2020-01-15-brynmawr

Kelsey Gonzalez, Amelia McNamara, The Carpentries

2020-01-03

Contents

Preface

Chapter 1

Introduction

teaching: 25
exercises: 15
adapted from: https://datacarpentry.org/r-socialsci/00-intro/index.html

questions:

- How to find your way around RStudio?
- How to interact with R?
- How to manage your environment?
- How to install packages?

objectives:

- Install latest version of R.
- Install latest version of RStudio.
- Navigate the RStudio GUI.
- Install additional packages using the packages tab.
- Install additional packages using R code.

keypoints:

- Use RStudio to write and run R programs.
- Use `install.packages()` to install packages (libraries).

1.1 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 currently a very popular way to not only write your R scripts but also to interact with the R software. To function correctly, RStudio needs R and therefore both need to be installed on your computer.

To make it easier to interact with R, we will use RStudio. RStudio is the most popular IDE (Integrated Development Interface) for R. An IDE is a piece of software that provides tools to make programming easier.

1.2 Why learn R?

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

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

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

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

1.2.2 R code is great for reproducibility

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

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

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

1.2.3 R is interdisciplinary and extensible

With 10,000+ packages that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyze

your data. For instance, R has packages for image analysis, GIS, time series, population genetics, and a lot more.

1.2.4 R works on data of all shapes and sizes

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

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

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

1.2.5 R produces high-quality graphics

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

1.2.6 R has a large and welcoming community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow, or on the RStudio community. Questions which are backed up with short, reproducible code snippets are more likely to attract knowledgeable responses.

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

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

Because R is open source and is supported by a large community of developers and users, there is a very large selection of third-party add-on packages which are freely available to extend R's native capabilities.

RStudio extends what R can do, and makes it easier to write R code and interact with R

1.3 Knowing your way around RStudio

Let's start by learning about RStudio, which is an Integrated Development Environment (IDE) for working with R.

The RStudio IDE open-source product is free under the Affero General Public License (AGPL) v3. The RStudio IDE is also available with a commercial license and priority email support from RStudio, Inc.

We will use the RStudio IDE to write code, navigate the files on our computer, inspect the variables we create, and visualize the plots we generate. RStudio can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we will not cover during the workshop.

One of the advantages of using RStudio is that all the information you need to write code is available in a single window. Additionally, RStudio provides many shortcuts, autocompletion, and highlighting for the major file types you use while developing in R. RStudio makes typing easier and less error-prone.

1.4 Getting set up

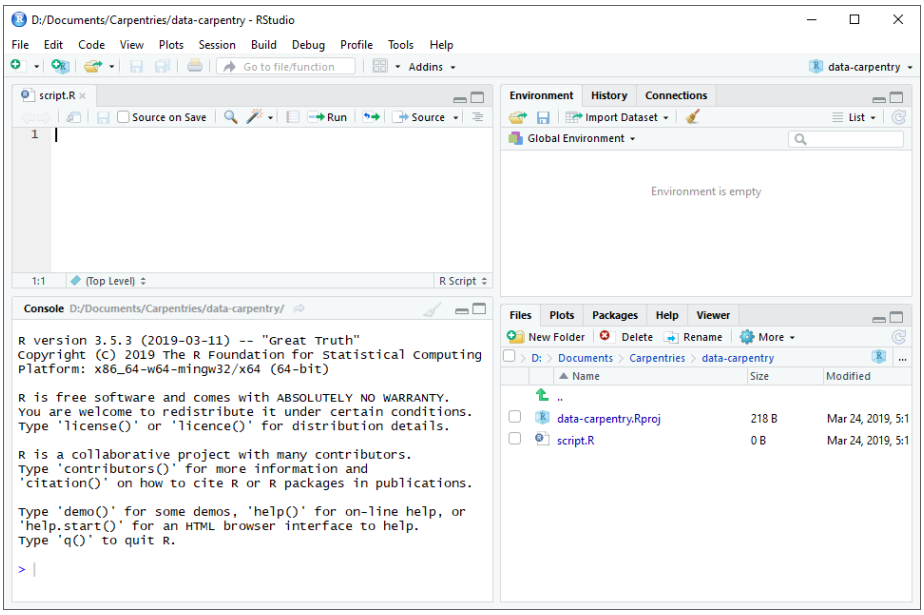
It is good practice to keep a set of related data, analyses, and text self-contained in a single folder called the **working directory**. All of the scripts within this folder can then use *relative paths* to files. Relative paths indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without having to directly modify file paths in the individual scripts.

RStudio provides a helpful set of tools to do this through its "Projects" interface, which not only creates a working directory for you but also remembers its location (allowing you to quickly navigate to it). The interface also (optionally) preserves custom settings and open files to make it easier to resume work after a break.

1.4.1 Create a new project

- Under the `File` menu, click on `New project`, choose `New directory`, then `New project`
- Enter a name for this new folder (or "directory") and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `~/data-carpentry`)
- Click on `Create project`
- Create a new file where we will type our scripts. Go to File > New File > R script. Click the save icon on your toolbar and save your script as "`script.R`".

1.4.2 The RStudio Interface



Let's take a quick tour of RStudio. RStudio is divided into four "panes". The placement of these panes and their content can be customized (see menu, Tools -> Global Options -> Pane Layout).

The Default Layout is: - Top Left - **Source**: your scripts and documents - Bottom Left - **Console**: what R would look and be like without RStudio - Top Right - **Enviornment/History**: look here to see what you have done - Bottom

Right - **Files** and more: see the contents of the project/working directory here, like your Script.R file

1.4.3 Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized and make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you might create directories (folders) for **scripts**, **data**, and **documents**. Here are some examples of suggested directories:

- `data/` Use this folder to store your raw data and intermediate datasets. For the sake of transparency and provenance, you should *always* keep a copy of your raw data accessible and do as much of your data cleanup and preprocessing programmatically (i.e., with scripts, rather than manually) as possible.
- `data_output/` When you need to modify your raw data, it might be useful to store the modified versions of the datasets in a different folder.
- `documents/` Used for outlines, drafts, and other text.
- `fig_output/` This folder can store the graphics that are generated by your scripts.
- `scripts/` A place to keep your R scripts for different analyses or plotting.

You may want additional directories or subdirectories depending on your project needs, but these should form the backbone of your working directory.

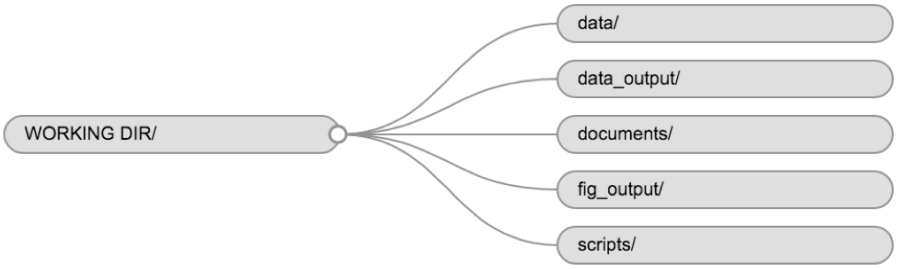


Figure 1.1: Example of a working directory structure

1.4.4 The working directory

The working directory is an important concept to understand. It is the place where R will look for and save files. When you write code for your project, your scripts should refer to files in relation to the root of your working directory and only to files within this structure.

Using RStudio projects makes this easy and ensures that your working

directory is set up properly. If you need to check it, you can use `getwd()`. If for some reason your working directory is not what it should be, you can change it in the RStudio interface by navigating in the file browser to where your working directory should be, clicking on the blue gear icon "More", and selecting "Set As Working Directory". Alternatively, you can use `setwd("/path/to/working/directory")` to reset your working directory. However, your scripts should not include this line, because it will fail on someone else's computer.

1.4.5 Downloading the data and getting set up

For this lesson we will use the following folders in our working directory: **`data/`**, **`data_output/`** and **`fig_output/`**. Let's write them all in lowercase to be consistent. We can create them using the RStudio interface by clicking on the "New Folder" button in the file pane (bottom right), or directly from R by typing at console:

```
dir.create("data")
dir.create("data_output")
dir.create("fig_output")
```

Go to the Figshare page for this curriculum and download the dataset called "`SAFI_clean.csv`". The direct download link is: https://ndownloader.figshare.com/files/11492171. Place this downloaded file in the **`data/`** you just created. You can do this directly from R by copying and pasting this in your terminal (your instructor can place this chunk of code in the Etherpad):

```
download.file("https://ndownloader.figshare.com/files/11492171",
              "data/SAFI_clean.csv", mode = "wb")
```

1.5 Interacting with R

The basis of programming is that we write down instructions for the computer to follow, and then we tell the computer to follow those instructions. We write, or *code*, instructions in R because it is a common language that both the computer and we can understand. We call the instructions *commands* and we tell the computer to follow the instructions by *executing* (also called *running*) those commands.

There are two main ways of interacting with R: by using the console or by using script files (plain text files that contain your code). The console pane (in RStudio, the bottom left panel) is the place where commands written in the R language can be typed and executed immediately by the computer. It is also where the results will be shown for commands that have been executed. You can type commands directly into the console and press Enter to execute those commands, but they will be forgotten when you close the session.

Because we want our code and workflow to be reproducible, it is better to type the commands we want in the script editor and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer.

RStudio allows you to execute commands directly from the script editor by using the Ctrl + Enter shortcut (on Mac, Cmd + Return will work). The command on the current line in the script (indicated by the cursor) or all of the commands in selected text will be sent to the console and executed when you press Ctrl + Enter. If there is information in the console you do not need anymore, you can clear it with Ctrl + L.

You can find other keyboard shortcuts in this RStudio cheatsheet about the RStudio IDE.

At some point in your analysis, you may want to check the content of a variable or the structure of an object without necessarily keeping a record of it in your script. You can type these commands and execute them directly in the console. RStudio provides the Ctrl + 1 and Ctrl + 2 shortcuts allow you to jump between the script and the console panes.

If R is ready to accept commands, the R console shows a > prompt. If R receives a command (by typing, copy-pasting, or sent from the script editor using Ctrl + Enter), R will try to execute it and, when ready, will show the results and come back with a new > prompt to wait for new commands.

If R is still waiting for you to enter more text, the console will show a + prompt. It means that you haven't finished entering a complete command. This is likely because you have not 'closed' a parenthesis or quotation, i.e. you don't have the same number of left-parentheses as right-parentheses or the same number of opening and closing quotation marks.

When this happens, and you thought you finished typing your command, click inside the console window and press Esc; this will cancel the incomplete command and return you to the > prompt. You can then proofread the command(s) you entered and correct the error.

1.6 Installing additional packages using the packages tab

In addition to the core R installation, there are in excess of 10,000 additional packages which can be used to extend the functionality of R. Many of these have been written by R users and have been made available in central repositories, like the one hosted at CRAN, for anyone to download and install into their own R environment. In the course of this lesson we will be making use of several of these packages, such as 'ggplot2' and 'dplyr'.

Additional packages can be installed from the 'packages' tab. On the packages

tab, click the 'Install' icon and start typing the name of the package you want in the text box. As you type, packages matching your starting characters will be displayed in a drop-down list so that you can select them.

