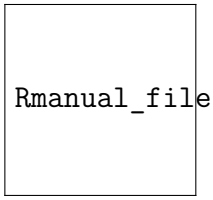
4.1.2 Scatter plot

The first function in building a graph is the ggplot function. It specifies the

- data frame containing the data to be plotted
- the mapping of the variables to visual properties of the graph. The mappings are placed within the `aes` function (where `aes` stands for aesthetics).

```
library(tidyverse)
library(ggplot2)
library(readr)


# df
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



Rmanual_files/figure-latex/unnamed-chunk-64-1.pdf

The graph demonstrates that engine size (`displ`) and fuel efficiency have a negative relationship (`hwy`). To put it another way, cars with large engines consume

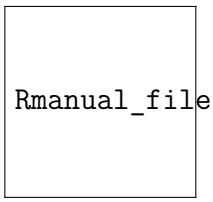
more fuel. Does this support or disprove your hypothesis concerning engine size and fuel efficiency?

The function `ggplot2` is used to start a plot in `ggplot2()`. `ggplot()` builds a coordinate system to which layers can be added. The dataset to utilize in the graph is the first parameter to `ggplot()`. So `ggplot(data = mpg_csv)` produces an empty graph, which I won't show here because it's not that interesting.

By matching the aesthetics of your plot to the variables in your dataset, you may convey information about your data. You can, for example, link the colors of your points to the class variable to reveal each car's class.

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file


# df
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
  geom_point(mapping = aes(x = displ, y = hwy, color=class))
```

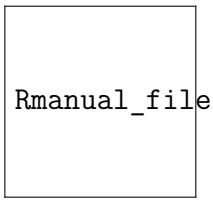


Rmanual_files/figure-latex/color_class-1.pdf

You can also set the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
# df
```

```
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
  geom_point(mapping = aes(x = displ, y = hwy), color="blue")
```



Rmanual_files/figure-latex/color_blue-1.pdf

The color modifies the appearance of the plot rather than conveying information about a variable. Set the aesthetic by name as an argument of your geom function to manually set an aesthetic; i.e. it goes outside of `aes()`. You'll need to choose a level that fits that aesthetic:

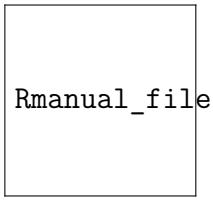
- The name of a color as a character string.
- The size of a point in mm.
- The shape of a point as a number

Facets in ggplot2:

Splitting your plot into facets, or subplots that each display one subset of the data, is another option, especially for categorical variables. Use `facet_wrap()` to facet your plot by a single variable. `facet_wrap()` expects a formula as the first input, followed by a variable name (note that "formula" is the name of a data structure in R, not a synonym for "equation"). `facet_wrap()` expects a discrete variable as a parameter.

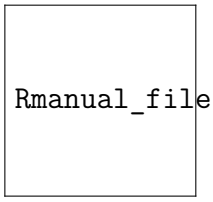
```
library(tidyverse)
library(ggplot2)
library(readr)
# df
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
```

```
  geom_point(mapping = aes(x = displ, y = hwy, color=class))+
  facet_wrap(~ manufacturer, nrow = 3)
```



Add `facet_grid()` to your plot call to facet your plot on the combination of two variables. Facet grid(first)'s parameter is likewise a formula. The formula should now have two variable names separated by a comma.

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
library(ggpubr)
# df
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
  geom_point(mapping = aes(x = displ, y = hwy, color=class))+
  facet_grid( drv ~ cyl)
```



Note: Use a `.` instead of a variable name if you don't want to facet in the rows or columns dimension, e.g. `+ facet grid(.  ~ cyl)`.

```
library(tidyverse)
library(ggplot2)
```

```
library(readr) # read dataset from file
# df
df <- read_csv("data/mpg.csv")
ggplot(data = df) +
  geom_point(mapping = aes(x = displ, y = hwy, color=class))+
  facet_grid(. ~ cyl)
```

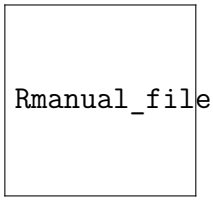
Geometric objects:_

A geom is a geometrical object used to represent data in a plot. Plots are frequently described by the sort of geom they employ. Bar charts, for example, use bar geoms, line charts use `line geoms`, and boxplots use `boxplot geoms`. *Scatterplots* use the `point geom` to break the trend. As we've seen, different ___geoms_ can be used to plot the same data. The point geom is used for scatter plot, and the smooth geom, a smooth line suited to the data.To display multiple geoms in the same plot, add multiple geom functions to `ggplot()` as shown blow:

```
library(tidyverse)
library(readr) # read dataset from file
library(ggpubr)
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()+
  labs(title = "Relationship between displacment and highway of 38 different car mod
       subtitle = "Feul economy data from 1999 to 2008",
```

```
       caption = "source: https://fueleconomy.gov/",
       x = "Engine displacement in liters",
       y = "Highway mileage",
       color = "vehicle class")+
  theme_pubr()+
  grids(linetype = "dashed")+
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 45, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```



Rmanual_files/figure-latex/geom_smooth-1.pdf

Exercise

```
# add facet wrap using manufacturer variable
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(method = 'loess')+
  labs(title = "Relationship between displacment and highway of 38 different car mod
       subtitle = "Feul economy data from 1999 to 2008",
       caption = "source: https://fueleconomy.gov/",
```

```
      x = "Engine displacement in liters",
       y = "Highway mileage",
       color = "vehicle class")+
   theme_pubr()+
  grids(linetype = "dashed")+
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 45, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(method = 'loess')+
  facet_wrap(~manufacturer)+
  labs(title = "Relationship between displacment and highway of 38 different car mod
       subtitle = "Feul economy data from 1999 to 2008",
       caption = "source: https://fueleconomy.gov/",
        x = "Engine displacement in liters",
       y = "Highway mileage",
       color = "vehicle class")+
  theme_pubr()+
  grids(linetype = "dashed")+
```

```
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 45, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```

4.1.3 Bar Plot

Stacked, clustered, or segmented bar charts are commonly used to show the relationship between two categorical variables.

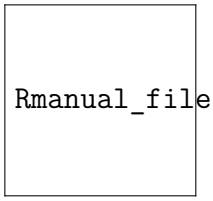
4.1.3.1 Stacked bar chart

For the autos in the Fuel economy dataset, plot the association between automotive class and drive type (front-wheel, rear-wheel, or 4-wheel drive).

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = class, fill=drv)) +
  geom_bar(position = "stack") +
  labs(title = "Vehicle class frequency by drive type for 38 different automobile mo
       subtitle = "Feul economy data from 1999 to 2008",
       caption = "source: https://fueleconomy.gov/",
       x = "vehicle class",
       y = "Count")+
  theme_pubclean()+
  grids(linetype = "dashed")+
  guides(fill=guide_legend("Drive Type"))+
```

```
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 45, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```

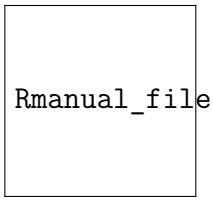


4.1.3.2 Grouped bar chart

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = class, fill=drv)) +
  geom_bar(position = "dodge") +
  labs(title = "Vehicle class frequency by drive type for 38 different automobile mo
       subtitle = "Feul economy data from 1999 to 2008",
       caption = "source: https://fueleconomy.gov/",
       x = "vehicle class",
       y = "Count")+
   theme_pubclean()+
  grids(linetype = "dashed")+
  guides(fill=guide_legend("Drive Type"))+
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 45, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
```

```
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```

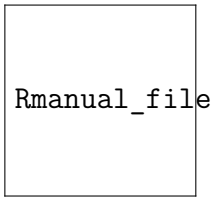


Notice that front-wheel drive is standard on all Minivans. Zero count bars are removed by default, and the remaining bars are made broader. This may not be the desired behavior. The `position = position dodge(preserve = "single")` option can be used to change this.

```
library(tidyverse)
library(ggplot2)
library(readr) # read dataset from file
library(ggthemes)
library(ggpubr)
# df
df <- read_csv("data/mpg.csv")

ggplot(data = df, mapping = aes(x = class, fill=drv)) +
  geom_bar(position = position_dodge(preserve = "single")) +
  labs(title = "Vehicle class frequency by drive type for 38 different automobile mo
       subtitle = "Feul economy data from 1999 to 2008",
       caption = "source: https://fueleconomy.gov/",
       x = "vehicle class",
       y = "Count",
       fill="Drive type")+
  theme_pubclean()+
  grids(linetype = "dashed")+
```

```
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 90, hjust =
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```

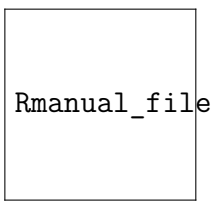


4.2 Boxplots

A boxplot displays the 25^{th} percentile, median, and 75^{th} percentile of a distribution. The whiskers (vertical lines) capture roughly 99% of a normal distribution, and observations outside this range are plotted as points representing outliers (see the figure below).

We can use boxplots to visualize the distribution of weight within each species:

```
library(tidyverse)
#read the visual treatment for patients dataset
df_vt <- read_csv("./data/visual_treatment.csv")
ggplot(data = df_vt,
       mapping = aes(x = Day, y = Value, color=Day)) +
    geom_boxplot(alpha=0.5)+
  ggtitle('Visual improvement after treatment')
```



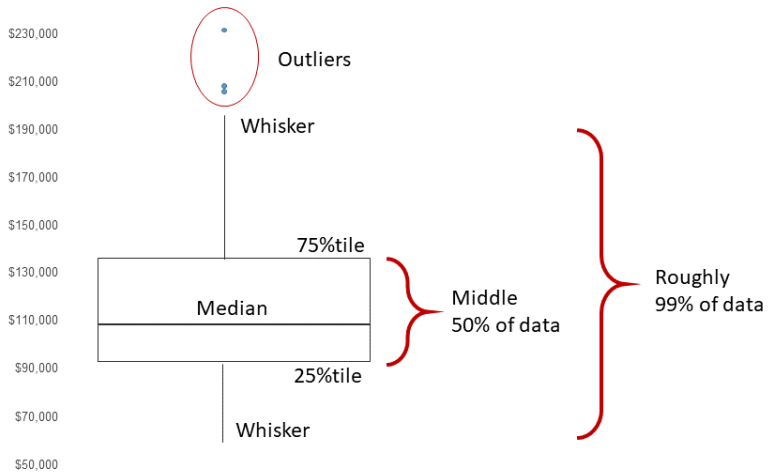


Figure 4.1 Boxplot schematics

- R appears to be unaware that the order should be **One -> Two -> Three**. Because the column Day is shown as **chr>**, which indicates character, R chose to follow the alphabetical order.
- When you want to display characters in a non-alphabetical order, factors are crucial. Factors are variables with a small number of possible values.
- Individual data can sometimes be plotted on top of the boxplot. This can be accomplished in a variety of ways.
- Using *geom jitter()*, here is a simple solution. For more information, type ?geom_jitter.

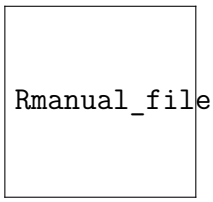
```
library(tidyverse)
library(ggthemes)
library(ggpubr)

#read the visual treatment for patients dataset
```

```
df_vt <- read_csv("./data/visual_treatment.csv")
# ordering the treatment
df_vt$Day <- factor(df_vt$Day, levels=c("One","Two","Three"))



ggplot(data = df_vt,
       mapping = aes(x = Day, y = Value, color=Day)) +
    geom_boxplot(alpha=0.5)+
  geom_jitter(width=0.15)+ # show individual data
  theme_pubclean()+ # change theme
  grids(linetype = "dashed")+ # modify grids type
  ggtitle('Visual improvement after treatment')+ # add title
  theme(axis.text.x = element_text(colour = "grey20", size = 8, angle = 0, hjust = 0
        axis.text.y = element_text(colour = "grey20", size = 12),
        strip.text = element_text(face = "italic"),
        text = element_text(size = 12))
```



Rmanual_files/figure-latex/unnamed-chunk-66-1.pdf

4.3 Plotting time series data

Let's calculate number of counts per year for each genus. First we need to group the data and count records within each group:

4.4 Line Plot