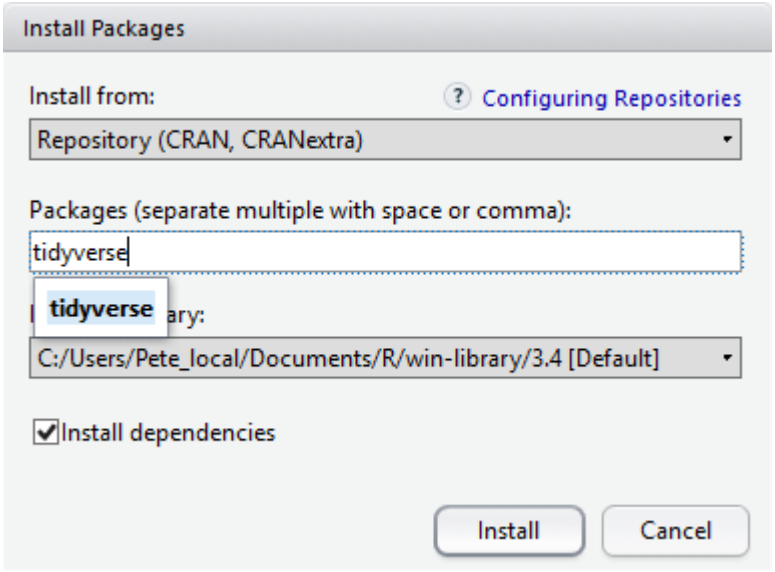


Figure 1.2: Install Packages Window

At the bottom of the Install Packages window is a check box to 'Install' dependencies. This is ticked by default, which is usually what you want. Packages can (and do) make use of functionality built into other packages, so for the functionality contained in the package you are installing to work properly, there may be other packages which have to be installed with them. The 'Install dependencies' option makes sure that this happens.

1.7 Exercise

Use the install option from the packages tab to install the 'tidyverse' package.

1.8 Solution

From the packages tab, click 'Install' from the toolbar and type 'tidyverse' into the textbox, then click 'install'. The 'tidyverse' package is really a package of packages, including 'ggplot2' and 'dplyr', both of which require other packages to run correctly. All of these packages will be installed automatically. Depending on what packages have

previously been installed in your R environment, the install of 'tidyverse' could be very quick or could take several minutes. As the install proceeds, messages relating to its progress will be written to the console. You will be able to see all of the packages which are actually being installed.
{: .solution} {: .challenge}

Because the install process accesses the CRAN repository, you will need an Internet connection to install packages.

It is also possible to install packages from other repositories, as well as Github or the local file system, but we won't be looking at these options in this lesson.

1.9 Installing additional packages using R code

If you were watching the console window when you started the install of 'tidyverse', you may have noticed that the line

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

was written to the console before the start of the installation messages.

You could also have installed the **tidyverse** packages by running this command directly at the R terminal.

Chapter 2

Project Management With RStudio

teaching: 20
exercises: 10
adapted from: http://swcarpentry.github.io/r-novice-gapminder/02-project-intro/index.html

questions:

- How can I manage my projects in R?

objectives:

- Create self-contained projects in RStudio

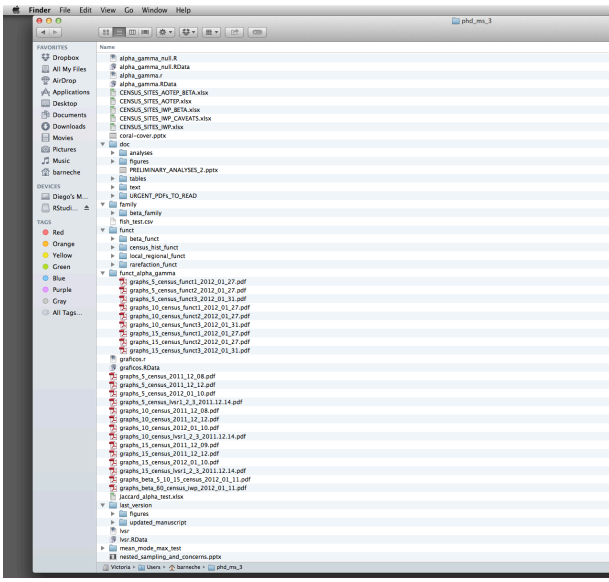
keypoints:

- Use RStudio to create and manage projects with consistent layout.
- Treat raw data as read-only.
- Treat generated output as disposable.
- Separate function definition and application.

2.1 Introduction

The scientific process is naturally incremental, and many projects start life as random notes, some code, then a manuscript, and eventually everything is a bit mixed together.

Most people tend to organize their projects like this:



There are many reasons why we should *ALWAYS* avoid this:

1. It is really hard to tell which version of your data is the original and which is the modified;
2. It gets really messy because it mixes files with various extensions together;
3. It probably takes you a lot of time to actually find things, and relate the correct figures to the exact code that has been used to generate it;

A good project layout will ultimately make your life easier:

- It will help ensure the integrity of your data;
- It makes it simpler to share your code with someone else (a lab-mate, collaborator, or supervisor);
- It allows you to easily upload your code with your manuscript submission;
- It makes it easier to pick the project back up after a break.

2.2 A possible solution

Fortunately, there are tools and packages which can help you manage your work effectively.

One of the most powerful and useful aspects of RStudio is its project management functionality. We'll be using this today to create a self-contained, reproducible project.

2.3 Challenge: Creating a self-contained project

We're going to create a new project in RStudio:

1. Click the "File" menu button, then "New Project".
2. Click "New Directory".
3. Click "New Project".
4. Type in the name of the directory to store your project, e.g. "my_project".
5. If available, select the checkbox for "Create a git repository."
6. Click the "Create Project" button. {: .challenge}

Now when we start R in this project directory, or open this project with RStudio, all of our work on this project will be entirely self-contained in this directory.

2.4 Best practices for project organization

Although there is no "best" way to lay out a project, there are some general principles to adhere to that will make project management easier:

2.4.1 Treat data as read only

This is probably the most important goal of setting up a project. Data is typically time consuming and/or expensive to collect. Working with them interactively (e.g., in Excel) where they can be modified means you are never sure of where the data came from, or how it has been modified since collection. It is therefore a good idea to treat your data as "read-only".

2.4.2 Data Cleaning

In many cases your data will be "dirty": it will need significant preprocessing to get into a format R (or any other programming language) will find useful. This task is sometimes called "data munging". Storing these scripts in a separate folder, and creating a second "read-only" data folder to hold the "cleaned" data sets can prevent confusion between the two sets.

2.4.3 Treat generated output as disposable

Anything generated by your scripts should be treated as disposable: it should all be able to be regenerated from your scripts.

There are lots of different ways to manage this output. Having an output folder with different sub-directories for each separate analysis makes it easier later. Since many analyses are exploratory and don't end up being used in the final project, and some of the analyses get shared between projects.

2.5 Tip: Good Enough Practices for Scientific Computing

Good Enough Practices for Scientific Computing gives the following recommendations for project organization:

1. Put each project in its own directory, which is named after the project.
2. Put text documents associated with the project in the `doc` directory.
3. Put raw data and metadata in the `data` directory, and files generated during cleanup and analysis in a `results` directory.
4. Put source for the project's scripts and programs in the `src` directory, and programs brought in from elsewhere or compiled locally in the `bin` directory.
5. Name all files to reflect their content or function.

{: .callout}

2.5.1 Separate function definition and application

One of the more effective ways to work with R is to start by writing the code you want to run directly in an .R script, and then running the selected lines (either using the keyboard shortcuts in RStudio or clicking the "Run" button) in the interactive R console.

When your project is in its early stages, the initial .R script file usually contains many lines of directly executed code. As it matures, reusable chunks get pulled into their own functions. It's a good idea to separate these functions into two separate folders; one to store useful functions that you'll reuse across analyses and projects, and one to store the analysis scripts.

2.6 Tip: avoiding duplication

You may find yourself using data or analysis scripts across several projects. Typically you want to avoid duplication to save space and avoid having to make updates to code in multiple places.

In this case, making "symbolic links", which are essentially shortcuts to files somewhere else on a filesystem, can let you use existing code without having to move or copy it. Plus, any changes made to that code will only have to be made once.

On Linux and OS X you can use the `ln -s` command, and on Windows you can either create a shortcut or use the `mklink` command from the windows terminal. {: .callout}

2.6.1 Save the data in the data directory

Now we have a good directory structure we will now place/save the data file in the `data/` directory.

2.7 Challenge 1

Download the SAFI (Studying African Farmer-Led Irrigation) data from here.

1. Download the file (CTRL + S, right mouse click -> "Save as", or File -> "Save page as")
2. Make sure it's saved under the name `SAFI_clean.csv`
3. Save the file in the `data/` folder within your project.

We will load and inspect these data later. {: .challenge}

2.8 Challenge 2

It is useful to get some general idea about the dataset, directly from the command line, before loading it into R. Understanding the dataset better will come in handy when making decisions on how to load it in R. Use the command-line shell to answer the following questions: 1. What is the size of the file? 2. How many rows of data does it contain? 3. What kinds of values are stored in this file?

2.9 Solution to Challenge 2

By running these commands in the shell:

```
ls -lh data/SAFI_clean.csv
```

```
## -rw-r--r-- 1 ckgon ckgon 22K Jan  3 17:01 data/SAFI_clean.csv
```

The file size is 80K.

```
wc -l data/SAFI_clean.csv
```

```
## 131 data/SAFI_clean.csv
```

There are 1705 lines. The data looks like:

```
head data/SAFI_clean.csv
```

```
## key_ID,village,interview_date,no_membrs,years_liv,respondent_wall_type,rooms,memb_ass
## 1,God,2016-11-17T00:00:00Z,3,4,muddaub,1,NULL,NULL,1,bicycle;television;solar_panel;t
## 1,God,2016-11-17T00:00:00Z,7,9, muddaub,1,yes,once,3,cow_cart;bicycle;radio;cow_ploug
## 3,God,2016-11-17T00:00:00Z,10,15, burntbricks,1,NULL,NULL,1,solar_torch,2,Jan;Feb;Mar
## 4,God,2016-11-17T00:00:00Z,7,6, burntbricks,1,NULL,NULL,2,bicycle;radio;cow_plough;so
```

```
## 5,God,2016-11-17T00:00:00Z,7,40,burntbricks,1,NULL,NULL,4,motorcyle;radio;
## 6,God,2016-11-17T00:00:00Z,3,3,muddaub,1,NULL,NULL,1,NULL,2,Aug;Sept;Oct,uu
## 7,God,2016-11-17T00:00:00Z,6,38,muddaub,1,no,never,1,motorcyle;cow_plough,3
## 8,Chirodzo,2016-11-16T00:00:00Z,12,70,burntbricks,3,yes,never,2,motorcyle;b
## 9,Chirodzo,2016-11-16T00:00:00Z,8,6,burntbricks,1,no,never,3,television;sol
```

{: .solution} {: .challenge}

2.10 Tip: command line in R Studio

You can quickly open up a shell in RStudio using the **Tools -> Shell...** menu item. {: .callout}

2.10.1 Version Control

It is important to use version control with projects. Go here for a good lesson which describes using Git with RStudio.

Chapter 3

Introduction to R

teaching: 50
exercises: 30
adapted from: https://datacarpentry.org/r-socialsci/01-intro-to-r/index.html

questions:

- What data types are available in R?
- What is an object?
- How can values be initially assigned to variables of different data types?
- What arithmetic and logical operators can be used?
- How can subsets be extracted from vectors and data frames?
- How does R treat missing values?
- How can we deal with missing values in R?

objectives:

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
- Assign values to objects in R.
- Learn how to name objects.
- Use comments to inform script.

- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.
- Inspect the content of vectors and manipulate their content.
- Subset and extract values from vectors.
- Analyze vectors with missing data.

keypoints:

- Access individual values by location using `[]`.
- Access arbitrary sets of data using `[c(...)]`.
- Use logical operations and logical vectors to access subsets of data.

3.1 Creating objects in R

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

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
area_hectares <- 1.0
```

`<-` 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.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid, but `x2` is). R is case sensitive (e.g., `age` is different from `Age`). There are some names that cannot be used because

they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within an object name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them. It is also recommended to use nouns for object names, and verbs for function names. It's important to be consistent in the styling of your code (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, three popular style guides are Google's, Jean Fan's and the tidyverse's. The tidyverse's is very comprehensive and may seem overwhelming at first. You can install the **`lintr`** package to automatically check for issues in the styling of your code.

3.2 Objects vs. variables

> What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects {: .callout}

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:

```
area_hectares <- 1.0    # doesn't print anything
(area_hectares <- 1.0)  # putting parenthesis around the call prints the value of `area_hectares`
```

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

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

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

Now that R has `area_hectares` in memory, we can do arithmetic with it. For instance, we may want to convert this area into acres (area in acres is 2.47 times the area in hectares):

```
2.47 * area_hectares
```

```
## [1] 2.47
```

We can also change an object's value by assigning it a new one:

```
area_hectares <- 2.5
2.47 * area_hectares
```

```
## [1] 6.175
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the plot's area in acres in a new object, `area_acres`:

```
area_acres <- 2.47 * area_hectares
```

and then change `area_hectares` to 50.

```
area_hectares <- 50
```

3.3 Exercise

What do you think is the current content of the object `area_acres`? 123.5 or 6.175?

3.4 Solution

The value of `area_acres` is still 6.175 because you have not re-run the line `area_acres <- 2.47 * area_hectares` since changing the value of `area_hectares`. {: .solution} {: .challenge}

3.5 Comments

All programming languages allow the programmer to include comments in their code. To do this in R we use the `#` character. Anything to the right of the `#` sign and up to the end of the line is treated as a comment and is ignored by R. You can start lines with comments or include them after any code on the line.

```
area_hectares <- 1.0            # land area in hectares
area_acres <- area_hectares * 2.47  # convert to acres
area_acres              # print land area in acres.
```

```
## [1] 2.47
```

RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard Ctrl + Shift + C. If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press Ctrl + Shift + C.

3.6 Exercise

Create two variables `length` and `width` and assign them values. It should be noted that R Studio might add "()" after width and if you leave the parentheses you will get unexpected results. This is why you might see other programmers abbreviate common words. Create a third variable `area` and give it a value based on the current values of `length` and `width`. Show that changing the values of either `length` and `width` does not affect the value of `area`.

3.7 Solution

```
length <- 2.5
width <- 3.2
area <- length * width
area
```

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

```
# change the values of length and width
length <- 7.0
width <- 6.5
# the value of area isn't changed
area
```

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

{: .solution} {: .challenge}

3.7.1 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
b <- sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even

a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

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

```
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```
args(round)
```

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

```
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to specify the names of all optional arguments. If you don't, someone reading your code might have to look up the

definition of a function with unfamiliar arguments to understand what you're doing.

3.8 Exercise

Type in `?round` at the console and then look at the output in the Help pane. What other functions exist that are similar to `round`? How do you use the `digits` parameter in the round function? {: .challenge}

3.9 Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of household members for the households we've interviewed and assign it to a new object `hh_members`:

```
hh_members <- c(3, 7, 10, 6)
hh_members
```

```
## [1]  3  7 10  6
```

A vector can also contain characters. For example, we can have a vector of the building material used to construct our interview respondents' walls (`respondent_wall_type`):

```
respondent_wall_type <- c("muddaub", "burntbricks", "sunbricks")
respondent_wall_type
```

```
## [1] "muddaub"     "burntbricks" "sunbricks"
```

The quotes around "muddaub", etc. are essential here. Without the quotes R will assume there are objects called `muddaub`, `burntbricks` and `sunbricks`. As these objects don't exist in R's memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(hh_members)
```

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

```
length(respondent_wall_type)
```

```
## [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(hh_members)
```

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

```
class(respondent_wall_type)
```

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

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(hh_members)
```

```
##  num [1:4] 3 7 10 6
```

```
str(respondent_wall_type)
```

```
##  chr [1:3] "muddaub" "burntbricks" "sunbricks"
```

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

```
possessions <- c("bicycle", "radio", "television")
possessions <- c(possessions, "mobile_phone") # add to the end of the vector
possessions <- c("car", possessions) # add to the beginning of the vector
possessions
```

```
## [1] "car"          "bicycle"      "radio"        "television"   "mobile_phone"
```

In the first line, we take the original vector `possessions`, add the value `"mobile_phone"` to the end of it, and save the result back into `possessions`. Then we add the value `"car"` to the beginning, again saving the result back into `possessions`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

An **atomic vector** is the simplest R **data type** and is a linear vector of a single type. Above, we saw 2 of the 6 main **atomic vector** types that R uses: `"character"` and `"numeric"` (or `"double"`). These are the basic building blocks that all R objects are built from. The other 4 **atomic vector** types are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` for bitstreams that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

3.10 Exercise

We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to mix these types in a single vector?

3.11 Solution

R implicitly converts them to all be the same type. {: .solution}

What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```
r  num_char <- c(1, 2, 3, "a")  num_logical <- c(1, 2, 3, TRUE)  char_logical <- c("a", "b", "c", TRUE)  tricky <- c(1, 2, 3, "4")
```

Why do you think it happens?

3.12 Solution

Vectors can be of only one data type. R tries to convert (coerce) the content of this vector to find a "common denominator" that doesn't lose any information. {: .solution}

How many values in `combined_logical` are `"TRUE"` (as a character) in the following example:

```
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
combined_logical <- c(num_logical, char_logical)
```

3.13 Solution

Only one. There is no memory of past data types, and the coercion happens the first time the vector is evaluated. Therefore, the `TRUE` in `num_logical` gets converted into a

`1` before it gets converted into `"1"` in `combined_logical`.
{: .solution}

You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

{: .challenge}

3.14 Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
respondent_wall_type <- c("muddaub", "burntbricks", "sunbricks")
respondent_wall_type[2]
```

```
## [1] "burntbricks"
```

```
respondent_wall_type[c(3, 2)]
```

```
## [1] "sunbricks"   "burntbricks"
```

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

```
more_respondent_wall_type <- respondent_wall_type[c(1, 2, 3, 2, 1, 3)]
more_respondent_wall_type
```

```
## [1] "muddaub"     "burntbricks" "sunbricks"   "burntbricks" "muddaub"
## [6] "sunbricks"
```

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.

3.14.1 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:

```
hh_members <- c(3, 7, 10, 6)
hh_members[c(TRUE, FALSE, TRUE, TRUE)]
```

```
## [1]  3 10  6
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 5:

```
hh_members > 5    # will return logicals with TRUE for the indices that meet the condition
```

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

```
## so we can use this to select only the values above 5
hh_members[hh_members > 5]
```

```
## [1]  7 10  6
```

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

```
hh_members[hh_members < 3 | hh_members > 5]
```

```
## [1]  7 10  6
```

```
hh_members[hh_members >= 7 & hh_members == 3]
```

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

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

A common task is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
possessions <- c("car", "bicycle", "radio", "television", "mobile_phone")
possessions[possessions == "car" | possessions == "bicycle"] # returns both car and bicycle
```

```
## [1] "car"     "bicycle"
```

```
possessions %in% c("car", "bicycle", "motorcycle", "truck", "boat")
```

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

```
possessions[possessions %in% c("car", "bicycle", "motorcycle", "truck", "boat")]
```

```
## [1] "car"     "bicycle"
```

3.15 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.

```
rooms <- c(2, 1, 1, NA, 4)
mean(rooms)
```

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

```
max(rooms)
```

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

```
mean(rooms, na.rm = TRUE)
```

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

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

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

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.
rooms[!is.na(rooms)]
```

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

```
## Returns the object with incomplete cases removed. The returned object is an atomic v
na.omit(rooms)
```

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

```
## Extract those elements which are complete cases. The returned object is an atomic ve
rooms[complete.cases(rooms)]
```

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

Recall that you can use the `typeof()` function to find the type of your atomic vector.

3.16 Exercise

1. Using this vector of rooms, create a new vector with the NAs removed.

```
rooms <- c(1, 2, 1, 1, NA, 3, 1, 3, 2, 1, 1, 8, 3, 1, NA, 1)
```

2. Use the function `median()` to calculate the median of the `rooms` vector.
3. Use R to figure out how many households in the set use more than 2 rooms for sleeping.

3.17 Solution

```
rooms <- c(1, 2, 1, 1, NA, 3, 1, 3, 2, 1, 1, 8, 3, 1, NA, 1)
rooms_no_na <- rooms[!is.na(rooms)]
# or
rooms_no_na <- na.omit(rooms)
# 2.
median(rooms, na.rm = TRUE)
```

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

```
# 3.
rooms_above_2 <- rooms_no_na[rooms_no_na > 2]
length(rooms_above_2)
```

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

{: .solution} {: .challenge}

Now that we have learned how to write scripts, and the basics of R's data structures, we are ready to start working with the SAFI dataset we have been using in the other lessons, and learn about data frames.

Chapter 4

Seeking Help

teaching: 10
exercises: 10
adapted from: http://swcarpentry.github.io/r-novice-gapminder/03-seeking-help/index.html
questions:

- "How can I get help in R?"

objectives:

- "To be able read R help files for functions and special operators."
- "To be able to use CRAN task views to identify packages to solve a problem."
- "To be able to seek help from your peers."

keypoints:

- "Use `help()` to get online help in R."

4.1 Reading Help files

R, and every package, provide help files for functions. The general syntax to search for help on any function, "function_name", from a specific function that is in a package loaded into your namespace (your interactive R session):

```
?function_name
help(function_name)
```

This will load up a help page in RStudio (or as plain text in R by itself).

Each help page is broken down into sections:

- Description: An extended description of what the function does.
- Usage: The arguments of the function and their default values.
- Arguments: An explanation of the data each argument is expecting.
- Details: Any important details to be aware of.
- Value: The data the function returns.
- See Also: Any related functions you might find useful.
- Examples: Some examples for how to use the function.

Different functions might have different sections, but these are the main ones you should be aware of.

4.2 Tip: Running Examples

From within the function help page, you can highlight code in the Examples and hit Ctrl+Return to run it in RStudio console. This is gives you a quick way to get a feel for how a function works.

4.3 Tip: Reading help files

One of the most daunting aspects of R is the large number of functions available. It would be prohibitive, if not impossible to remember the correct usage for every function you use. Luckily, the help files mean you don't have to! {: .callout}

4.4 Special Operators

To seek help on special operators, use quotes:

```
?"<-"
```

4.5 Getting help on packages

Many packages come with "vignettes": tutorials and extended example documentation. Without any arguments, `vignette()` will list all vignettes for all installed packages; `vignette(package="package-name")` will list all available vignettes for `package-name`, and `vignette("vignette-name")` will open the specified vignette.

If a package doesn't have any vignettes, you can usually find help by typing `help("package-name")`.

4.6 When you kind of remember the function

If you're not sure what package a function is in, or how it's specifically spelled you can do a fuzzy search:

```
??function_name
```

4.7 When you have no idea where to begin

If you don't know what function or package you need to use CRAN Task Views is a specially maintained list of packages grouped into fields. This can be a good starting point.

4.8 When your code doesn't work: seeking help from your peers

If you're having trouble using a function, 9 times out of 10, the answers you are seeking have already been answered on Stack Overflow. You can search using the `[r]` tag.