```
data(country.map, package = "choroplethrMaps")
head(unique(country.map$region), 30)
```

```
##  [1] "afghanistan" "angola"      "azerbaijan"  "moldova"     "madagascar"
##  [6] "mexico"      "macedonia"   "mali"        "myanmar"     "montenegro"
## [11] "mongolia"    "mozambique"  "mauritania"  "burundi"     "malawi"
## [16] "malaysia"    "namibia"     "france"      "niger"       "nigeria"
## [21] "nicaragua"   "netherlands" "norway"      "nepal"       "belgium"
## [26] "new zealand" "oman"        "pakistan"    "panama"      "peru"
```

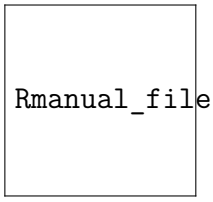
```
library(readr)
library(tidyverse)

gap <- read_csv("data/gapminder.csv")

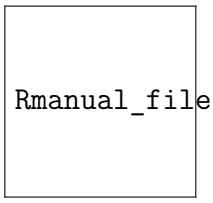
# Select US cases
library(dplyr)
plotdata <- filter(gap,
                   country == "Ethiopia")

# simple line plot
ggplot(plotdata,
       aes(x = year,
           y = lifeExp)) +
  geom_line()+
```

```
  theme_pubr()+
  grids(linetype = "dashed")
```




```
library(ggplot2)
library(ggthemes)
gap <- read_csv("data/gapminder.csv")
# line plot with points
# and improved labeling
ggplot(plotdata,
       aes(x = year,
           y = lifeExp)) +
  geom_line(size = 1.5,
            color = "lightgrey") +
  geom_point(size = 3,
             color = "steelblue") +
  labs(y = "Life Expectancy (years)",
       x = "Year",
       title = "Life expectancy changes over time",
       subtitle = "Ethiopia (1952-2007)",
       caption = "Source: http://www.gapminder.org/data/")+
  theme_pubclean()+
  grids(linetype = "dashed")
```

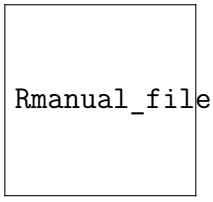



```
# prepare dataset
data(gapminder, package = "gapminder")
plotdata <- gapminder %>%
  filter(year == 2007) %>%
  rename(region = country,
         value = lifeExp) %>%
  mutate(region = tolower(region)) %>%
  mutate(region = recode(region,
                         "united states"    = "united states of america",
                         "congo, dem. rep." = "democratic republic of the congo",
                         "congo, rep."      = "republic of congo",
                         "korea, dem. rep." = "south korea",
                         "korea. rep."      = "north korea",
                         "tanzania"         = "united republic of tanzania",
                         "serbia"           = "republic of serbia",
                         "slovak republic"  = "slovakia",
                         "ethiopia"         = "democratic republic of ethiopia",
                         "yemen, rep."      = "yemen"))
```

Now lets create the map

4.5 Histograpm

```
Marriage <- read_csv("data/marriage.csv")
# plot the histogram with a binwidth of 5
ggplot(Marriage, aes(x = age)) +
  geom_histogram(fill = "cornflowerblue",
                 color = "white",
                 binwidth = 5) +
  labs(title="Participants by age",
       subtitle = "binwidth = 5 years",
       x = "Age")
```



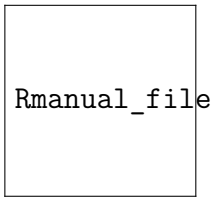
Rmanual_files/figure-latex/unnamed-chunk-71-1.pdf

```
data(gapminder, package="gapminder")

# subset Asian countries in 2007
library(dplyr)
plotdata <- gapminder %>%
  filter(continent == "Africa" &
         year == 2007)

# Sorted Cleveland plot
ggplot(plotdata,
       aes(x=lifeExp,
```

```
          y=reorder(country, lifeExp))) +
  geom_point(color="blue",
             size = 2) +
  geom_segment(aes(x = 40,
                   xend = lifeExp,
                   y = reorder(country, lifeExp),
                   yend = reorder(country, lifeExp)),
                   color = "lightgrey") +
  labs (x = "Life Expectancy (years)",
        y = "",
        title = "Life Expectancy by Country",
        subtitle = "GapMinder data for Africa - 2007") +
  theme_minimal() +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank())
```



Rmanual_files/figure-latex/unnamed-chunk-72-1.pdf

- Reunion clearly has the highest life expectancy, while Afghanistan has the lowest by far. This last plot is also called a lollipop graph (you can see why).

4.6 Map

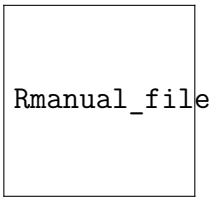
First, let us start with creating a base map of the world using `ggplot2`. This base map will then be extended with different map elements, as well as zoomed in to an area of interest. We can check that the world map was properly retrieved and converted into an sf object, and plot it with `ggplot2`:

```
library(sf)
library(tidyverse)
library(rnaturalearth)
library(rnaturalearthdata)

world <- ne_countries(scale = "medium", returnclass = "sf")
class(world)
```

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

```
ggplot(data = world) +
    geom_sf()
```



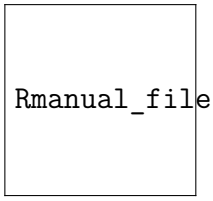
A title and a subtitle can be added to the map using the function ggtitle, passing any valid character string (e.g. with quotation marks) as arguments. Axis names are absent by default on a map, but can be changed to something more suitable (e.g. “Longitude” and “Latitude”), depending on the map:

```
library(sf)
library(tidyverse)
library(rnaturalearth)
library(rnaturalearthdata)

world <- ne_countries(scale = "medium", returnclass = "sf")
class(world)
```

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

```
ggplot(data = world) +
    geom_sf()+
    ggtitle("World map", subtitle = paste0("(", length(unique(world$NAME)), " countr
```



4.6.1 Map color

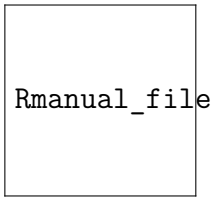
In many ways, `sf` geometries are no different than regular geometries, and can be displayed with the same level of control on their attributes. Here is an example with the polygons of the countries filled with a green color (argument fill), using black for the outline of the countries (argument color):

```
library(sf)
library(tidyverse)
library(rnaturalearth)
library(rnaturalearthdata)

world <- ne_countries(scale = "medium", returnclass = "sf")
class(world)
```

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

```
ggplot(data = world) +
    geom_sf(color = "black", fill = "lightgreen")
```



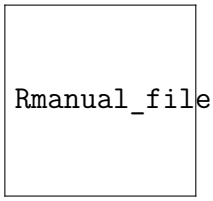
The package ggplot2 allows the use of more complex color schemes, such as a gradient on one variable of the data. Here is another example that shows the population of each country. In this example, we use the "viridis" colorblind-friendly palette for the color gradient (with option = "plasma" for the plasma variant), using the square root of the population (which is stored in the variable POP_EST of the world object):

```
library(sf)
library(tidyverse)
library(rnaturalearth)
library(rnaturalearthdata)

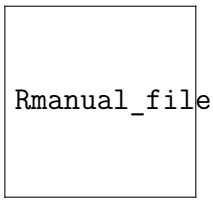
world <- ne_countries(scale = "medium", returnclass = "sf")
class(world)
```

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

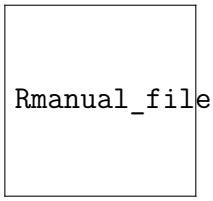
```
ggplot(data = world) +
    geom_sf(aes(fill = pop_est)) +
    scale_fill_viridis_c(option = "plasma", trans = "sqrt")
```




```
ggplot(data = world) +
    geom_sf() +
    coord_sf(xlim = c(22, 50), ylim = c(-5, 21), expand = FALSE)
```




```
library("ggspatial")
ggplot(data = world) +
    geom_sf() +
    annotation_scale(location = "br", width_hint = 0.5) +
    annotation_north_arrow(location = "bl", which_north = "true",
        pad_x = unit(3.75, "in"), pad_y = unit(0.5, "in"),
        style = north_arrow_fancy_orienteering) +
    coord_sf(xlim = c(22, 50), ylim = c(-5, 21))
```



It can be difficult to work with maps. The sf package includes ggplot2-compatible utilities like `geom_sf_()`. High-quality mapping coordinates are provided by the `rnaturalearth` package.

```
library(sf)          # for mapping geoms
library(rnaturalearth) # for map data

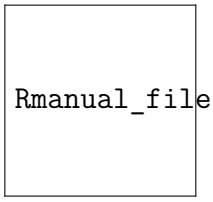
# get and bind country data
eth_sf <- ne_states(country = "Ethiopia", returnclass = "sf")
eri_sf <- ne_states(country = "Eritrea", returnclass = "sf")
dji_sf <- ne_states(country = "Djibouti", returnclass = "sf")
kenya_sf <- ne_states(country = "Kenya", returnclass = "sf")
#ssudan_sf <- ne_states(country = "South Sudan", returnclass = "sf")
somal_sf <- ne_states(country = "Somalia" , returnclass = "sf")
somald_sf <- ne_states(country = "Somaliland" , returnclass = "sf")
sudan_sf <- ne_states(country = "Sudan", returnclass = "sf")
lands <- bind_rows(eth_sf, eri_sf, dji_sf, kenya_sf,
                   sudan_sf,somal_sf,somald_sf) %>%
  filter(!is.na(geonunit))

#df <- ne_countries(continent = "Africa", returnclass = "sf")
#df$admin



# set colours
country_colours <- c("Eritrea" = "#0962BA",
                     "Djibouti" = "#00AC48",
                     "Ethiopia" = "#FF0000",
                     "Somalia" = "#FFCD2C",
                     "Somaliland"= "#FFCD2C",
                     "Puntland" = "#FFCD2C",
                     "Sudan" = "#F77613",
                     "Kenya" = "#00008B")
```