If you can't find the answer, there are a few useful functions to help you ask a question from your peers:

```
?dput
```

Will dump the data you're working with into a format so that it can be copy and pasted by anyone else into their R session.

```
sessionInfo()
```

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 18362)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
```

```
##  [1] lubridate_1.7.4 forcats_0.4.0   stringr_1.4.0   dplyr_0.8.3
##  [5] purrr_0.3.3     readr_1.3.1     tidyr_1.0.0     tibble_2.1.3
##  [9] ggplot2_3.2.1   tidyverse_1.3.0
##
## loaded via a namespace (and not attached):
##  [1] tidyselect_0.2.5 xfun_0.11        haven_2.2.0      lattice_0.20-38
##  [5] colorspace_1.4-1 vctrs_0.2.0      generics_0.0.2   htmltools_0.4.0
##  [9] yaml_2.2.0       utf8_1.1.4       rlang_0.4.2      pillar_1.4.2
## [13] withr_2.1.2      glue_1.3.1       DBI_1.1.0        dbplyr_1.4.2
## [17] modelr_0.1.5     readxl_1.3.1     lifecycle_0.1.0  munsell_0.5.0
## [21] gtable_0.3.0     cellranger_1.1.0 rvest_0.3.5      evaluate_0.14
## [25] labeling_0.3     knitr_1.26       fansi_0.4.0      broom_0.5.3
## [29] Rcpp_1.0.3       backports_1.1.5  scales_1.1.0     jsonlite_1.6
## [33] farver_2.0.1     fs_1.3.1         hms_0.5.2        digest_0.6.23
## [37] stringi_1.4.3    bookdown_0.16    grid_3.6.1       cli_2.0.0
## [41] tools_3.6.1      magrittr_1.5     lazyeval_0.2.2   crayon_1.3.4
## [45] pkgconfig_2.0.3  zeallot_0.1.0    ellipsis_0.3.0   xml2_1.2.2
## [49] reprex_0.3.0     assertthat_0.2.1 rmarkdown_2.0    httr_1.4.1
## [53] rstudioapi_0.10  R6_2.4.1         nlme_3.1-143     compiler_3.6.1
```

Will print out your current version of R, as well as any packages you have loaded. This can be useful for others to help reproduce and debug your issue.

4.9 Challenge 1

Look at the help for the `c` function. What kind of vector do you expect you will create if you evaluate the following:

```
c(1, 2, 3)
c('d', 'e', 'f')
c(1, 2, 'f')
```

4.10 Solution to Challenge 1

The `c()` function creates a vector, in which all elements are the same type. In the first case, the elements are numeric, in the second, they are characters, and in the third they are characters: the numeric values are "coerced" to be characters. {: .solution} {: .challenge}

4.11 Challenge 2

Look at the help for the `paste` function. You'll need to use this later. What is the difference between the `sep` and `collapse` arguments?

4.12 Solution to Challenge 2

To look at the help for the `paste()` function, use:

```
help("paste")
?paste
```

The difference between `sep` and `collapse` is a little tricky. The `paste` function accepts any number of arguments, each of which can be a vector of any length. The `sep` argument specifies the string used between concatenated terms — by default, a space. The result is a vector as long as the longest argument supplied to `paste`. In contrast, `collapse` specifies that after concatenation the elements are *collapsed* together using the given separator, the result being a single string. e.g.

```
paste(c("a","b"), "c")
```

```
## [1] "a c" "b c"
```

```
paste(c("a","b"), "c", sep = ",")
```

```
## [1] "a,c" "b,c"
```

```
paste(c("a","b"), "c", collapse = "|")
```

```
## [1] "a c|b c"
```

```
paste(c("a","b"), "c", sep = ",", collapse = "|")
```

```
## [1] "a,c|b,c"
```

(For more information, scroll to the bottom of the `?paste` help page and look at the examples, or try `example('paste')`.) {: .solution} {: .challenge}

4.13 Challenge 3

Use help to find a function (and its associated parameters) that you could use to load data from a tabular file in which columns are delimited with "⇳" (tab) and the decimal point is a "." (period). This check for decimal separator is important, especially if you are working with international colleagues, because different countries have different conventions for the decimal point (i.e. comma vs period). hint: use `??"read table"` to look up functions related to reading in tabular data. > ## Solution to Challenge 3 > > The standard R function for reading tab-delimited files with a period > decimal separator is read.delim(). You can also do this with >

`read.table(file, sep="\t")` (the period is the *default* decimal > separator for `read.table()`, although you may have to change > the `comment.char` argument as well if your data file contains > hash (#) characters {: .solution} {: .challenge}

4.14 Other ports of call

- Quick R
- RStudio cheat sheets
- Cookbook for R

Chapter 5

Starting with Data

teaching: 50
exercises: 30
adapted from: https://datacarpentry.org/r-socialsci/02-starting-with-data/index.html

questions:

- What is a data.frame?
- How can I read a complete csv file into R?
- How can I get basic summary information about my dataset?
- How can I change the way R treats strings in my dataset?
- Why would I want strings to be treated differently?
- How are dates represented in R and how can I change the format?

objectives:

- Describe what a data frame is.
- Load external data from a .csv file into a data frame.
- Summarize the contents of a data frame.
- Describe the difference between a factor and a string.
- Convert between strings and factors.

- Reorder and rename factors.
- Change how character strings are handled in a data frame.
- Examine and change date formats.

keypoints:

- Use read.csv to read tabular data in R.
- Use factors to represent categorical data in R.

5.1 Presentation of the SAFI Data

SAFI (Studying African Farmer-Led Irrigation) is a study looking at farming and irrigation methods in Tanzania and Mozambique. The survey data was collected through interviews conducted between November 2016 and June 2017. For this lesson, we will be using a subset of the available data. For information about the full teaching dataset used in other lessons in this workshop, see the dataset description.

We will be using a subset of the cleaned version of the dataset that was produced through cleaning in OpenRefine. Each row holds information for a single interview respondent, and the columns represent:

column_name	description
key_id	Added to provide a unique Id for each observation. (The InstanceID field does this as well but it is not as convenient to use)
village	Village name
interview_date	Date of interview

column_name	description
no_membrs	How many members in the household?
years_liv	How many years have you been living in this village or neighboring village?
respondent_wall_type	What type of walls does their house have (from list)
rooms	How many rooms in the main house are used for sleeping?
memb_assoc	Are you a member of an irrigation association?
affect_conflicts	Have you been affected by conflicts with other irrigators in the area?
liv_count	Number of livestock owned.
items_owned	Which of the following items are owned by the household? (list)

column_name	description
no_meals	How many meals do people in your household normally eat in a day?
months_lack_food	Indicate which months, In the last 12 months have you faced a situation when you did not have enough food to feed the household?
instanceID	Unique identifier for the form data submission

You are going load the data in R's memory using the function `read_csv()` from the **`readr`** package which is part of the **`tidyverse`**. So, before we can use the `read_csv()` function, we need to load the package. Also, if you recall, the missing data is encoded as "NULL" in the dataset. We'll tell it to the function, so R will automatically convert all the "NULL" entries in the dataset into `NA`.

```
library(tidyverse)
interviews <- read_csv("data/SAFI_clean.csv", na = "NULL")
```

This statement creates a data frame but doesn't show any data because, as you might recall, assignments don't display anything. (Note, however, that `read_csv` may show informational text about the data frame that is created.) If we want to check that our data has been loaded, we can see the contents of the data frame by typing its name: `interviews`.

```
interviews
## Try also
## View(interviews)
## head(interviews)
```

```
## # A tibble: 131 x 14
##    key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##     <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>            <dbl>
##  1      1 God     2016-11-17 00:00:00         3         4 muddaub              1
##  2      1 God     2016-11-17 00:00:00         7         9 muddaub              1
##  3      3 God     2016-11-17 00:00:00        10        15 burntbricks          1
##  4      4 God     2016-11-17 00:00:00         7         6 burntbricks          1
##  5      5 God     2016-11-17 00:00:00         7        40 burntbricks          1
##  6      6 God     2016-11-17 00:00:00         3         3 muddaub              1
##  7      7 God     2016-11-17 00:00:00         6        38 muddaub              1
##  8      8 Chirod… 2016-11-16 00:00:00        12        70 burntbricks          3
##  9      9 Chirod… 2016-11-16 00:00:00         8         6 burntbricks          1
## 10     10 Chirod… 2016-12-16 00:00:00        12        23 burntbricks          5
## # … with 121 more rows, and 7 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>
```

5.2 Note

`read_csv()` assumes that fields are delineated by commas, however, in several countries, the comma is used as a decimal separator and the semicolon (;) is used as a field delineator. If you want to read in this type of files in R, you can use the `read_csv2` function. It behaves exactly like `read_csv` but uses different parameters for the decimal and the field separators. If you are working with another format, they can be both specified by the user. Check out the help for `read_csv()` by typing `?read_csv` to learn more. There is also the `read_tsv()` for tab-separated data files, and `read_delim()` allows you to specify more details about the structure of your file. {: .callout}

5.3 What are data frames and tibbles?

Data frames are the *de facto* data structure for tabular data in `R`, and what we use for data processing, statistics, and plotting.

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.

A data frame can be created by hand, but most commonly they are generated by the functions `read_csv()` or `read_table()`; in other words, when importing spreadsheets from your hard drive (or the web).

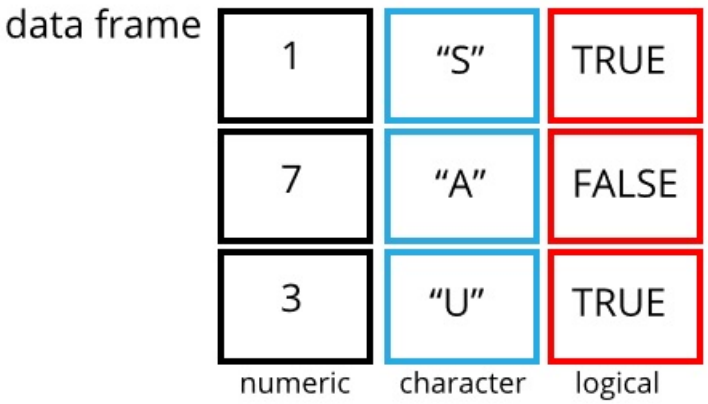


Figure 5.1: data frame example

A tibble is an extension of `R` data frames used by the *tidyverse*. When the data is read using `read_csv()`, it is stored in an object of class `tbl_df`, `tbl`, and `data.frame`. You can see the class of an object with

```
class(interviews)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

As a `tibble`, the type of data included in each column is listed in an abbreviated fashion below the column names. For instance, here `key_ID` is a column of integers (abbreviated `<int>`), `village` is a column of characters (`<chr>`) and the `interview_date` is a column in the "date and time" format (`<dttm>`).

5.4 Inspecting data frames

When calling a `tbl_df` object (like `interviews` here), there is already a lot of information about our data frame being displayed such as the number of rows, the number of columns, the names of the columns, and as we just saw the class of data stored in each column. However, there are functions to extract this information from data frames. Here is a non-exhaustive list of some of these functions. Let's try them out!