```
ggplot() +
  geom_sf(data = lands,
          mapping = aes(fill = geonunit),
          colour = NA,
          alpha = 0.75) +
  coord_sf(crs = sf::st_crs(4326),
           xlim = c(22, 50),
           ylim = c(-5, 21)) +
  scale_fill_manual(name = "Country",
                    values = country_colours)
```



Rmanual_files/figure-latex/map-1.pdf

5

Manuscript Preparation with RMarkdown in R

5.1 Publishing with R Markdown

Yihui Xie, who has written several of the most essential packages in this space, including knitr, first introduced R Markdown in 2012. Xie presents R Markdown as "an authoring framework for data science" in his book R Markdown: The Definitive Guide. The `.Rmd` file is used to automatically "save and execute code, as well as generate high-quality reports." Documents and presentations in **HTML, PDF, Word, Markdown, Powerpoint, LaTeX**, and more formats are all supported. Homework assignments, journal articles, full-length books, formal reports, presentations, web pages, and interactive dashboards are just a few of the possibilities he offers, with examples of each. It is an understatement to suggest that it is a flexible and dynamic environment.

5.2 R Markdown Syntax

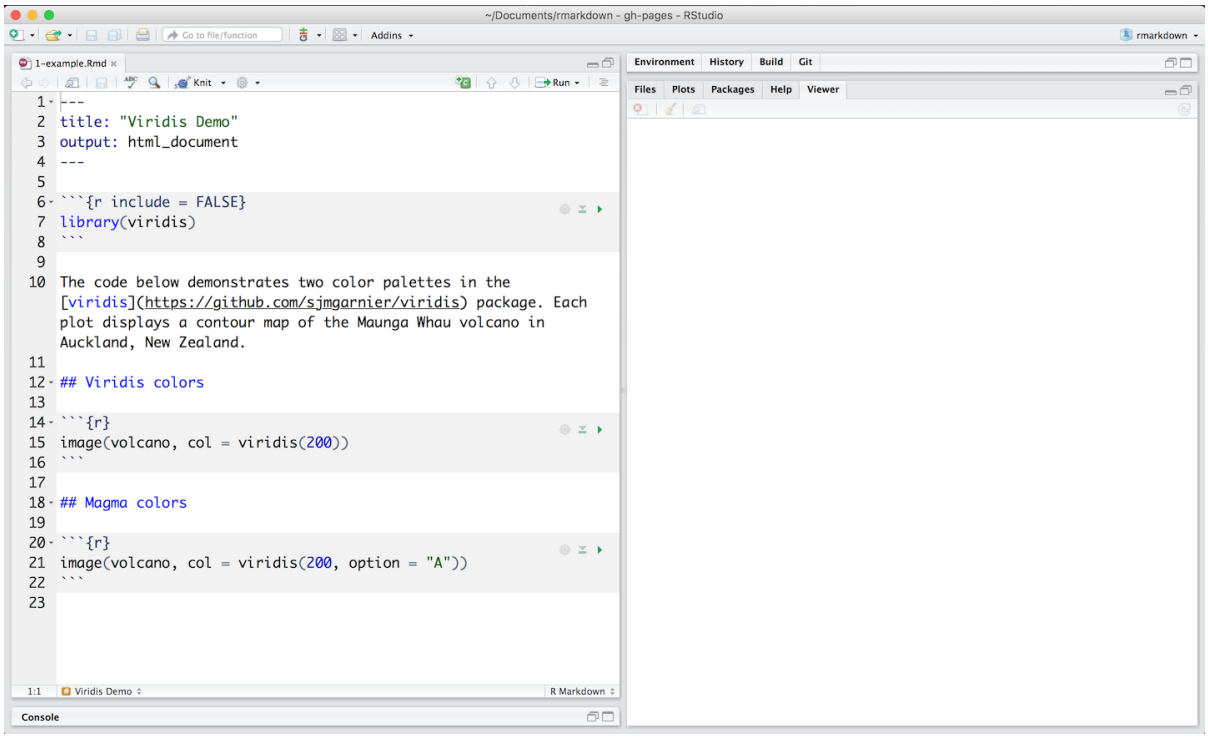
R Markdown files consist of the following elements:

- A metadata header that uses the YAML syntax to configure multiple output choices.
- Markdown-formatted content in the body
- Three `backticks` separate code chunks, which are separated by a curly braced block that identifies the language and sets chunk settings.

R Markdown also allows users to embed code expressions in the prose by enclosing them in single backticks. A minimal `.Rmd` example is included below. This is based on the Xie's sample, which I've modified to demonstrate all of the elements mentioned above.

```
---
02: title: "Hello Dinsho"
03: author: "Mesfin Diro"
04: output: html_document
05: ---
06:
07: This is a paragraph in an _R Markdown_ document.
08: Below is a code chunk:
09:
10: ```{r fit-plot, echo=TRUE}
11: fit = lm(dist ~ speed, data = cars)
12: b   = coef(fit)
13: plot(cars)
14: abline(fit)
15: ```
16:
17: The slope of the regression is `r round(b[2], digits = 3)`.
```

- R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both
  - save and execute code
  - generate high quality reports that can be shared with an audience
- R Markdown documents are fully reproducible and support dozens of static and dynamic output formats. This is an R Markdown file, a plain text file that has the extension `.Rmd`.




```
---
title: "Viridis Demo"
output: html_document
---

```{r include = FALSE}
library(viridis)
```

The code below demonstrates two color palettes in the
[viridis](https://github.com/sjmgarnier/viridis) package. Each
plot displays a contour map of the Maunga Whau volcano in
Auckland, New Zealand.

Viridis colors

```{r}
image(volcano, col = viridis(200))
```

Magma colors

```{r}
image(volcano, col = viridis(200, option = "A"))
```
```

- The file contains three types of content: 1. An (optional) YAML header surrounded by `---`s 2. R code chunks surrounded by ```` ```s ```` 3. text mixed with simple text formatting

---

5.3 Rendering output

- To generate a report from the file, run the render command:

```
library(rmarkdown)
render("example_1.Rmd)
```

- Better still, use the **"Knit"** button in the RStudio IDE to render the file and preview the output with a single click or keyboard shortcut.r

- R Markdown generates a new file that contains selected text, code, and results

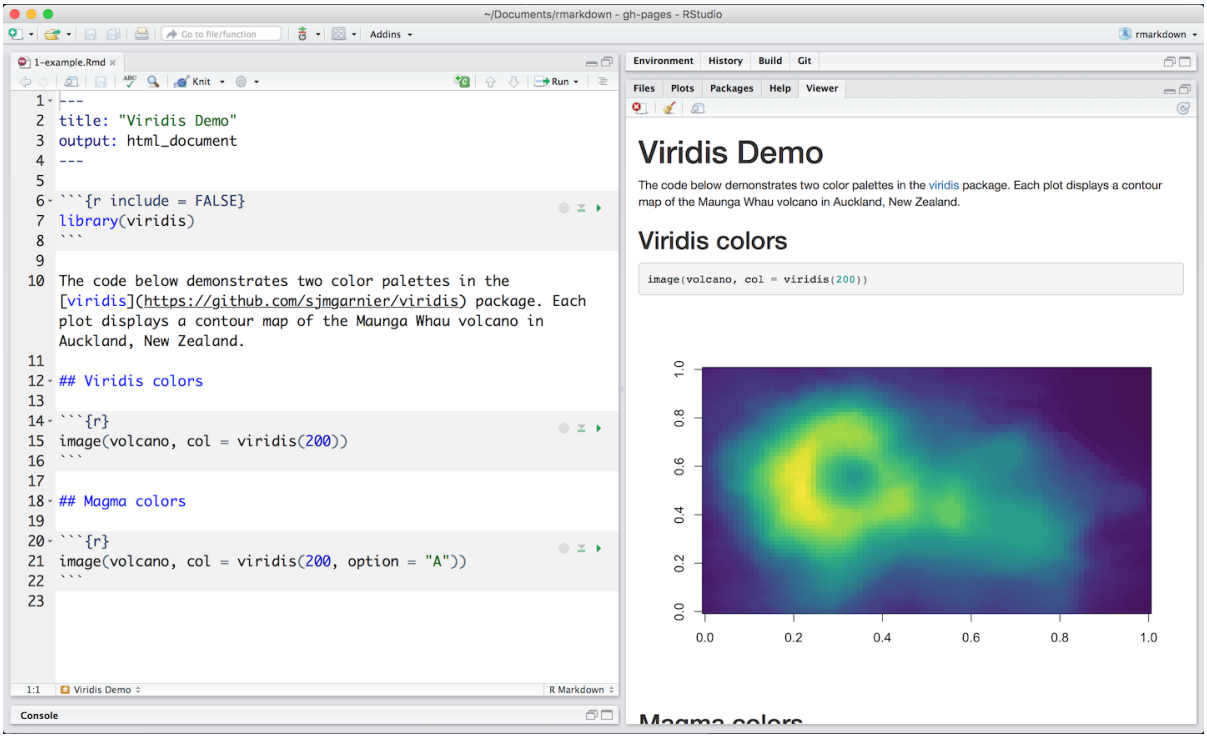


Figure 5.1 R Markdown with Knit button in RStudio

from the .Rmd file. The new file can be a finished **web page, PDF, MS Word document, slide show, notebook, handout, book, dashboard, package vignette** or other **format**.

5.4 How it works

Figure 5.2 R Markdown with Knit button in RStudio

- When you run render, R Markdown sends the `.Rmd` file to knitr, which runs all

of the code chunks and generates a new markdown (`.md`) document with the code and its output.

- **Knitr** generates a markdown file, which is then processed by pandoc, which generates the final format.
- While this may appear hard, R Markdown simplifies the process by combining all of the above processing into a single render function.

5.5 Code Chunks

- The R Markdown file below contains three code chunks:

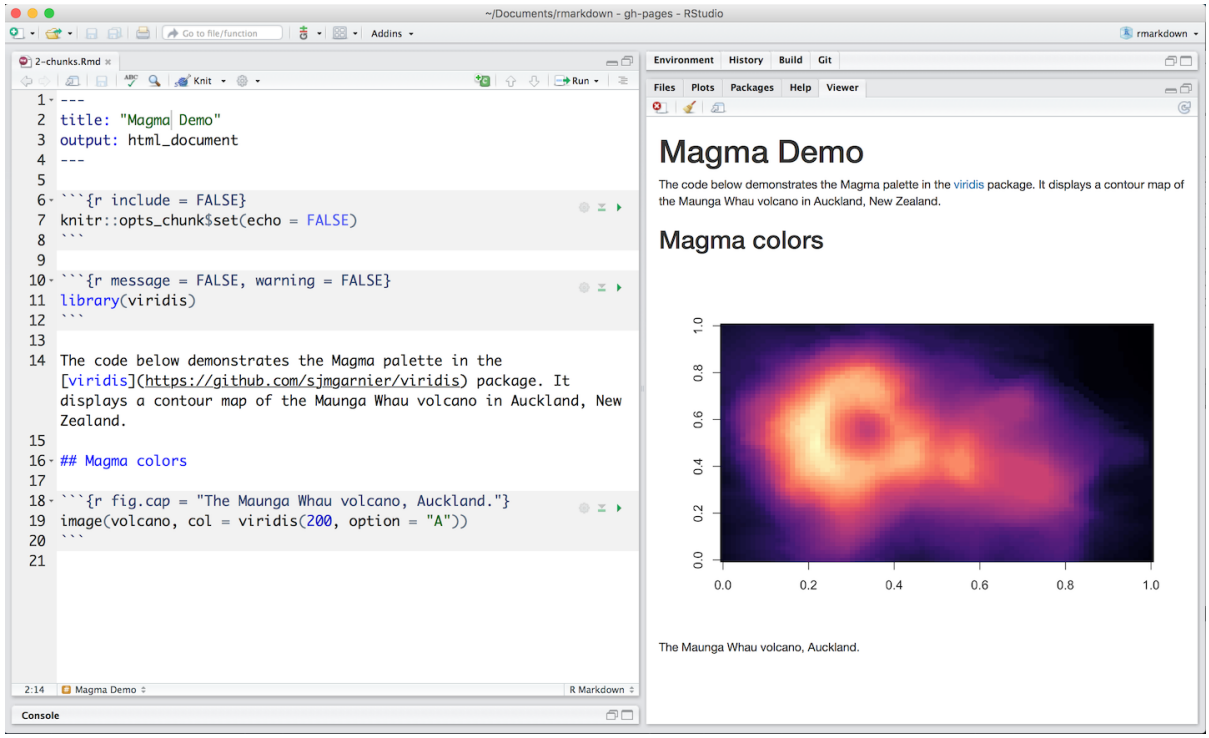


Figure 5.3 R Markdown with Knit button in RStudio

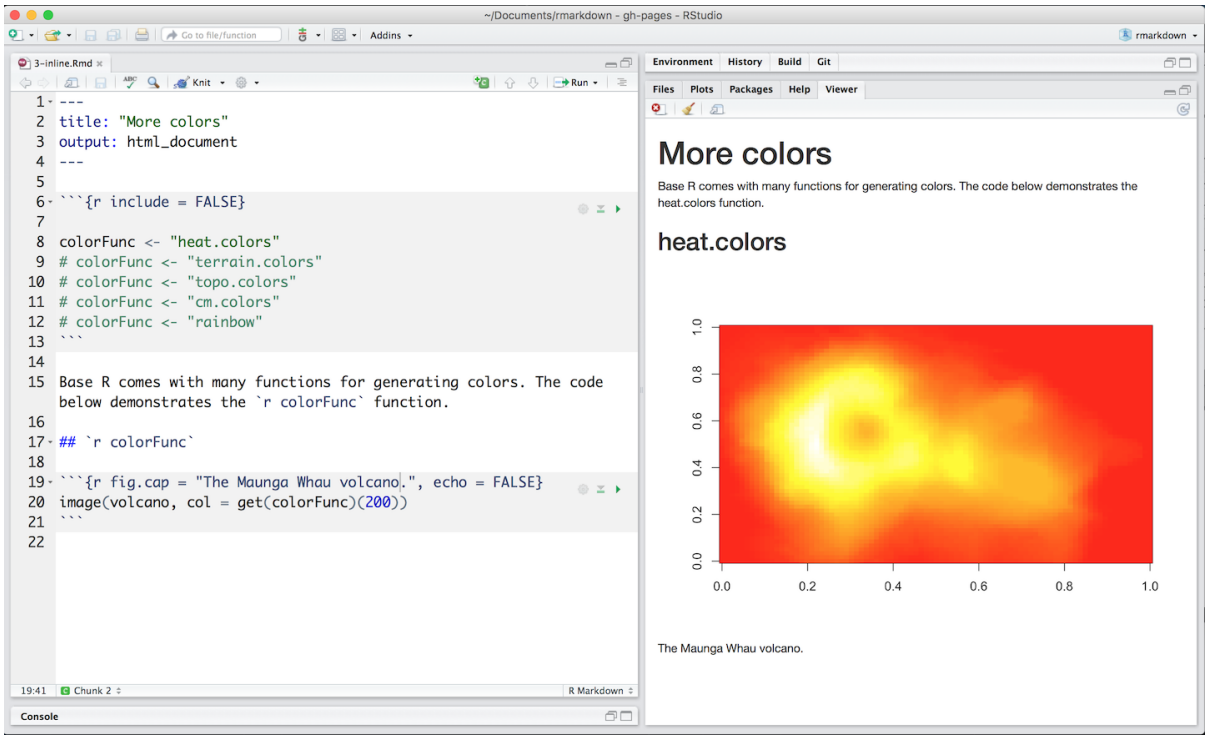
- You can quickly insert chunks like these into your file with
  - the keyboard shortcut **Ctrl + Alt + I** (OS X: **Cmd + Option + I**)

– the Add Chunk  command in the editor toolbar or or by typing the chunk delimiters **{r}__ and __**.

- When you render your .Rmd file, R Markdown will run each code chunk and embed the results beneath the code chunk in your final report.

5.6 Inline Code

- Code results can be inserted directly into the text of a `.Rmd` file by enclosing the code with `` `r` ``. The file below uses `` `r` `` twice to call **colorFunc**, which returns "heat.colors".



- Using `` `r` `` makes it easy to update the report to refer to another function.

5.7 Code Languages

- knitr can execute code in many languages besides R. Some of the available language engines include:
  - Python
  - SQL
  - Bash
  - Rcpp
  - Stan
  - JavaScript
  - CSS