- Size:
- `dim(interviews)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
- `nrow(interviews)` - returns the number of rows

- `ncol(interviews)` - returns the number of columns
- Content:
- `head(interviews)` - shows the first 6 rows
- `tail(interviews)` - shows the last 6 rows
- Names:
- `names(interviews)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
- Summary:
- `str(interviews)` - structure of the object and information about the class, length and content of each column
- `summary(interviews)` - summary statistics for each column

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

5.5 Indexing and subsetting data frames

Our interviews 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)
interviews[1, 1]
```

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

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

```
## # A tibble: 1 x 1
##   respondent_wall_type
##   <chr>
## 1 muddaub
```

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

```
##   [1]   1   1   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
```

```
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  21  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 127
##  [73] 133 152 153 155 178 177 180 181 182 186 187 195 196 197 198 201 202  72
##  [91]  73  76  83  85  89 101 103 102  78  80 104 105 106 109 110 113 118 125
## [109] 119 115 108 116 117 144 143 150 159 160 165 166 167 174 175 189 191 192
## [127] 126 193 194 199 200
```

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

```
## # A tibble: 131 x 1
##    key_ID
##     <dbl>
##  1      1
##  2      1
##  3      3
##  4      4
##  5      5
##  6      6
##  7      7
##  8      8
##  9      9
## 10     10
## # … with 121 more rows
```

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

```
## # A tibble: 3 x 1
##   rooms
##   <dbl>
## 1     1
## 2     1
## 3     1
```

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

```
## # A tibble: 1 x 14
##   key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##    <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>            <dbl>
## 1      3 God     2016-11-17 00:00:00        10        15 burntbricks          1
## # … with 7 more variables: memb_assoc <chr>, affect_conflicts <chr>,
## #   liv_count <dbl>, items_owned <chr>, no_meals <dbl>, months_lack_food <chr>,
## #   instanceID <chr>
```

```
## equivalent to head_interviews <- head(interviews)
head_interviews <- interviews[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:

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

```
## # A tibble: 131 x 13
##    village interview_date      no_membrs years_liv respondent_wall… rooms
##    <chr>   <dttm>                  <dbl>     <dbl> <chr>            <dbl>
##  1 God     2016-11-17 00:00:00         3         4 muddaub              1
##  2 God     2016-11-17 00:00:00         7         9 muddaub              1
##  3 God     2016-11-17 00:00:00        10        15 burntbricks          1
##  4 God     2016-11-17 00:00:00         7         6 burntbricks          1
##  5 God     2016-11-17 00:00:00         7        40 burntbricks          1
##  6 God     2016-11-17 00:00:00         3         3 muddaub              1
##  7 God     2016-11-17 00:00:00         6        38 muddaub              1
##  8 Chirod… 2016-11-16 00:00:00        12        70 burntbricks          3
##  9 Chirod… 2016-11-16 00:00:00         8         6 burntbricks          1
## 10 Chirod… 2016-12-16 00:00:00        12        23 burntbricks          5
## # … with 121 more rows, and 7 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>
```

```
interviews[-c(7:131), ]   # Equivalent to head(interviews)
```

```
## # A tibble: 6 x 14
##   key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##    <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>            <dbl>
## 1      1 God     2016-11-17 00:00:00         3         4 muddaub              1
## 2      1 God     2016-11-17 00:00:00         7         9 muddaub              1
## 3      3 God     2016-11-17 00:00:00        10        15 burntbricks          1
## 4      4 God     2016-11-17 00:00:00         7         6 burntbricks          1
## 5      5 God     2016-11-17 00:00:00         7        40 burntbricks          1
## 6      6 God     2016-11-17 00:00:00         3         3 muddaub              1
## # … with 7 more variables: memb_assoc <chr>, affect_conflicts <chr>,
## #   liv_count <dbl>, items_owned <chr>, no_meals <dbl>, months_lack_food <chr>,
## #   instanceID <chr>
```

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

```
interviews["village"]       # Result is a data frame
interviews[, "village"]     # Result is a data frame
interviews[["village"]]     # Result is a vector
interviews$village          # Result is a vector
```

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

5.6 Exercise

1. Create a data frame (`interviews_100`) containing only the data in row 100 of the `interviews` dataset.
2. Notice how `nrow()` gave you the number of rows in a data frame?
   - Use that number to pull out just that last row in the data frame.
   - Compare that with what you see as the last row using `tail()` to make sure it's meeting expectations.
   - Pull out that last row using `nrow()` instead of the row number.
   - Create a new data frame (`interviews_last`) from that last row.
3. Use `nrow()` to extract the row that is in the middle of the data frame. Store the content of this row in an object named `interviews_middle`.
4. Combine `nrow()` with the `-` notation above to reproduce the behavior of `head(interviews)`, keeping just the first through 6th rows of the interviews dataset.

5.7 Solution

```
## 1.
interviews_100 <- interviews[100, ]
## 2.
# Saving `n_rows` to improve readability and reduce duplication
n_rows <- nrow(interviews)
interviews_last <- interviews[n_rows, ]
## 3.
interviews_middle <- interviews[(n_rows / 2), ]
## 4.
interviews_head <- interviews[-(7:n_rows), ]
```

{: .solution} {: .challenge}

5.8 Factors

R has a special data class, called factor, to deal with categorical data that you may encounter when creating plots or doing statistical analyses. 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:

```
respondent_floor_type <- factor(c("earth", "cement", "cement", "earth"))
```

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

```
levels(respondent_floor_type)
```

```
## [1] "cement" "earth"
```

```
nlevels(respondent_floor_type)
```

```
## [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 `respondent_floor_type` vector would be:

```
respondent_floor_type # current order
```

```
## [1] earth  cement cement earth
## Levels: cement earth
```

```
respondent_floor_type <- factor(respondent_floor_type, levels = c("earth", "cement"))
respondent_floor_type # after re-ordering
```

```
## [1] earth  cement cement earth
## Levels: earth cement
```

In R's memory, these factors are represented by integers (1, 2), but are more informative than integers because factors are self describing: `"cement"`, `"earth"` is more descriptive than `1`, and `2`. Which one is "earth"? 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. It also makes renaming levels easier. Let's say we made a mistake and need to recode "cement" to "brick".

```
levels(respondent_floor_type)
```

```
## [1] "earth"  "cement"
```

```
levels(respondent_floor_type)[2] <- "brick"
levels(respondent_floor_type)
```

```
## [1] "earth" "brick"
```

```
respondent_floor_type
```

```
## [1] earth brick brick earth
## Levels: earth brick
```

5.8.1 Converting factors

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

```
as.character(respondent_floor_type)
```

```
## [1] "earth" "brick" "brick" "earth"
```

Converting factors where the levels appear as numbers (such as concentration levels, or years) to a numeric vector 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 recommended `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

5.8.2 Renaming factors

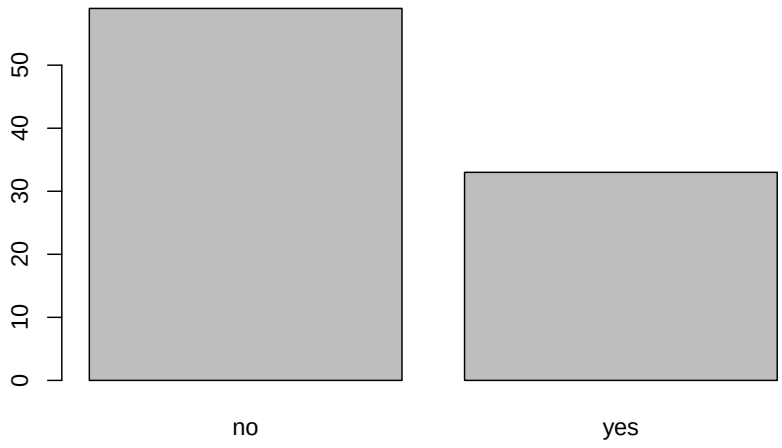
When your data is stored as a factor, you can use the `plot()` function to get a quick glance at the number of observations represented by each factor level.

Let's extract the `memb_assoc` column from our data frame, convert it into a factor, and use it to look at the number of interview respondents who were or were not members of an irrigation association:

```
## create a vector from the data frame column "memb_assoc"
memb_assoc <- interviews$memb_assoc
## convert it into a factor
memb_assoc <- as.factor(memb_assoc)
## let's see what it looks like
memb_assoc
```

```
##   [1] <NA> yes  <NA> <NA> <NA> <NA> no   yes  no   no   <NA> yes  no   <NA> yes
##  [16] <NA> <NA> <NA> <NA> <NA> no   <NA> <NA> no   no   no   <NA> no   yes  <NA>
##  [31] <NA> yes  no   yes  yes  yes  <NA> yes  <NA> yes  <NA> no   no   <NA> no
##  [46] no   yes  <NA> <NA> yes  <NA> no   yes  no   <NA> yes  no   no   <NA> no
##  [61] yes  <NA> <NA> <NA> no   yes  no   no   no   no   yes  <NA> no   yes  <NA>
##  [76] <NA> yes  no   no   yes  no   no   yes  no   yes  no   no   <NA> yes  yes
##  [91] yes  yes  yes  no   no   no   no   yes  no   no   yes  yes  no   <NA> no
## [106] no   <NA> no   no   <NA> no   <NA> <NA> no   no   no   no   yes  no   no
## [121] no   no   no   no   no   no   no   no   no   yes  <NA>
## Levels: no yes
```

```
## bar plot of the number of interview respondents who were
## members of irrigation association:
plot(memb_assoc)
```



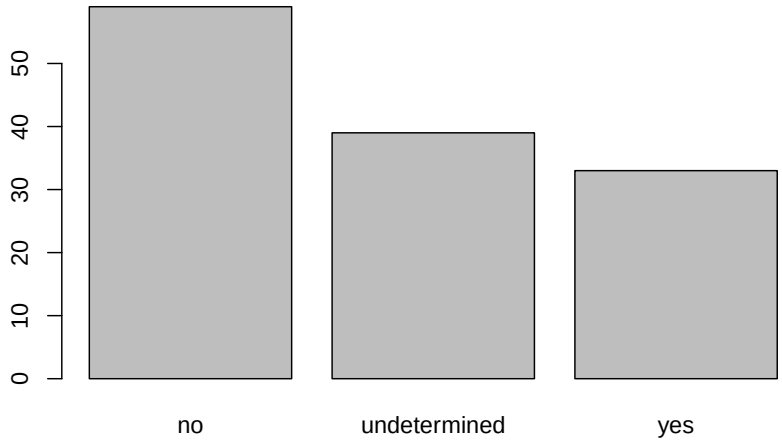
Looking at the plot compared to the output of the vector, we can see that n

addition to "no"s and "yes"s, there are about some respondents for which the information about whether they were part of an irrigation association hasn't been recorded, and encoded as missing data. They do not appear on the plot. Let's encode them differently so they can counted and visualized in our plot.

```
## Let's recreate the vector from the data frame column "memb_assoc"
memb_assoc <- interviews$memb_assoc
## replace the missing data with "undetermined"
memb_assoc[is.na(memb_assoc)] <- "undetermined"
## convert it into a factor
memb_assoc <- as.factor(memb_assoc)
## let's see what it looks like
memb_assoc
```

```
##   [1] undetermined yes          undetermined undetermined undetermined
##   [6] undetermined no           yes          no           no
##  [11] undetermined yes          no           undetermined yes
##  [16] undetermined undetermined undetermined undetermined undetermined
##  [21] no           undetermined undetermined no           no
##  [26] no           undetermined no           yes          undetermined
##  [31] undetermined yes          no           yes          yes
##  [36] yes          undetermined yes          undetermined yes
##  [41] undetermined no           no           undetermined no
##  [46] no           yes          undetermined undetermined yes
##  [51] undetermined no           yes          no           undetermined
##  [56] yes          no           no           undetermined no
##  [61] yes          undetermined undetermined undetermined no
##  [66] yes          no           no           no           no
##  [71] yes          undetermined no           yes          undetermined
##  [76] undetermined yes          no           no           yes
##  [81] no           no           yes          no           yes
##  [86] no           no           undetermined yes          yes
##  [91] yes          yes          yes          no           no
##  [96] no           no           yes          no           no
## [101] yes          yes          no           undetermined no
## [106] no           undetermined no           no           undetermined
## [111] no           undetermined undetermined no           no
## [116] no           no           yes          no           no
## [121] no           no           no           no           no
## [126] no           no           no           no           yes
## [131] undetermined
## Levels: no undetermined yes
```

```
## bar plot of the number of interview respondents who were
## members of irrigation association:
plot(memb_assoc)
```



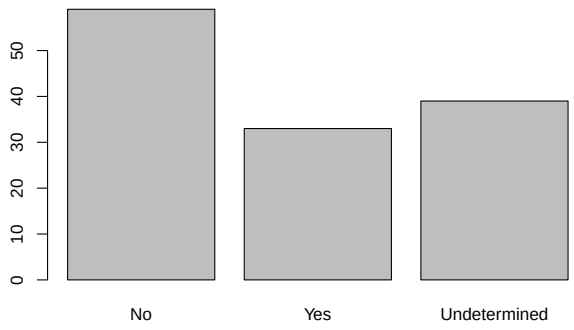
5.9 Exercise

- Rename the levels of the factor to have the first letter in uppercase: “No”,“Undetermined”, and “Yes”.

- Now that we have renamed the factor level to “Undetermined”, can you recreate the barplot such that “Undetermined” is last (after “Yes”)?

5.10 Solution

```
levels(memb_assoc) <- c("No", "Undetermined", "Yes")
memb_assoc <- factor(memb_assoc, levels = c("No", "Yes", "Undetermined"))
plot(memb_assoc)
```



{: .solution} {: .challenge}

5.11 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. In our dataset, we have a column `interview_date` which contains information about the year, month, and day that the interview was conducted. Let's convert those dates into three separate columns.