- To process a code chunk using an alternate language engine, replace the `r` at the start of your chunk declaration with the name of the language:

  {bash}

Appendix 1

.1 Installing R and Rstudio

- R is free
- Often, less codes are needed in R to plot an elegant graph
- for/while loops (basic concepts in programming) are not necessary in R to make a production-quality graph
- R is the best software for statistical analysis

.1.1 installing R

1. You can download R from the Comprehensive R Archive Network (CRAN). Search for CRAN on your browser:

2. Select the version for your operating system from the CRAN page: Linux, Mac OS X, or Windows.

3. You have numerous options once you get at the CRAN download page. The base subfolder is what you want to install. This sets up the foundational packages you'll need to get started. Instead of using this URL, we'll learn how to install additional required programs from within R.

4. To begin the download, click the link for the most **latest version**.

5. When the installer file has finished downloading, click that tab to begin the installation process. As a result, you'll need to locate where they store downloaded files and click on them to begin the process.

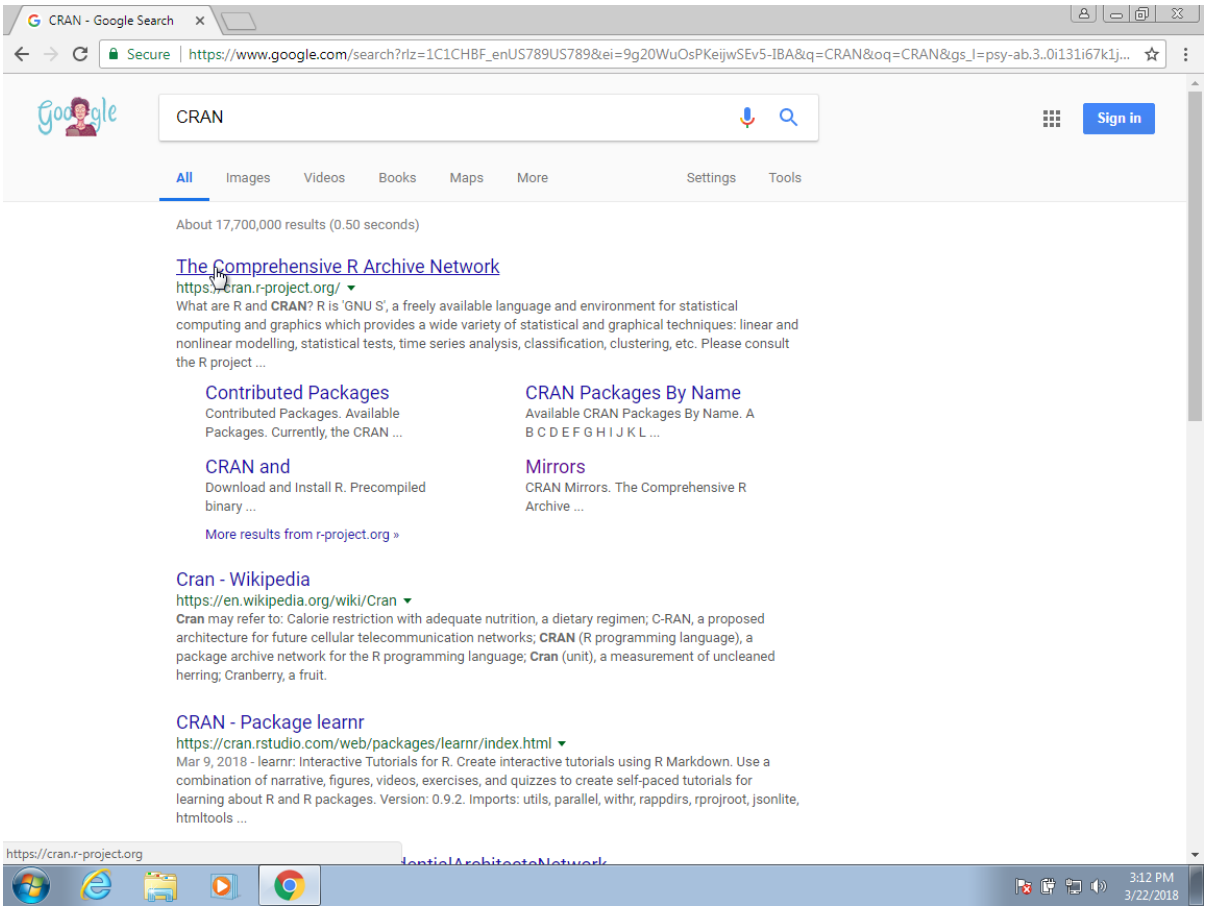


Figure 4 Comprehensive R Archive Network

6. To complete the installation, navigate through the various options. All of the default options are recommended.

7. Select the default even when you get an ominous warning.

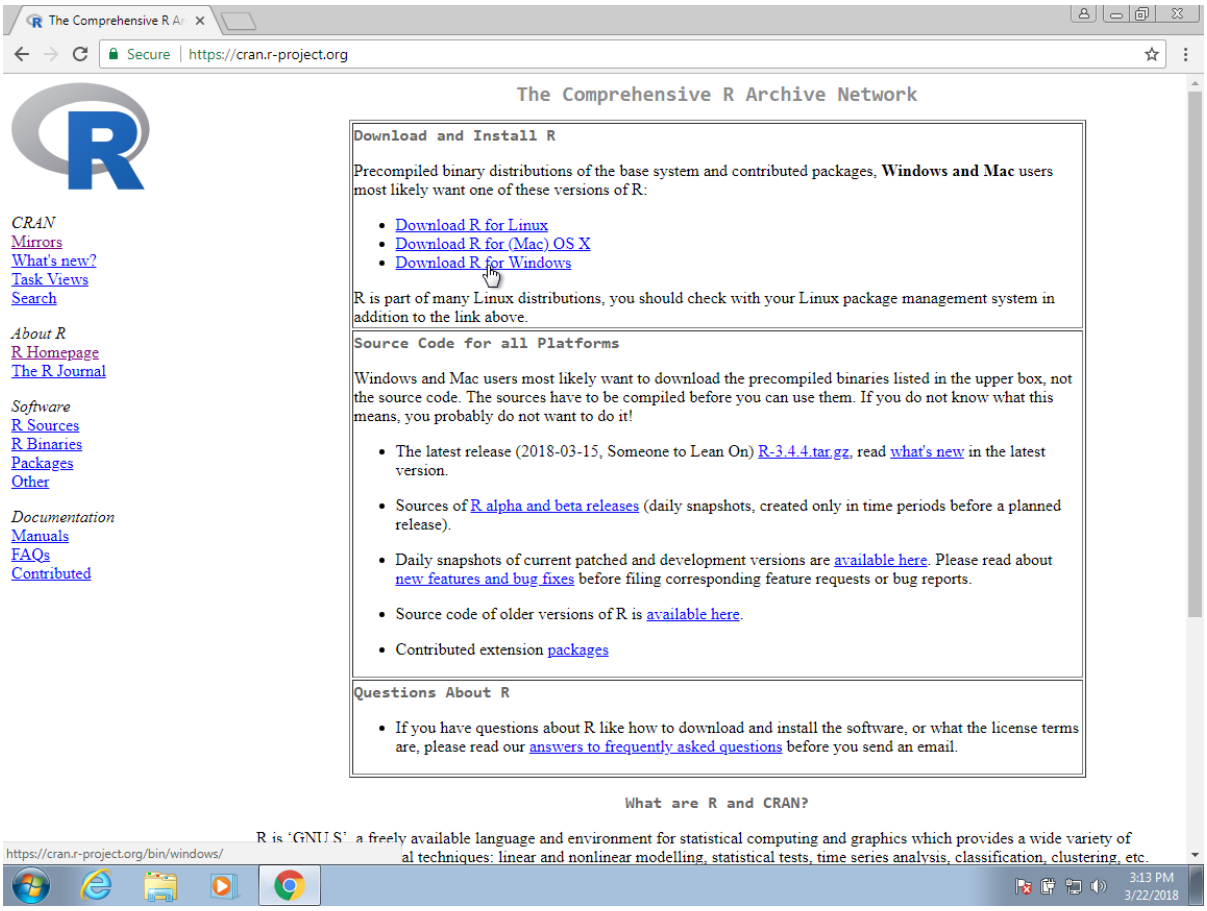
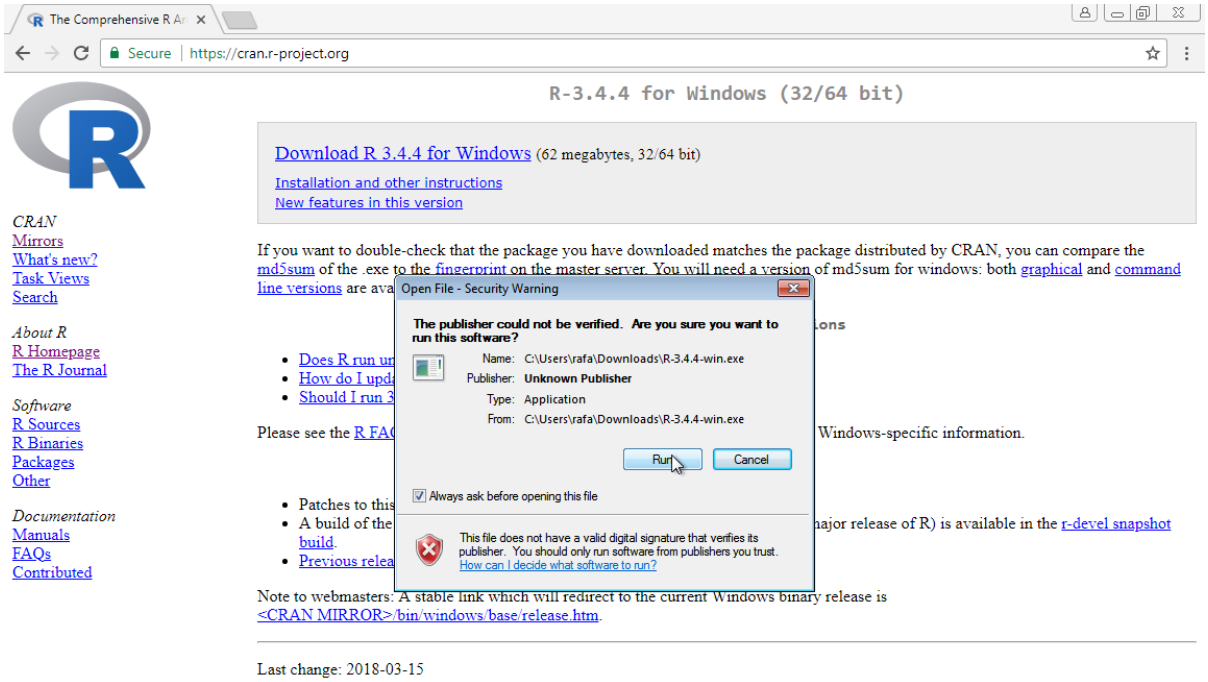


Figure 5 R Version for Linux, Mac OS X or Windows



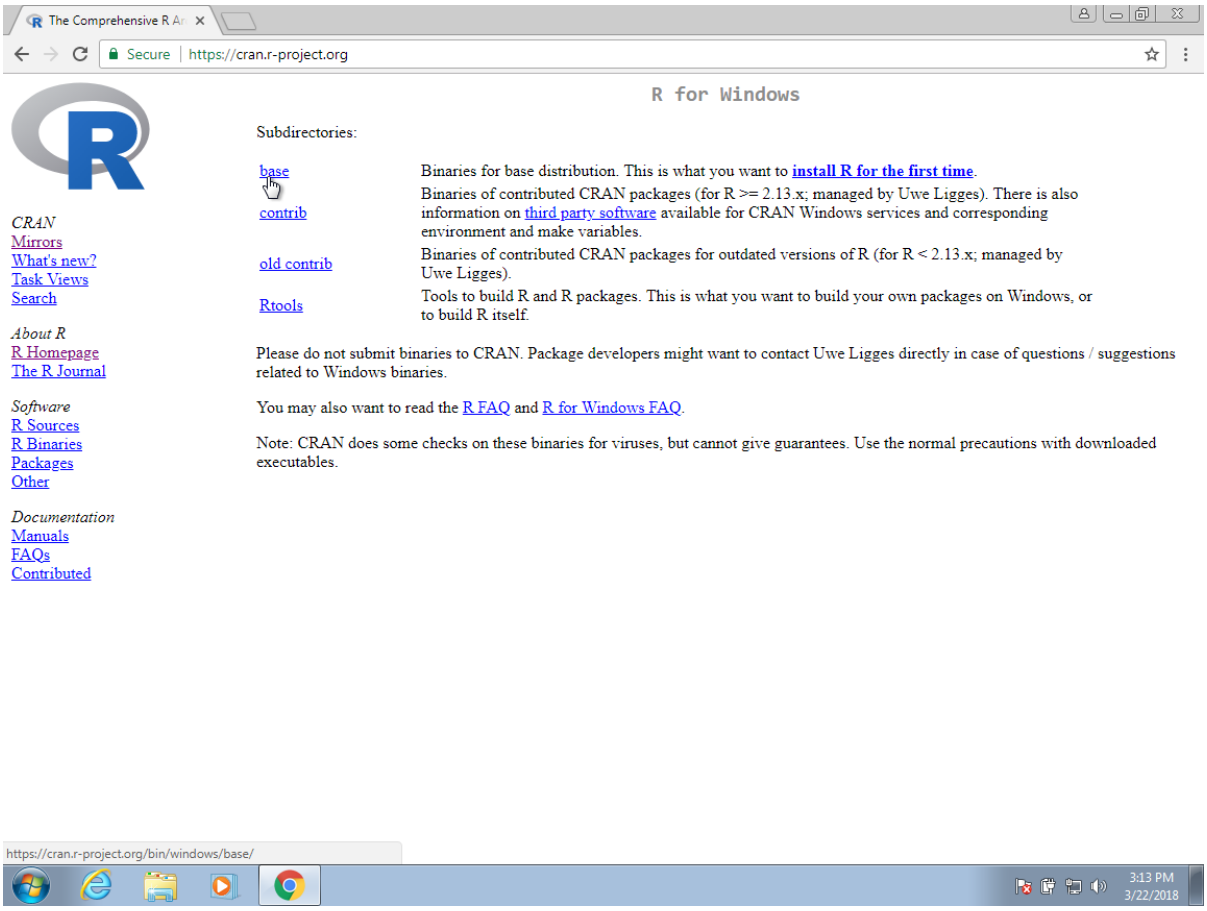


Figure 6 R Version for Linux, Mac OS X or Windows

8. When selecting the language, consider that it will be easier to follow this book if you select English

9. Continue to select all the defaults:

.1.2 Installing RStudio

RStudio is an interactive desktop environment, but it is not R, and when you download and install it, it does not include R. As a result, before we can utilize RStudio, we must first install R.

Download requirements for RStudio

- Download R: http://cran.r-project.org/
- RStudio: http://www.rstudio.org/

Figure 7 R Version for Linux, Mac OS X or Windows

- For Mac users only : XQuartz: https://www.xquartz.org/

.1.3 Open RStudio

When you start RStudio for the first time, you will see three panes. The left pane shows the R console. On the right, the top pane includes tabs such as Environment and History, while the bottom pane shows five tabs: File, Plots, Packages, Help, and Viewer (these tabs may change in new versions). You can click on each tab to move across the different features.

To start a new script, you can click on **file**, then **New File**, then **R Script**

- Rstudio is where people do R programming. You can type codes (commands) into the console (bottom-left panel).
  - > means that the console is ready to receive more code.

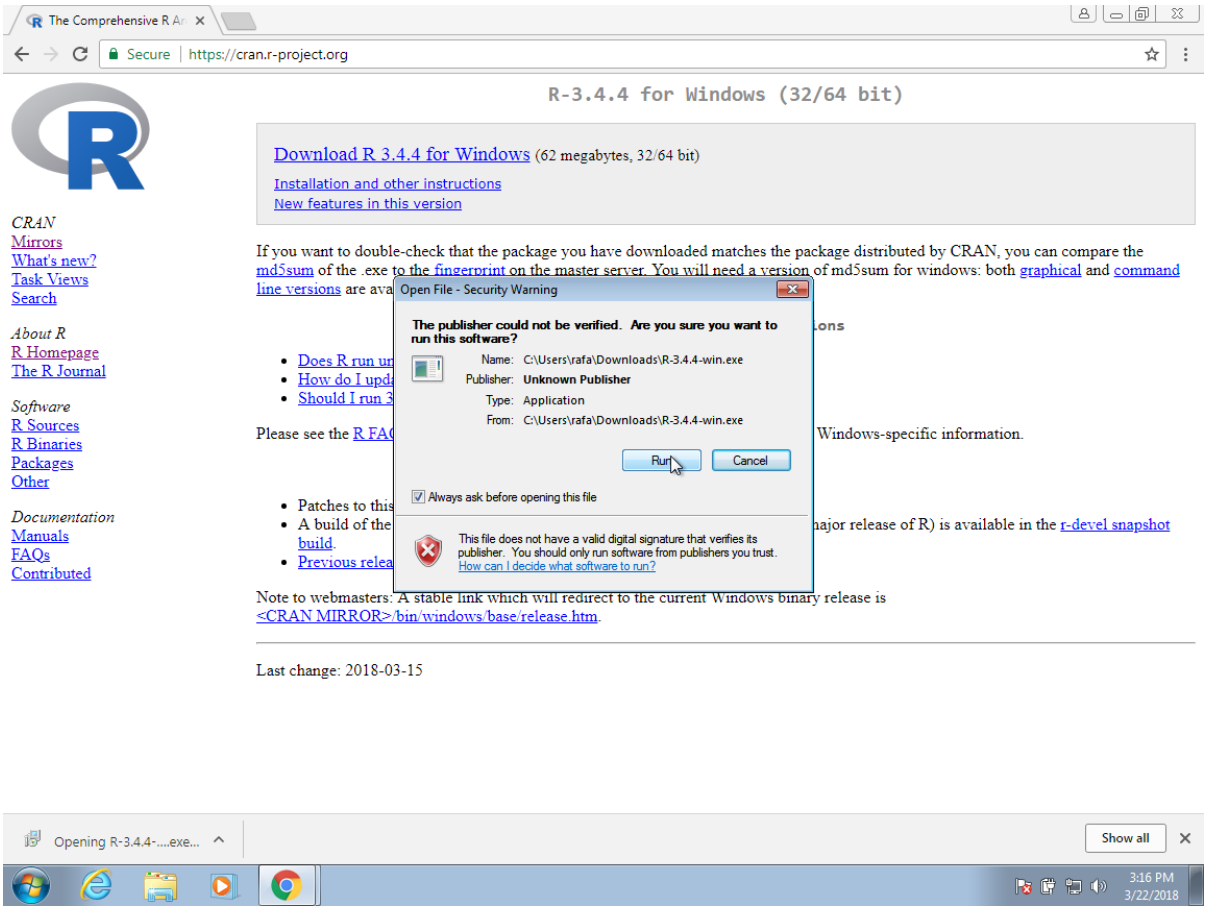


Figure 8 R Version for Linux, Mac OS X or Windows

  - + means your code is not complete.

- You can also write (longer) codes in the script within the code editor (top-left panel).

  - The code editor will run the script into the console.
  - A new script can be opened by clicking: **File -> New -> R Script**.

- You can run a script by clicking **"Run"** with the green arrow or by typing **ctrl + enter**. It is labeled with the red circle.

  - Or you can just type your codes directly into the console.

How to set working directory

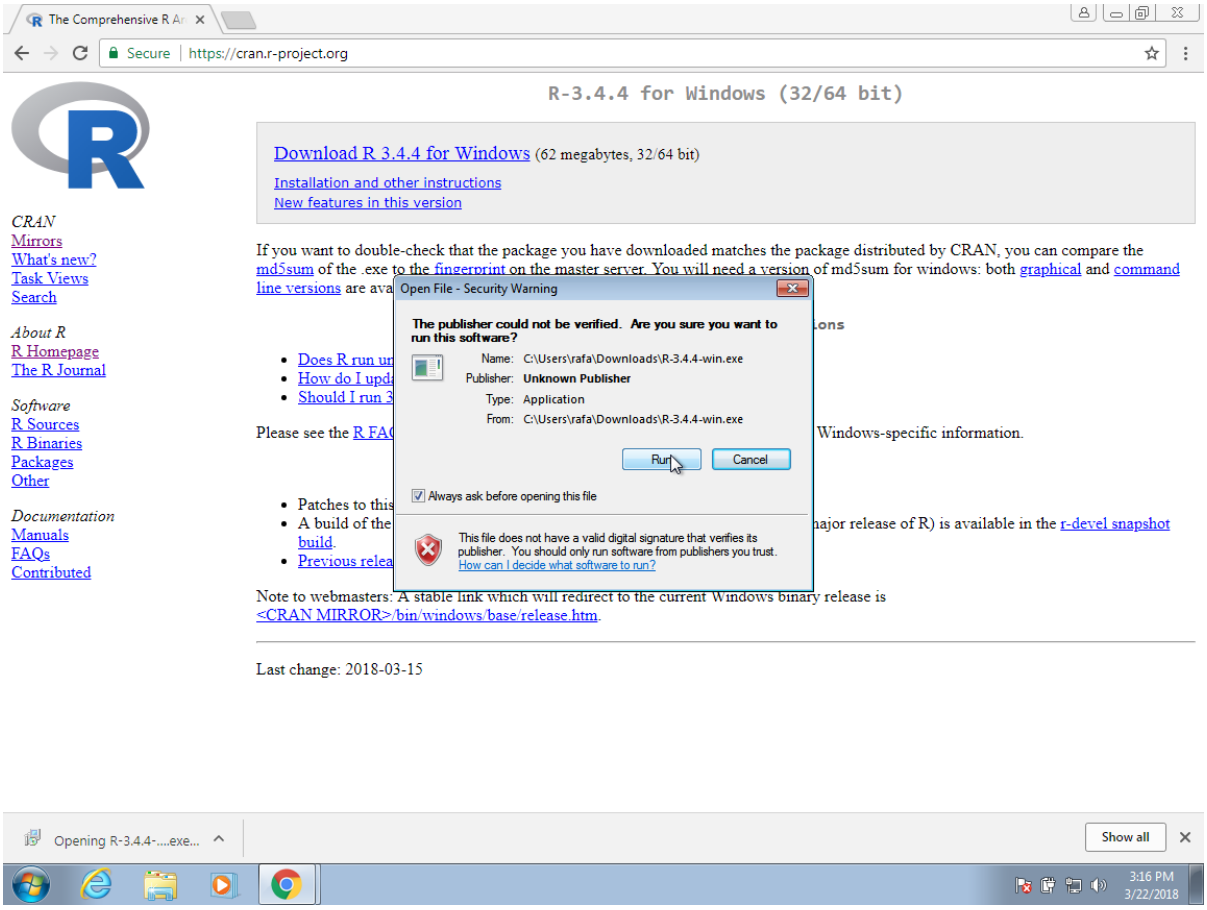


Figure 9 R installation startup

.1.4 Installing R packages

A fresh install of R only provides a small portion of the capability available. In reality, what you get after your initial setup is referred to as base R. Developer-created add-ons provide the additional capabilities. Hundreds of these are presently available through CRAN, with many more provided via other sources such as GitHub. R, on the other hand, makes distinct components available via packages because not everyone requires all of the available capabilities. R makes installing packages from within R very simple. To install the **tidyverse** package, for example, which we use to share datasets and code for this book, type:

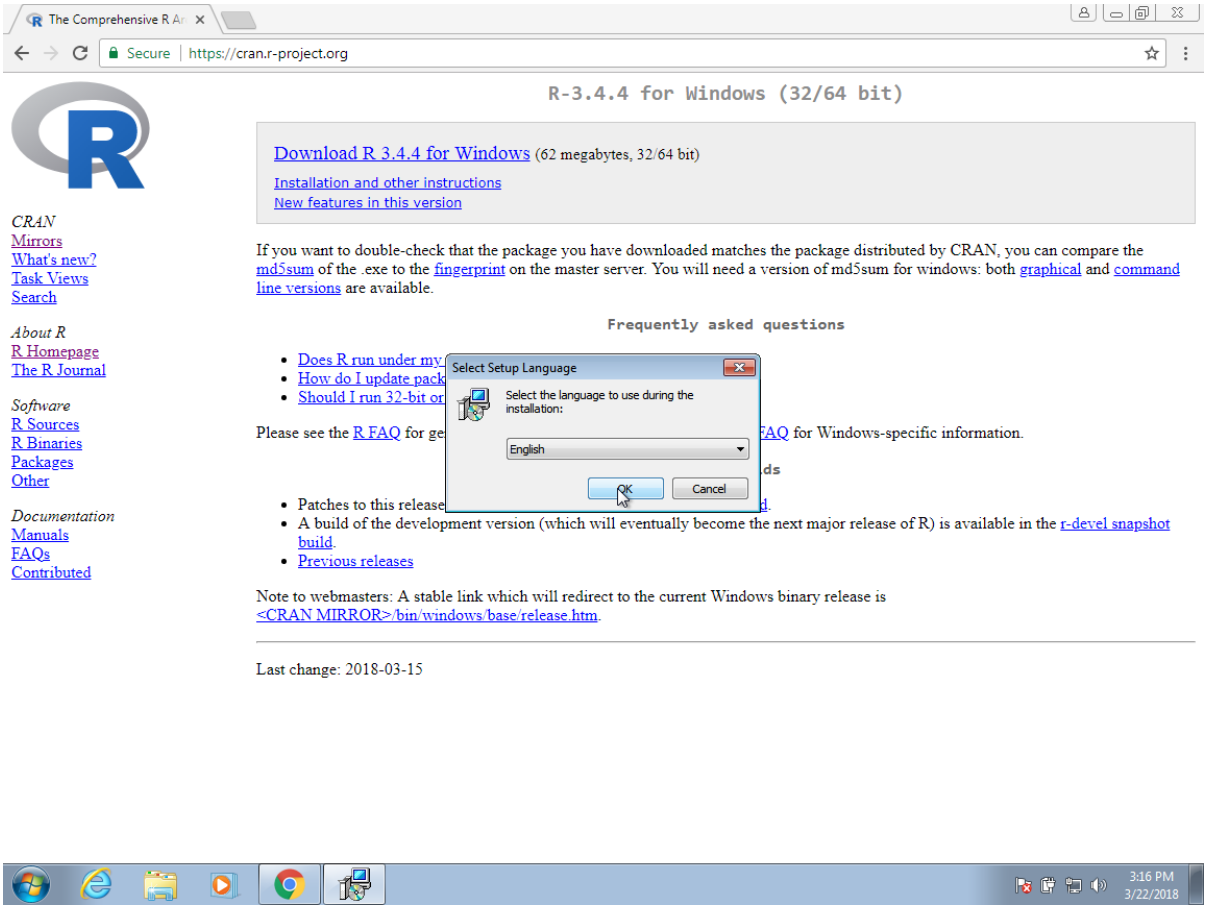


Figure 10 R Installation language selection