```
str(interviews)
```

We are going to use the package **`lubridate`** (which belongs to the **`tidyverse`**; learn more here) to work with dates. **`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)
```

The lubridate function `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 extract our `interview_date` column and inspect the structure:

```
dates <- interviews$interview_date
str(dates)
```

```
##  POSIXct[1:131], format: "2016-11-17" "2016-11-17" "2016-11-17" "2016-11-17" "2016-11-17" ...
```

When we imported the data in R, `read_csv()` recognized that this column contained date information. We can now use the `day()`, `month()` and `year()` functions to extract this information from the date, and create new columns in our data frame to store it:

```
interviews$day <- day(dates)
interviews$month <- month(dates)
interviews$year <- year(dates)
interviews
```

```
## # A tibble: 131 x 17
##    key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##     <dbl> <chr>   <dttm>                   <dbl>     <dbl> <chr>            <dbl>
##  1      1 God     2016-11-17 00:00:00          3         4 muddaub              1
##  2      1 God     2016-11-17 00:00:00          7         9 muddaub              1
##  3      3 God     2016-11-17 00:00:00         10        15 burntbricks          1
##  4      4 God     2016-11-17 00:00:00          7         6 burntbricks          1
##  5      5 God     2016-11-17 00:00:00          7        40 burntbricks          1
##  6      6 God     2016-11-17 00:00:00          3         3 muddaub              1
##  7      7 God     2016-11-17 00:00:00          6        38 muddaub              1
##  8      8 Chirod… 2016-11-16 00:00:00         12        70 burntbricks          3
##  9      9 Chirod… 2016-11-16 00:00:00          8         6 burntbricks          1
## 10     10 Chirod… 2016-12-16 00:00:00         12        23 burntbricks          5
## # … with 121 more rows, and 10 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>, day <int>, month <dbl>,
## #   year <dbl>
```

Notice the three new columns at the end of our data frame.

Chapter 6

Data Wrangling with `dplyr`

teaching: 50
exercises: 30
adapted from: https://datacarpentry.org/r-socialsci/03-dplyr-tidyr/index.html

questions:

- How can I select specific rows and/or columns from a data frame?
- How can I combine multiple commands into a single command?
- How can create new columns or remove existing columns from a data frame?
- How can I reformat a dataframe to meet my needs?

objectives:

- Describe the purpose of an R package and the **`dplyr`** and **`tidyr`** packages.
- Select certain columns in a data frame with the **`dplyr`** function `select`.
- Select certain rows in a data frame according to filtering conditions with the **`dplyr`** function `filter`.
- Link the output of one **`dplyr`** function to the input of another function with the 'pipe' operator `%>%`.
- Add new columns to a data frame that are functions of existing columns with `mutate`.

- Use the split-apply-combine concept for data analysis.
- Use `summarize`, `group_by`, and `count` to split a data frame into groups of observations, apply a summary statistics for each group, and then combine the results.
- Describe the concept of a wide and a long table format and for which purpose those formats are useful.
- Describe what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `spread` and `gather` commands from the **`tidyr`** package.
- Export a data frame to a csv file.

keypoints:

- Use the `dplyr` package to manipulate dataframes.
- Use `select()` to choose variables from a dataframe.
- Use `filter()` to choose data based on values.
- Use `group_by()` and `summarize()` to work with subsets of data.
- Use `mutate()` to create new variables.
- Use the `tidyr` package to change the layout of dataframes.
- Use `gather()` to go from wide to long format.
- Use `spread()` to go from long to wide format.

6.1 Data Manipulation using `dplyr` and `tidyr`

`dplyr` is a package for making tabular data manipulation easier by using a limited set of functions that can be combined to extract and summarize insights from your data. It pairs nicely with **`tidyr`** which enables you to swiftly convert between different data formats (long vs. wide) for plotting and analysis.

Similarly to **`readr`**, **`dplyr`** and **`tidyr`** are also part of the tidyverse. These packages were loaded in R's memory when we called `library(tidyverse)` earlier.

6.2 What is an R package?

An R package is a complete unit for sharing code with others. Each R package contains the code for a set of R functions, the documentation (or description) for each of the functions, as well as a practice dataset to learn the functions on.

Generally, each R package is built with a specific task in mind. For instance, 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++) (not all R packages are written in R!).

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. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

But there are also packages available for a wide range of tasks including building plots (**ggplot2**, which we'll see later), downloading data from the NCBI database, or performing statistical analysis on your data set. Many packages such as these are housed on, and downloadable from, the **C**omprehensive **R** **A**rchive **N**etwork (CRAN) using `install.packages`. This function makes the package accessible by your R installation with the command `library()`, as you did with `tidyverse` earlier.

To easily access the documentation for a package within R or RStudio, use `help(package = "package_name")`.

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**.

6.3 Learning `dplyr` and `tidyr`

To make sure, everyone will use the same dataset for this lesson, we'll read again the SAFI dataset that we downloaded earlier.

```
## load the tidyverse
library(tidyverse)

interviews <- read_csv("data/SAFI_clean.csv", na = "NULL")

## inspect the data
interviews

## preview the data
```

```
# View(interviews)
```

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 statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

6.4 Selecting columns and filtering rows

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

```
select(interviews, village, no_membrs, years_liv)
```

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

```
filter(interviews, village == "God")
```

```
## # A tibble: 43 x 14
##    key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##     <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>             <dbl>
##  1      1 God     2016-11-17 00:00:00         3         4 muddaub               1
##  2      1 God     2016-11-17 00:00:00         7         9 muddaub               1
##  3      3 God     2016-11-17 00:00:00        10        15 burntbricks           1
##  4      4 God     2016-11-17 00:00:00         7         6 burntbricks           1
##  5      5 God     2016-11-17 00:00:00         7        40 burntbricks           1
##  6      6 God     2016-11-17 00:00:00         3         3 muddaub               1
##  7      7 God     2016-11-17 00:00:00         6        38 muddaub               1
##  8     11 God     2016-11-21 00:00:00         6        20 sunbricks             1
##  9     12 God     2016-11-21 00:00:00         7        20 burntbricks           3
## 10     13 God     2016-11-21 00:00:00         6         8 burntbricks           1
## # … with 33 more rows, and 7 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>
```

6.5 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:

```
interviews2 <- filter(interviews, village == "God")
interviews_god <- select(interviews2, no_membrs, years_liv)
```

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:

```
interviews_god <- select(filter(interviews, village == "God"), no_membrs, years_liv)
```

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.

```
interviews %>%
    filter(village == "God") %>%
    select(no_membrs, years_liv)
```

```
## # A tibble: 43 x 2
##    no_membrs years_liv
##        <dbl>     <dbl>
##  1         3         4
##  2         7         9
##  3        10        15
##  4         7         6
##  5         7        40
##  6         3         3
##  7         6        38
##  8         6        20
##  9         7        20
## 10         6         8
## # … with 33 more rows
```

In the above code, we use the pipe to send the `interviews` dataset first through `filter()` to keep rows where `village` is "God", then through `select()` to keep only the `no_membrs` and `years_liv` 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 take the data frame `interviews`, *then* we `filter` for rows with `village == "God"`, *then* we `select` columns `no_membrs` and `years_liv`. 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:

```
interviews_god <- interviews %>%
    filter(village == "God") %>%
    select(no_membrs, years_liv)

interviews_god
```

```
## # A tibble: 43 x 2
##    no_membrs years_liv
##        <dbl>     <dbl>
##  1         3         4
##  2         7         9
##  3        10        15
##  4         7         6
##  5         7        40
##  6         3         3
##  7         6        38
##  8         6        20
##  9         7        20
## 10         6         8
## # … with 33 more rows
```

Note that the final data frame (`interviews_god`) is the leftmost part of this expression.

6.6 Exercise

Using pipes, subset the `interviews` data to include interviews where respondents were members of an irrigation association (`memb_assoc`) and retain only the columns `affect_conflicts`, `liv_count`, and `no_meals`.

6.7 Solution

```
interviews %>%
    filter(memb_assoc == "yes") %>%
    select(affect_conflicts, liv_count, no_meals)
```

```
## # A tibble: 33 x 3
##    affect_conflicts liv_count no_meals
##    <chr>                <dbl>    <dbl>
##  1 once                     3        2
##  2 never                    2        2
##  3 never                    2        3
##  4 once                     3        2
##  5 frequently               1        3
##  6 more_once                5        2
##  7 more_once                3        2
##  8 more_once                2        3
##  9 once                     3        3
## 10 never                    3        3
## # … with 23 more rows
```

{: .solution} {: .challenge}

6.7.1 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()`.

We might be interested in the ratio of number of household members to rooms used for sleeping (i.e. avg number of people per room):

```
interviews %>%
    mutate(people_per_room = no_membrs / rooms)
```

```
## # A tibble: 131 x 15
##    key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##     <dbl> <chr>   <dttm>                   <dbl>     <dbl> <chr>            <dbl>
##  1      1 God     2016-11-17 00:00:00          3         4 muddaub              1
##  2      1 God     2016-11-17 00:00:00          7         9 muddaub              1
##  3      3 God     2016-11-17 00:00:00         10        15 burntbricks          1
##  4      4 God     2016-11-17 00:00:00          7         6 burntbricks          1
##  5      5 God     2016-11-17 00:00:00          7        40 burntbricks          1
##  6      6 God     2016-11-17 00:00:00          3         3 muddaub              1
##  7      7 God     2016-11-17 00:00:00          6        38 muddaub              1
##  8      8 Chirod… 2016-11-16 00:00:00         12        70 burntbricks          3
##  9      9 Chirod… 2016-11-16 00:00:00          8         6 burntbricks          1
## 10     10 Chirod… 2016-12-16 00:00:00         12        23 burntbricks          5
## # … with 121 more rows, and 8 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>, people_per_room <dbl>
```

We may be interested in investigating whether being a member of an irrigation association had any effect on the ratio of household members to rooms. To

look at this relationship, we will first remove data from our dataset where the respondent didn't answer the question of whether they were a member of an irrigation association. These cases are recorded as "NULL" in the dataset.

To remove these cases, we could insert a `filter()` in the chain:

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    mutate(people_per_room = no_membrs / rooms)
```

```
## # A tibble: 92 x 15
##    key_ID village interview_date      no_membrs years_liv respondent_wall… rooms
##     <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>            <dbl>
##  1      1 God     2016-11-17 00:00:00         7         9 muddaub              1
##  2      7 God     2016-11-17 00:00:00         6        38 muddaub              1
##  3      8 Chirod… 2016-11-16 00:00:00        12        70 burntbricks          3
##  4      9 Chirod… 2016-11-16 00:00:00         8         6 burntbricks          1
##  5     10 Chirod… 2016-12-16 00:00:00        12        23 burntbricks          5
##  6     12 God     2016-11-21 00:00:00         7        20 burntbricks          3
##  7     13 God     2016-11-21 00:00:00         6         8 burntbricks          1
##  8     15 God     2016-11-21 00:00:00         5        30 sunbricks            2
##  9     21 God     2016-11-21 00:00:00         8        20 burntbricks          1
## 10     24 Ruaca   2016-11-21 00:00:00         6         4 burntbricks          2
## # … with 82 more rows, and 8 more variables: memb_assoc <chr>,
## #   affect_conflicts <chr>, liv_count <dbl>, items_owned <chr>, no_meals <dbl>,
## #   months_lack_food <chr>, instanceID <chr>, people_per_room <dbl>
```

The `!` symbol negates the result, so we're asking for every row where `memb_assoc` *is not* missing..

6.8 Exercise

Create a new data frame from the `interviews` data that meets the following criteria: contains only the `village` column and a new column called `total_meals` containing a value that is equal to the total number of meals served in the household per day on average (`no_membrs` times `no_meals`). Only the rows where `total_meals` is greater than 20 should be shown in the final data frame.

Hint: think about how the commands should be ordered to produce this data frame!

6.9 Solution

```
interviews_total_meals <- interviews %>%
    mutate(total_meals = no_membrs * no_meals) %>%
```

```
    filter(total_meals > 20) %>%
    select(village, total_meals)
```

{: .solution} {: .challenge}

6.9.1 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.

6.9.1.1 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 average household size by village:

```
interviews %>%
    group_by(village) %>%
    summarize(mean_no_membrs = mean(no_membrs))
```

```
## # A tibble: 3 x 2
##   village  mean_no_membrs
##   <chr>             <dbl>
## 1 Chirodzo           7.08
## 2 God                6.86
## 3 Ruaca              7.57
```

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:

```
interviews %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs))
```

```
## # A tibble: 9 x 3
## # Groups:   village [3]
##   village  memb_assoc mean_no_membrs
##   <chr>    <chr>               <dbl>
## 1 Chirodzo no                   8.06
## 2 Chirodzo yes                  7.82
## 3 Chirodzo <NA>                 5.08
```

```
## 4 God      no                  7.13
## 5 God      yes                 8
## 6 God      <NA>                6
## 7 Ruaca    no                  7.18
## 8 Ruaca    yes                 9.5
## 9 Ruaca    <NA>                6.22
```

When grouping both by `village` and `membr_assoc`, we see rows in our table for respondents who did not specify whether they were a member of an irrigation association. We can exclude those data from our table using a filter step.

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs))
```

```
## # A tibble: 6 x 3
## # Groups:   village [3]
##   village  memb_assoc mean_no_membrs
##   <chr>    <chr>               <dbl>
## 1 Chirodzo no                   8.06
## 2 Chirodzo yes                  7.82
## 3 God      no                   7.13
## 4 God      yes                  8
## 5 Ruaca    no                   7.18
## 6 Ruaca    yes                  9.5
```

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 household size for each village for each group (members of an irrigation association vs not):

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs),
              min_membrs = min(no_membrs))
```

```
## # A tibble: 6 x 4
## # Groups:   village [3]
##   village  memb_assoc mean_no_membrs min_membrs
##   <chr>    <chr>               <dbl>      <dbl>
## 1 Chirodzo no                   8.06          4
## 2 Chirodzo yes                  7.82          2
## 3 God      no                   7.13          3
## 4 God      yes                  8             5
## 5 Ruaca    no                   7.18          2
## 6 Ruaca    yes                  9.5           5
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_membrs` to put the group with the smallest household first:

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs), min_membrs = min(no_membrs)) %>%
    arrange(min_membrs)
```

```
## # A tibble: 6 x 4
## # Groups:   village [3]
##   village  memb_assoc mean_no_membrs min_membrs
##   <chr>    <chr>               <dbl>      <dbl>
## 1 Chirodzo yes                  7.82          2
## 2 Ruaca    no                   7.18          2
## 3 God      no                   7.13          3
## 4 Chirodzo no                   8.06          4
## 5 God      yes                  8             5
## 6 Ruaca    yes                  9.5           5
```

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

```
interviews %>%
    filter(!is.na(memb_assoc)) %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs),
              min_membrs = min(no_membrs)) %>%
    arrange(desc(min_membrs))
```

```
## # A tibble: 6 x 4
## # Groups:   village [3]
##   village  memb_assoc mean_no_membrs min_membrs
##   <chr>    <chr>               <dbl>      <dbl>
## 1 God      yes                  8             5
## 2 Ruaca    yes                  9.5           5
## 3 Chirodzo no                   8.06          4
## 4 God      no                   7.13          3
## 5 Chirodzo yes                  7.82          2
## 6 Ruaca    no                   7.18          2
```

6.9.1.2 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 village, we would do:

```
interviews %>%
    count(village)
```

```
## # A tibble: 3 x 2
##   village      n
##   <chr>    <int>
## 1 Chirodzo    39
## 2 God         43
## 3 Ruaca       49
```

For convenience, `count()` provides the `sort` argument to get results in decreasing order:

```
interviews %>%
    count(village, sort = TRUE)
```

```
## # A tibble: 3 x 2
##   village      n
##   <chr>    <int>
## 1 Ruaca       49
## 2 God         43
## 3 Chirodzo    39
```

6.10 Exercise

1. How many households in the survey have an average of two meals per day? Three meals per day? Are there any other numbers of meals represented?

6.11 Solution

```
interviews %>%
   count(no_meals)
```

```
## # A tibble: 2 x 2
##   no_meals     n
##      <dbl> <int>
## 1        2    52
## 2        3    79
```

{: .solution}

2. Use `group_by()` and `summarize()` to find the mean, min, and max number of household members for each village. Also add the number of observations (hint: see `?n`).

6.12 Solution

```
interviews %>%
  group_by(village) %>%
  summarize(
      mean_no_membrs = mean(no_membrs),
      min_no_membrs = min(no_membrs),
      max_no_membrs = max(no_membrs),
      n = n()
  )
```

```
## # A tibble: 3 x 5
##   village  mean_no_membrs min_no_membrs max_no_membrs     n
##   <chr>             <dbl>         <dbl>         <dbl> <int>
## 1 Chirodzo           7.08             2            12    39
## 2 God                6.86             3            15    43
## 3 Ruaca              7.57             2            19    49
```

{: .solution}

3. What was the largest household interviewed in each month?

6.13 Solution

```
# if not already included, add month, year, and day columns
library(lubridate) # load lubridate if not already loaded
interviews %>%
    mutate(month = month(interview_date),
           day = day(interview_date),
           year = year(interview_date)) %>%
    group_by(year, month) %>%
    summarize(max_no_membrs = max(no_membrs))
```

```
## # A tibble: 5 x 3
## # Groups:   year [2]
##    year month max_no_membrs
##   <dbl> <dbl>         <dbl>
## 1  2016    11            19
## 2  2016    12            12
## 3  2017     4            17
## 4  2017     5            15
## 5  2017     6            15
```

{: .solution} {: .challenge}

6.14 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 `interviews`, each row contains the values of variables associated with each record (the unit), values such as the number of household members or posessions associated with each record. What if instead of comparing records, we wanted to look at differences in households grouped by different types of housing construction materials?

We'd need to create a new table where each row (the unit) is comprised of values of variables associated with each housing material (e.g. for `respondent_wall_type`). In practical terms this means the values of the wall construction materials in `respondent_wall_type` would become the names of column variables and the cells would contain `TRUE` or `FALSE`.

Having created a new table, we can now explore the relationship within and between household types - for example we could compare the ratio of household members to sleeping rooms grouped by type of construction material. 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.

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()`.

6.14.0.1 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 interviews to create new columns for each type of wall construction material. We use the pipe as before too. Because both the `key` and `value` parameters must come from column values, we will create a dummy column (we'll name it `wall_type_logical`) to hold the value `TRUE`,

which we will then place into the appropriate column that corresponds to the wall construction material for that respondent. When using `mutate()` if you give a single value, it will be used for all observations in the dataset. We will use `fill = FALSE` in `spread()` to fill the rest of the new columns for that row with `FALSE`.

```
interviews_spread <- interviews %>%
    mutate(wall_type_logical = TRUE) %>%
    spread(key = respondent_wall_type, value = wall_type_logical, fill = FALSE)
```

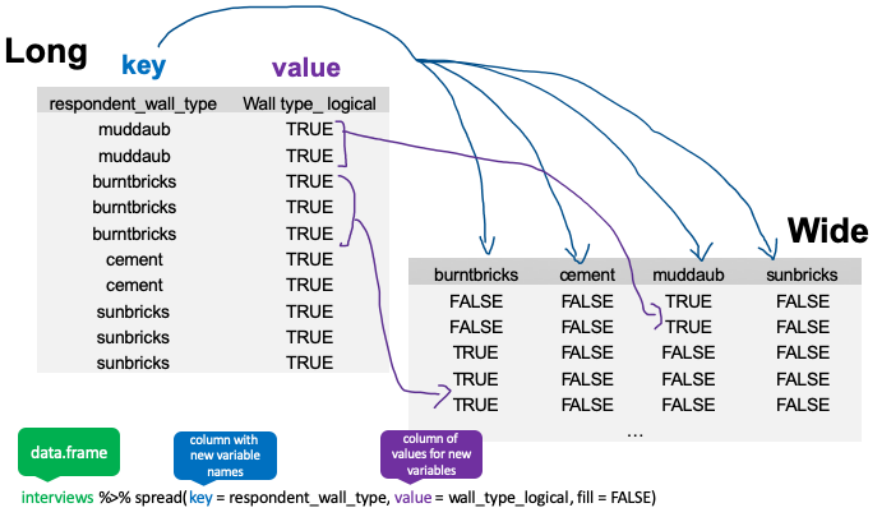


Figure 6.1:

View the `interviews_spread` data frame and notice that there is no longer a column titled `respondent_wall_type`. This is because there is a default parameter in `spread()` that drops the original column.

6.15 Gathering

The opposing situation could occur if we had been provided with data in the form of `interviews_spread`, where the building materials are column names, but we wish to treat them as values of a `respondent_wall_type` variable instead.

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values, and the other variable contains the values previously associated with the column names. We will do this in two steps to make this process a bit clearer.

`gather()` takes four principal arguments:

1. the data

2. the *key* column variable we wish to create from column names.
3. the *value* column variable we wish to create and fill with values associated with the key.
4. the names of the columns we use to fill the key variable (or to drop).

To recreate our original data frame, we will use the following:

1. the data - `interviews_spread`
2. the *key* column will be "respondent_wall_type" (as a character string). This is the name of the new column we want to create.
3. the *value* column will be `wall_type_logical`. This will be either `TRUE` or `FALSE`.
4. the names of the columns we will use to fill the key variable are `burntbricks:sunbricks` (the column named "burntbricks" up to and including the column named "sunbricks" as they are ordered in the data frame).