```
install.packages("tidyverse")
```

Navigate to the Tools menu in RStudio and click Install Packages. We can then use the **library** function to load the package into our R sessions:

```
library(tidyverse)
```

Until we exit the R session, the package remains loaded. If you encounter an error when trying to load a package, it implies you need to install it first. By passing a character vector to this function, we can also install multiple packages at once:

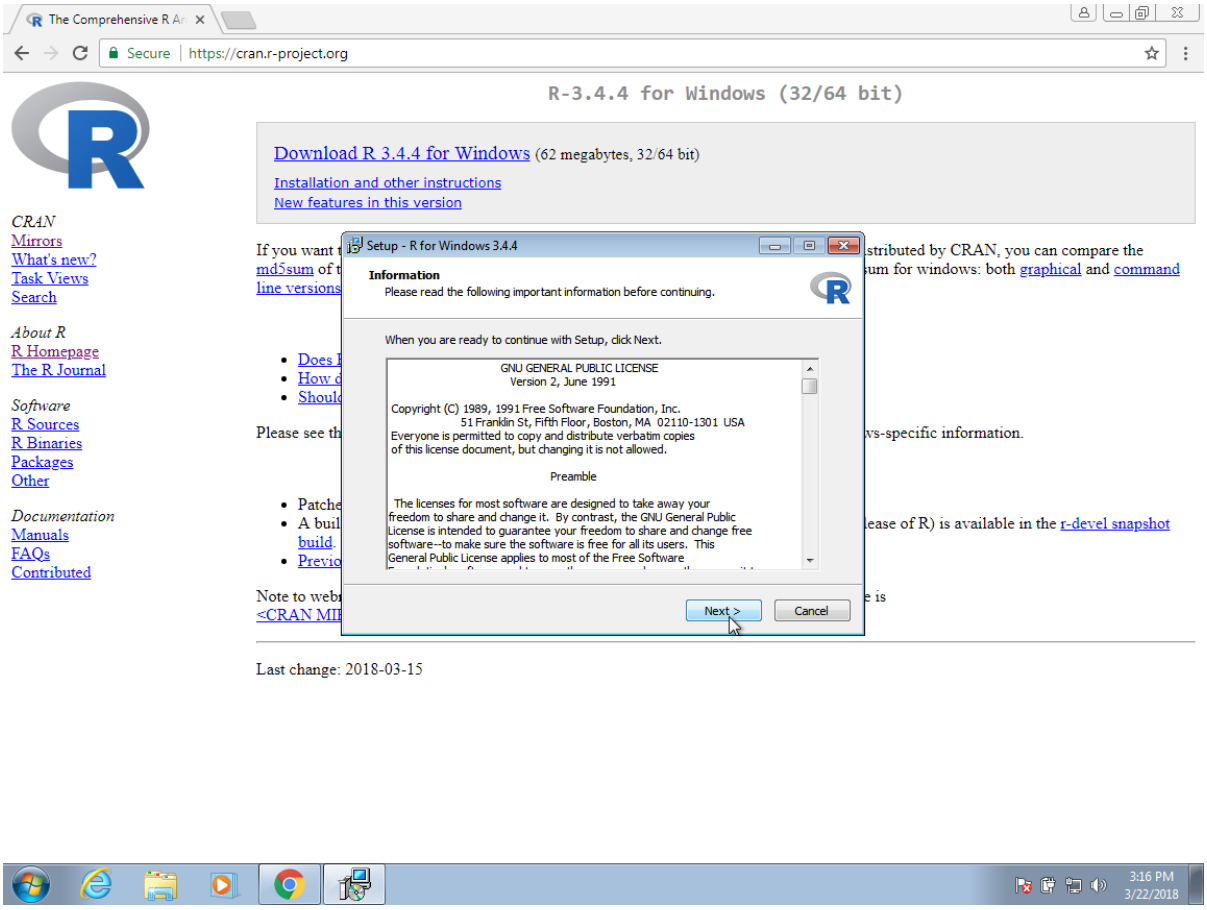


Figure 11 R Installation defaults

```
install.packages(c("tidyverse","ggpubr"))
```

It's worth noting that tidyverse installs multiple packages. When a package contains dependencies or uses functions from other packages, this happens frequently. When you use library to load a package, you also load its dependencies.

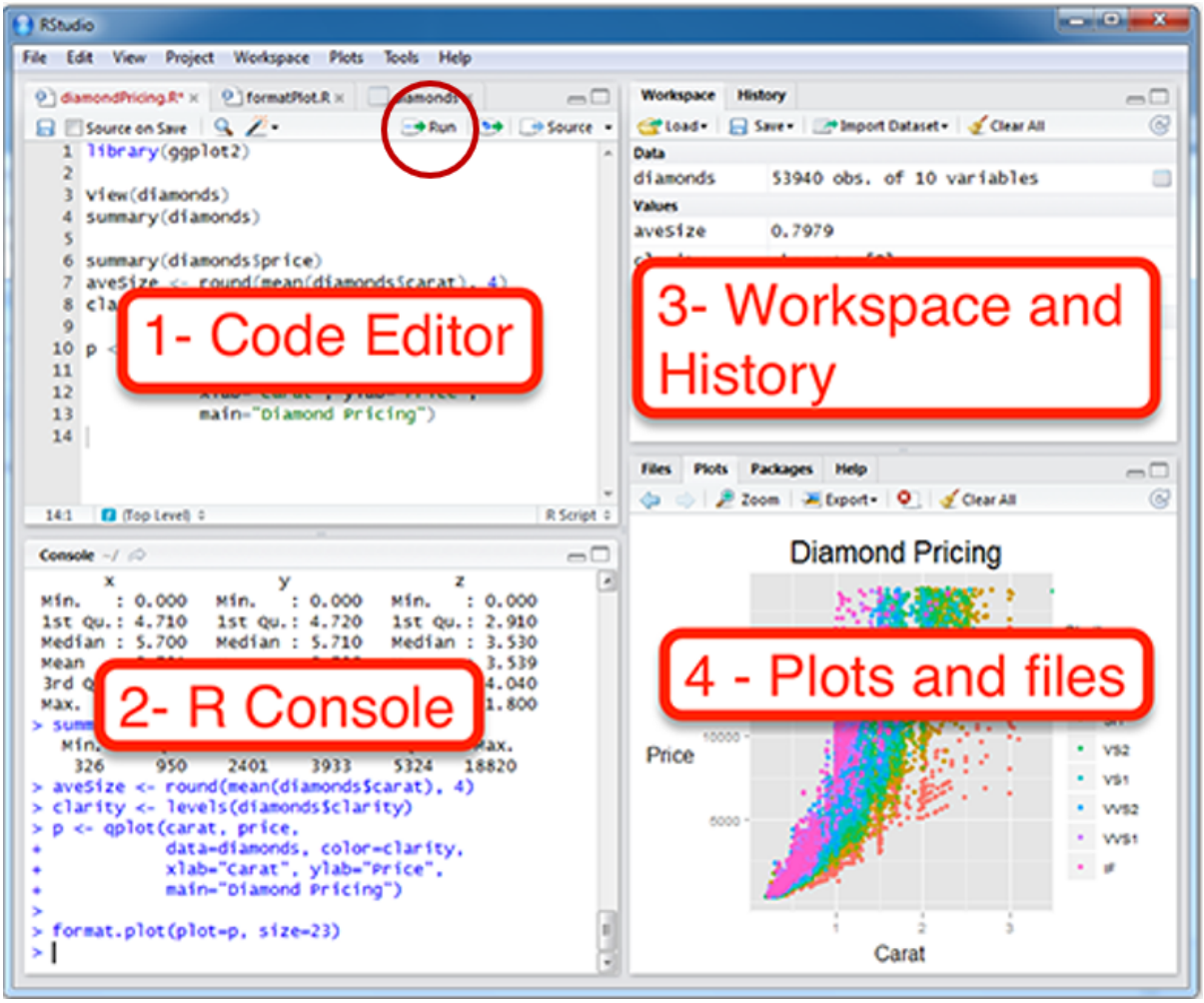


Figure 12 Rstudio layout

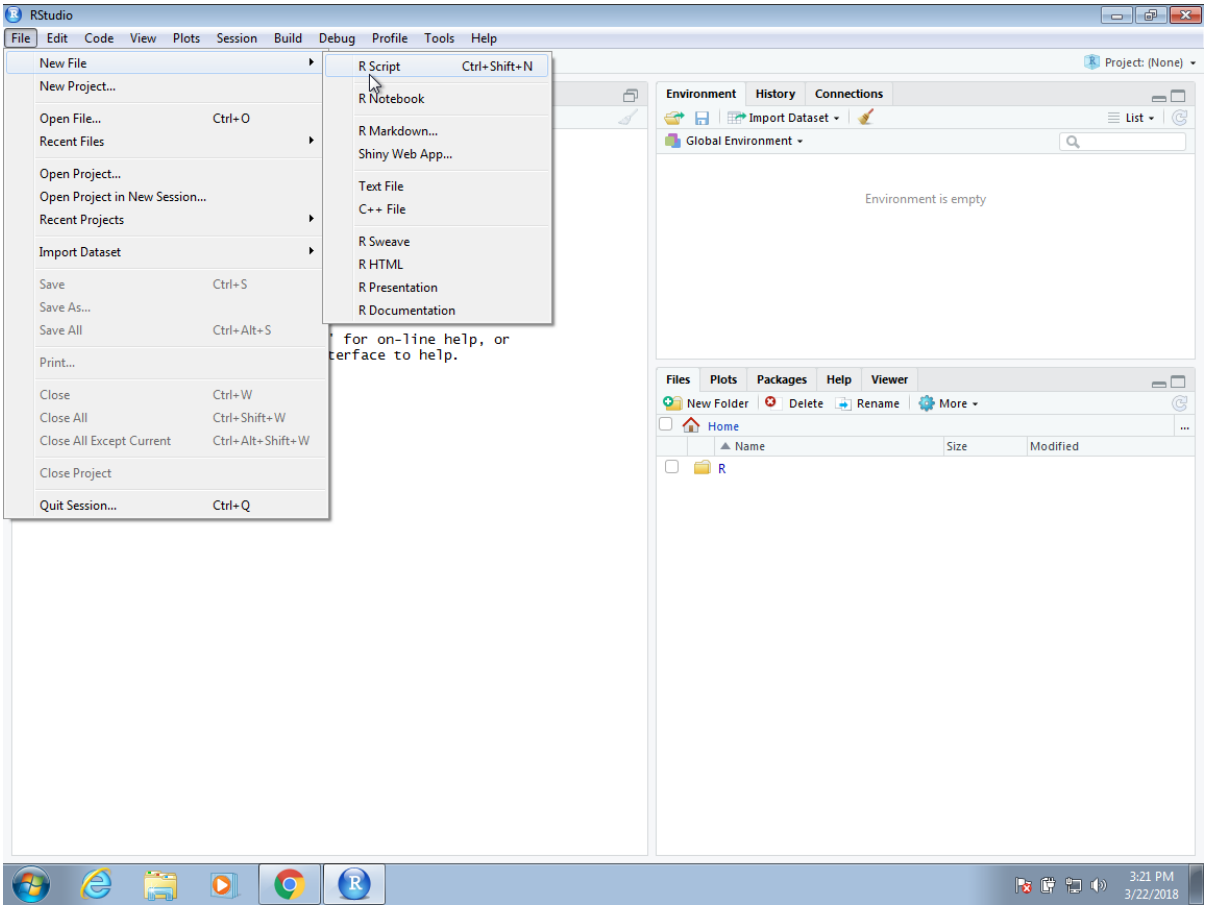


Figure 13 Setting your working directory

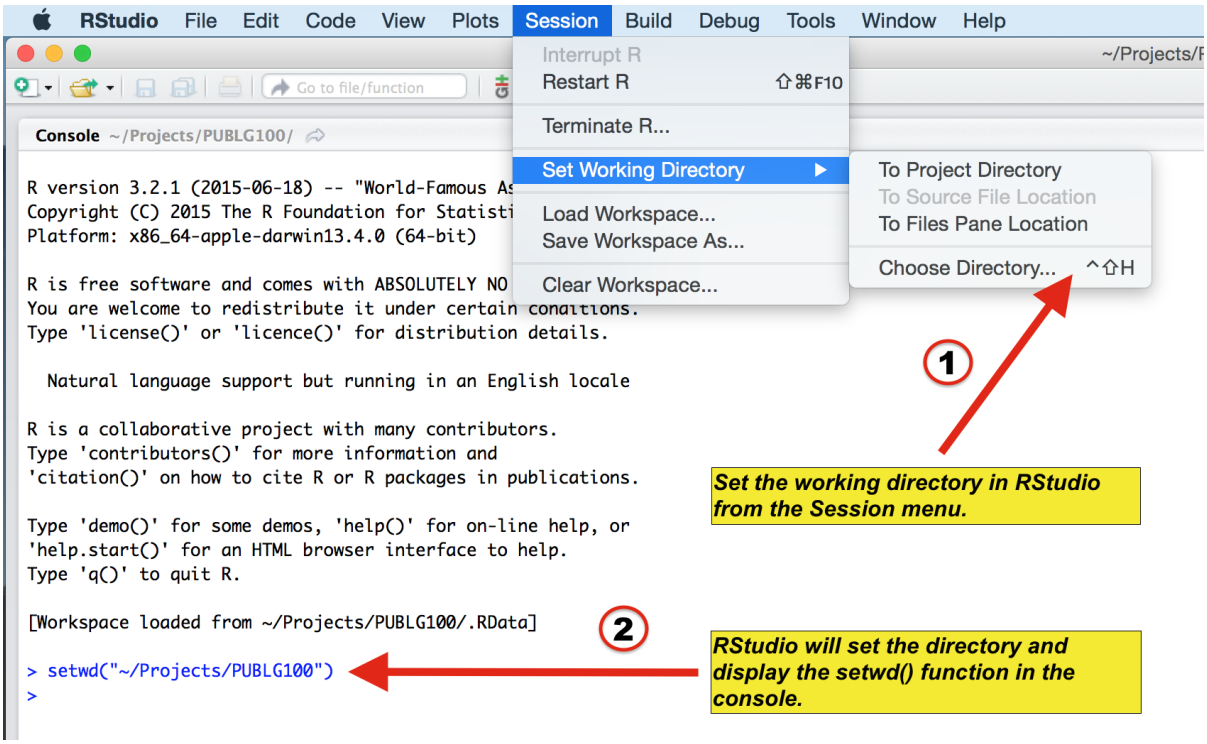


Figure 14 Setting your working directory

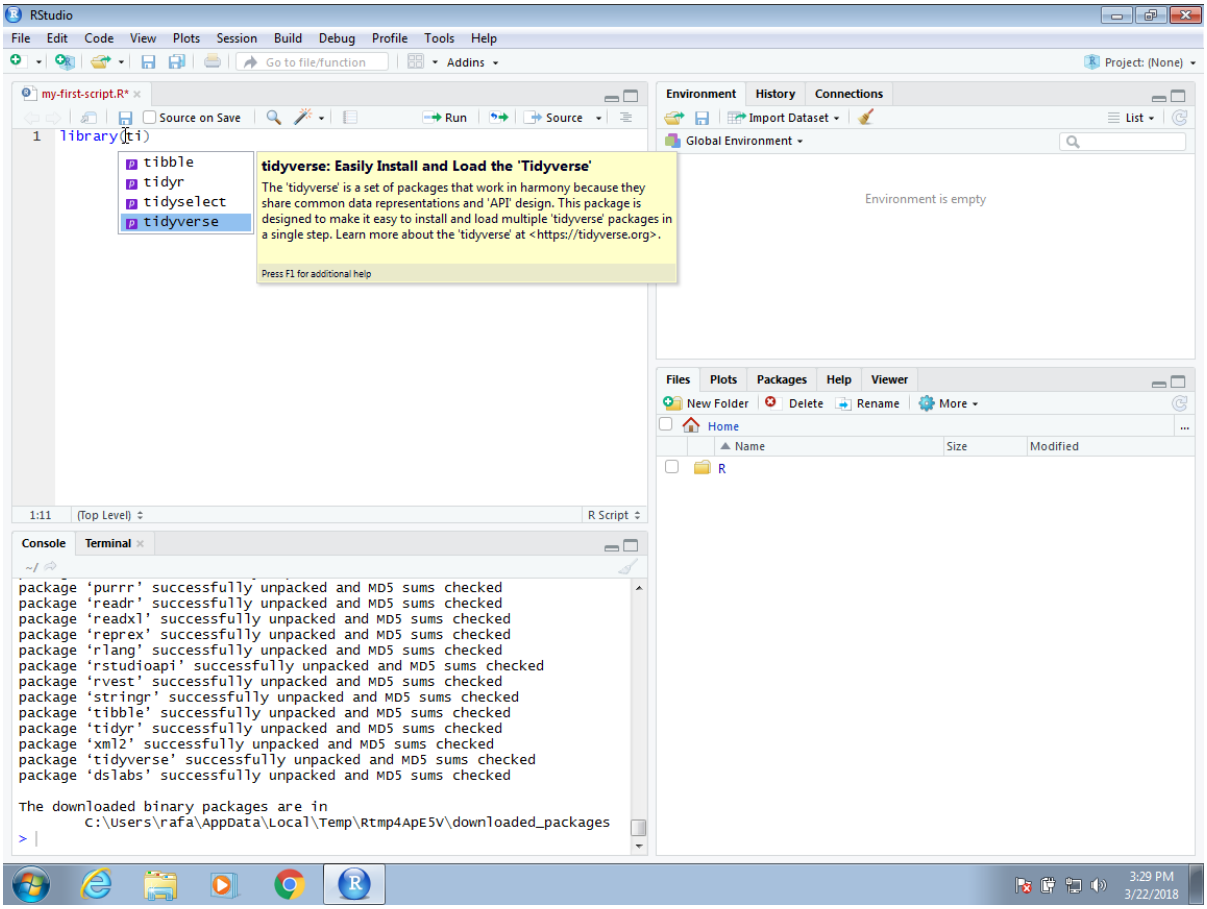


Figure 15 Setting your working directory

Appendix 2

.2 Installing OpenRefine

The core functions of OpenRefine do not require internet access. It runs as a small web server on your own computer when you download and install it, and you access it using your browser. To import data from the online, reconcile data using a web service, or export data to the web, all you need is an internet connection.

To run OpenRefine, you'll need three things on your computer:

.2.1 Compatible operating system

- OpenRefine is designed to work with Windows, Mac, and Linux operating systems

.2.2 Java

- To run OpenRefine, you must have Java installed and configured on your machine. The Mac version of OpenRefine contains Java, while a Windows package with Java was added in OpenRefine 3.4.
- If you install and run OpenRefine on a Windows computer without Java, it will open a browser window with instructions for downloading Java.
- Before installing OpenRefine, we recommend that you first download and install Java. Please keep in mind that OpenRefine 3.5 supports Java 8 through Java 17.

.2.3 Compatible browser

- OpenRefine works well with Webkit-based browsers, such as:
  - Chromium Google Chrome

  - Opera
  - Edge by Microsoft

- Other browsers, such as Firefox, have some minor rendering and performance difficulties. **Internet Explorer is not supported**.

References

1. Xie, Y. *Dynamic documents with R and knitr*. (Chapman; Hall/CRC, 2015).

2. Xie, Y. *Bookdown: Authoring books and technical documents with r markdown*. (2020).

3. Team, R. C. *R: A language and environment for statistical computing*. (R Foundation for Statistical Computing, 2020).

4. Wickham, H., François, R., Henry, L. & Müller, K. *Dplyr: A grammar of data manipulation*. (2020).

5. Wickham, H. *Forcats: Tools for working with categorical variables (factors)*. (2020).

6. Attali, D. & Baker, C. *ggExtra: Add marginal histograms to 'ggplot2', and more 'ggplot2' enhancements*. (2019).

7. Wickham, H. *et al. ggplot2: Create elegant data visualisations using the grammar of graphics*. (2020).

8. Arnold, J. B. *Ggthemes: Extra themes, scales and geoms for 'ggplot2'*. (2019).

9. Auguie, B. *gridExtra: Miscellaneous functions for "grid" graphics*. (2017).

10. Urbanek, S. *Jpeg: Read and write JPEG images*. (2019).

11. Zhu, H. *kableExtra: Construct complex table with 'kable' and pipe syntax*. (2019).

12. Xie, Y. *Knitr: A general-purpose package for dynamic report generation in r*. (2020).

13. Cheng, J., Karambelkar, B. & Xie, Y. *Leaflet: Create interactive web maps with the JavaScript leaflet library*. (2019).

14. Rudis, B., Lohmann, N., Bandyopadhyay, D. & Kettner, L. *Ndjson: Wicked-fast streaming 'JSON' ('ndjson') reader*. (2019).

15. Urbanek, S. *Png: Read and write PNG images*. (2013).

16. Henry, L. & Wickham, H. *Purrr: Functional programming tools*. (2020).

17. Temple Lang, D. *RCurl: General network (HTTP/FTP/...) Client interface for r*. (2020).

18. Wickham, H., Hester, J. & Francois, R. *Readr: Read rectangular text data*. (2018).

19. Ushey, K., Allaire, J. & Tang, Y. *Reticulate: Interface to 'python'*. (2020).

20. Couture-Beil, A. *Rjson: JSON for r*. (2018).

21. Allaire, J. *et al. Rmarkdown: Dynamic documents for r*. (2020).

22. Chamberlain, S. *Rorcid: Interface to the 'orcid.org' API*. (2020).

23. Barbera, P. *streamR: Access to twitter streaming API via r*. (2018).

24. Wickham, H. *Stringr: Simple, consistent wrappers for common string operations*. (2019).

25. Müller, K. & Wickham, H. *Tibble: Simple data frames*. (2020).

26. Wickham, H. & Henry, L. *Tidyr: Tidy messy data*. (2020).

27. Wickham, H. *Tidyverse: Easily install and load the 'tidyverse'*. (2019).

28. Xie, Y. *Bookdown: Authoring books and technical documents with R markdown*. (Chapman; Hall/CRC, 2016).

29. Wickham, H. *ggplot2: Elegant graphics for data analysis*. (Springer-Verlag New York, 2016).

30. Xie, Y. *Dynamic documents with R and knitr*. (Chapman; Hall/CRC, 2015).

31. Xie, Y., Allaire, J. J. & Grolemund, G. *R markdown: The definitive guide*. (Chapman; Hall/CRC, 2018).

32. Wickham, H. *et al.* Welcome to the tidyverse. *Journal of Open Source Software* **4**, 1686 (2019).

33. Becke, A. D. Perspective: Fifty years of density-functional theory in chemical physics. *The Journal of Chemical Physics* **140**, 18A301 (2014).

34. Lewis, N. S. Powering the planet. *MRS Bulletin* **32**, 808–820 (2007).

35. Fert, A. The origin, development and future of spintronics. *Phys. Usp.* **51**, 1336 (2008).

36. Fert, A. Nobel lecture: Origin, development, and future of spintronics. *Rev. Mod. Phys.* **80**, 1517–1530 (2008).

37. Wynants, L. *et al.* Prediction models for diagnosis and prognosis of covid-19 infection: Systematic review and critical appraisal. *BMJ* **369**, (2020).

38. Mahapatra, S. *et al.* Repurposing therapeutics for COVID-19: Rapid prediction of commercially available drugs through machine learning and docking. *medRxiv* (2020) doi:10.1101/2020.04.05.20054254.

39. Xie, Y. Knitr: A comprehensive tool for reproducible research in R. in *Implementing reproducible computational research* (eds. Stodden, V., Leisch, F. & Peng, R. D.) (Chapman; Hall/CRC, 2014).

40. Blum, L. C. & Reymond, J.-L. 970 million druglike small molecules for virtual screening in the chemical universe database GDB-13. *Journal of the American Chemical Society* **131**, 8732–8733 (2009).

41. Ruddigkeit, L., Deursen, R. van, Blum, L. C. & Reymond, J.-L. Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. *Journal of Chemical Information and Modeling* **52**, 2864–2875 (2012).

42. Ruddigkeit, L., Blum, L. C. & Reymond, J.-L. Visualization and virtual screening of the chemical universe database GDB-17. *Journal of Chemical Information and Modeling* **53**, 56–65 (2013).

43. Redox-active organic electrodes for pseudocapacitor applications. *ECS Meeting Abstracts* (2017) doi:10.1149/ma2017-01/1/98.

44. Organic redox active molecular engineer for non-aqueous redox flow battery. *ECS Meeting Abstracts* (2014) doi:10.1149/ma2014-03/2/7.

45. Ionic liquids-promoted utilization of redox-active organic materials for flow batteries. *ECS Meeting Abstracts* (2019) doi:10.1149/ma2019-04/5/242.

46. (Invited) redox-active organic electrode materials for safe energy storage. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-02/55/1969.

47. Redox-active organic molecules for non-aqueous flow batteries. *ECS Meeting Abstracts* (2012) doi:10.1149/ma2012-02/5/410.

48. (Keynote) redox-active organic species for rechargeable batteries, and beyond! *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-02/2/89.

49. Organic active species for nonaqueous redox flow batteries. *ECS Meeting Abstracts* (2015) doi:10.1149/ma2015-03/3/685.

50. Jang, S. S. (Invited) density functional theory modeling - machine learning approach for redox potential of organic electrode materials. *ECS Meeting Abstracts* **MA2020-02**, 199–199 (2020).

51. Phenothiazine-based redox polymers as cathode-active materials in li/organic batteries. *ECS Meeting Abstracts* (2019) doi:10.1149/ma2019-04/6/349.

52. Allam, O. *et al.* Molecular structure–redox potential relationship for organic electrode materials: Density functional theory–machine learning approach. *Materials Today Energy* **17**, 100482 (2020).

53. Wang, X., Chai, J. & Jiang, J. "Jimmy". Redox flow batteries based on insoluble redox-active materials. A review. *Nano Materials Science* **3**, 17–24 (2021).

54. Chen, R. Redox flow batteries for energy storage: Recent advances in using organic active materials. *Current Opinion in Electrochemistry* **21**, 40–45 (2020).

55. Wang, S., Li, F., Easley, A. D. & Lutkenhaus, J. L. Real-time insight into the doping mechanism of redox-active organic radical polymers. *Nature Materials* **18**, 69–75 (2018).

56. Highly stable active materials for nonaqueous redox flow batteries. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-02/1/68.

57. (Invited) computational discovery of metal-organic frameworks for hydrogen storage: Combining high-throughput screening, machine learning, and experimental demonstration. *ECS Meeting Abstracts* (2019) doi:10.1149/ma2019-02/42/1995.

58. Hybrid organic and inorganic redox active components for non-aqueous redox flow batteries. *ECS Meeting Abstracts* (2014) doi:10.1149/ma2014-01/4/396.

59. Organic anolyte species for aqueous redox flow batteries. *ECS Meeting Abstracts* (2017) doi:10.1149/ma2017-02/1/16.

60. Derivatives of naphthoquinone as potential electroactive species for redox FLOW batteries. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-02/2/124.

61. Multi-redox molecule for high-energy redox flow batteries. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-02/5/375.

62. Properties of redox couples for use in organic redox flow batteries. *ECS Meeting Abstracts* (2015) doi:10.1149/ma2015-01/3/704.

63. (Invited) organic molecules for redox flow batteries. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-03/4/234.

64. Developing new chemistries for redox flow batteries. *ECS Meeting Abstracts* (2019) doi:10.1149/ma2019-02/1/4.

65. Flexible ceramic membranes for redox flow batteries. *ECS Meeting Abstracts* (2018) doi:10.1149/ma2018-01/41/2373.

66. (Invited) high-energy-density redox-flow batteries: Fundamental redox processes and materials design strategies. *ECS Meeting Abstracts* (2019) doi:10.1149/ma2019-01/3/424.

67. Pan, F. & Wang, Q. Redox species of redox flow batteries: A review. *Molecules* **20**, 20499–20517 (2015).

68. Fink, T. & Reymond, J.-L. Virtual Exploration of the Chemical Universe up to 11 Atoms of C, N, O, F: Assembly of 26.4 Million Structures (110.9 Million Stereoisomers) and Analysis for New Ring Systems, Stereochemistry, Physicochemical Properties, Compound Classes, and Drug Discovery. *J. Chem. Inf. Model.* **47**, 342–353 (2007).

69. Goodfellow, I. J. *et al.* Generative Adversarial Networks. http://arxiv.org/abs/1406.2661 (2014).

70. Weininger, D. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.* **28**, 31–36 (1988).

71. G'omez-Bombarelli, R. & Aspuru-Guzik, A. Machine Learning and Big-Data in Computational Chemistry. *Handb. Mater. Model.* 1–24 (2018) doi:gjprg7.

72. G'omez-Bombarelli, R. *et al.* Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. *ACS Cent. Sci.* **4**, 268–276 (2018).