```
interviews_gather <- interviews_spread %>%
    gather(key = respondent_wall_type, value = "wall_type_logical",
           burntbricks:sunbricks)
```

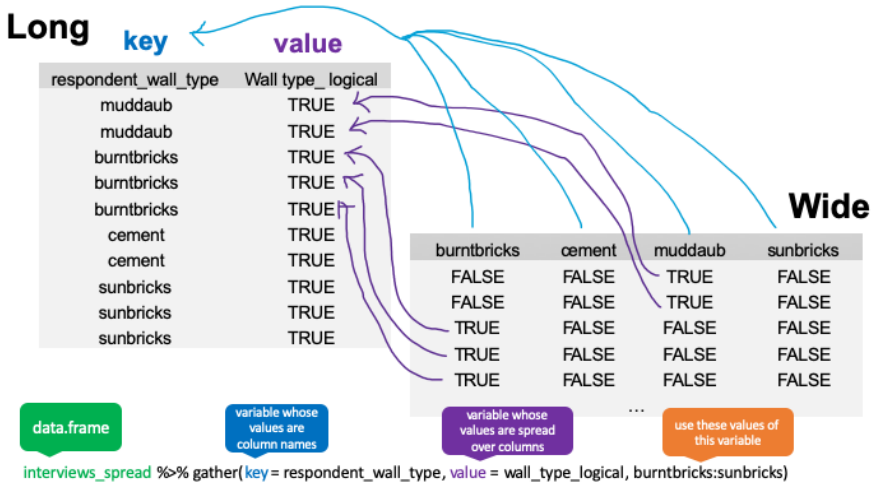


Figure 6.2:

This creates a data frame with 524 rows (4 rows per interview respondent). The four rows for each respondent differ only in the value of the "respondent_wall_type" and "dummy" columns. View the data to see what this looks like.

Only one row for each interview respondent is informative - we know that if the house walls are made of "sunbrick" they aren't made of any other the other

materials. Therefore, we can get filter our dataset to only keep values where `wall_type_logical` is `TRUE`. Because, `wall_type_logical` is already either `TRUE` or `FALSE`, when passing the column name to `filter()`, it will automatically already only keep rows where this column has the value `TRUE`. We can then remove the `wall_type_logical` column. We do all of these steps together in the next chunk of code:

```
interviews_gather <- interviews_spread %>%
    gather(key = "respondent_wall_type", value = "wall_type_logical",
           burntbricks:sunbricks) %>%
    filter(wall_type_logical) %>%
    select(-wall_type_logical)
```

View both `interviews_gather` and `interviews_spread` and compare their structure. Notice that the rows have been reordered in `interviews_gather` such that all of the respondents with a particular wall type are grouped together.

6.16 Applying spread() to clean our data

Now that we've learned about `gather()` and `spread()` we're going to put these functions to use to fix a problem with the way that our data is structured. In the spreadsheets lesson, we learned that it's best practice to have only a single piece of information in each cell of your spreadsheet. In this dataset, we have several columns which contain multiple pieces of information. For example, the `items_owned` column contains information about whether our respondents owned a fridge, a television, etc. To make this data easier to analyze, we will split this column and create a new column for each item. Each cell in that column will either be `TRUE` or `FALSE` and will indicate whether that interview respondent owned that item.

```
interviews_items_owned <- interviews %>%
    separate_rows(items_owned, sep=";") %>%
    mutate(items_owned_logical = TRUE) %>%
    spread(key = items_owned, value = items_owned_logical, fill = FALSE)

nrow(interviews_items_owned)
```

```
## [1] 131
```

There are a couple of new concepts in this code chunk. Let's walk through it line by line. First we create a new object (`interviews_items_owned`) based on the `interviews` dataframe.

```
interviews_items_owned <- interviews %>%
```

Then we use the new function `separate_rows()` to split the column `items_owned` based on the presence of semi-colons (`;`). This creates a long

format version of the dataset. In this long format version, there are 131 rows (one row for each unique item for each respondent).

```
separate_rows(items_owned, sep=";") %>%
```

Lastly, we use `spread()` to switch from long format to wide format. This creates a new column for each of the unique values in the `split_items` column and fills those columns with `TRUE` or `FALSE`.

```
mutate(items_owned_logical = TRUE) %>%
    spread(key = items_owned, value = items_owned_logical, fill = FALSE)
```

View the `interviews_items_owned` data frame. It should have `r nrow(interviews)` rows (the same number of rows you had originally), but extra columns for each item.

You may notice that the last column in called `\``. `This is because the respondents did not own any of the items that was in the interviewer's list. We can use the`rename()` function to change this name to something more meaningful:

```
interviews_items_owned <- interviews_items_owned %>%
    rename(no_listed_items = `<NA>`)
```

This format of the data allows us to do interesting things, like make a table showing the number of respondents in each village who owned a particular item:

```
interviews_items_owned %>%
    filter(bicycle) %>%
    group_by(village) %>%
    count(bicycle)
```

```
## # A tibble: 3 x 3
## # Groups:   village [3]
##   village  bicycle     n
##   <chr>    <lgl>   <int>
## 1 Chirodzo TRUE       17
## 2 God      TRUE       23
## 3 Ruaca    TRUE       20
```

Or calculate the average number of items from the list owned by respondents in each village:

```
interviews_items_owned %>%
    mutate(number_items = rowSums(select(., bicycle:television))) %>%
    group_by(village) %>%
    summarize(mean_items = mean(number_items))
```

```
## # A tibble: 3 x 2
##   village  mean_items
```

```
##   <chr>         <dbl>
## 1 Chirodzo       4.54
## 2 God            3.98
## 3 Ruaca          5.57
```

6.17 Exercise

1. Create a new data frame (named `interviews_months_lack_food`) that has one column for each month and records `TRUE` or `FALSE` for whether each interview respondent was lacking food in that month.

6.18 Solution

```
interviews_months_lack_food <- interviews %>%
  separate_rows(months_lack_food, sep=";") %>%
  mutate(months_lack_food_logical  = TRUE) %>%
  spread(key = months_lack_food, value = months_lack_food_logical, fill = FALSE)
```

{: .solution}

2. How many months (on average) were respondents without food if they did belong to an irrigation association? What about if they didn't?

6.19 Solution

```
interviews_months_lack_food %>%
  mutate(number_months = rowSums(select(., Apr:Sept))) %>%
  group_by(memb_assoc) %>%
  summarize(mean_months = mean(number_months))
```

```
## # A tibble: 3 x 2
##   memb_assoc mean_months
##   <chr>            <dbl>
## 1 no                2.31
## 2 yes               2.64
## 3 <NA>              2.95
```

{: .solution} {: .challenge}

6.20 Exporting data

Now that you have learned how to use **dplyr** to extract information from or summarize your raw data, you may want to export these new data sets to share them with your collaborators or for archival.

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from data frames.

Before using `write_csv()`, we are going to create a new folder, `data_output`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data_output` directory, so even if the files it contains are deleted, we can always re-generate them.

In preparation for our next lesson on plotting, we are going to create a version of the dataset where each of the columns includes only one data value. To do this, we will use spread to expand the `months_lack_food` and `items_owned` columns. We will also create a couple of summary columns.

```
interviews_plotting <- interviews %>%
    ## spread data by items_owned
    separate_rows(items_owned, sep=";") %>%
    mutate(items_owned_logical = TRUE) %>%
    spread(key = items_owned, value = items_owned_logical, fill = FALSE) %>%
    rename(no_listed_items = `<NA>`) %>%
    ## spread data by months_lack_food
    separate_rows(months_lack_food, sep=";") %>%
    mutate(months_lack_food_logical = TRUE) %>%
    spread(key = months_lack_food, value = months_lack_food_logical, fill = FALSE) %>%
    ## add some summary columns
    mutate(number_months_lack_food = rowSums(select(., Apr:Sept))) %>%
    mutate(number_items = rowSums(select(., bicycle:television)))
```

Now we can save this data frame to our `data_output` directory.

```
write_csv(interviews_plotting, path = "data_output/interviews_plotting.csv")
```

Chapter 7

Data Visualization with `ggplot2`

teaching: 80
exercises: 35
adapted from: https://datacarpentry.org/r-socialsci/04-ggplot2/index.html

questions:

- What are the components of a ggplot?
- How do I create scatterplots, boxplots, and barplots?
- How can I change the aesthetics (ex. colour, transparency) of my plot?
- How can I create multiple plots at once?

objectives:

- Produce scatter plots, boxplots, and time series plots using ggplot.
- Set universal plot settings.
- Describe what faceting is and apply faceting in ggplot.
- Modify the aesthetics of an existing ggplot plot (including axis labels and color).
- Build complex and customized plots from data in a data frame.

keypoints:

- `ggplot2` is a flexible and useful tool for creating plots in R.
- The data set and coordinate system can be defined using the `ggplot` function.
- Additional layers, including geoms, are added using the `+` operator.
- Boxplots are useful for visualizing the distribution of a continuous variable.
- Barplot are useful for visualizing categorical data.
- Faceting allows you to generate multiple plots based on a categorical variable.

We start by loading the required package. **`ggplot2`** is also included in the **`tidyverse`** package.

```
library(tidyverse)
```

If not still in the workspace, load the data we saved in the previous lesson.

```
interviews_plotting <- read_csv("data_output/interviews_plotting.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_logical(),
##   key_ID = col_double(),
##   village = col_character(),
##   interview_date = col_datetime(format = ""),
##   no_membrs = col_double(),
##   years_liv = col_double(),
##   respondent_wall_type = col_character(),
##   rooms = col_double(),
##   memb_assoc = col_character(),
##   affect_conflicts = col_character(),
##   liv_count = col_double(),
##   no_meals = col_double(),
##   instanceID = col_character(),
##   number_months_lack_food = col_double(),
##   number_items = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

7.1 Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots from data stored in a data frame. It provides a programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot2 functions like data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with **ggplot2**

ggplot graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a ggplot, we will use the following basic template that can be used for different types of plots:

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

- use the `ggplot()` function and bind the plot to a specific data frame using the `data` argument

```
ggplot(data = interviews_plotting)
```

- define a mapping (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, color, etc.

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items))
```

- add 'geoms' – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:

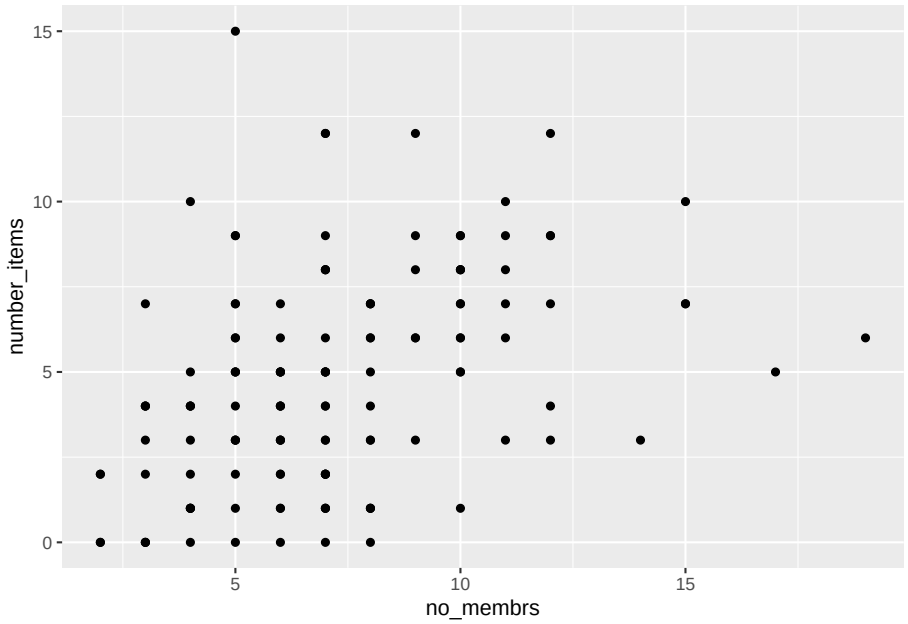
- `geom_point()` for scatter plots, dot plots, etc.

- `geom_boxplot()` for, well, boxplots!

- `geom_line()` for trend lines, time series, etc.

To add a geom to the plot use the `+` operator. Because we have two continuous variables, let's use `geom_point()` first:

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_point()
```



The + in the **ggplot2** package is particularly useful because it allows you to modify existing ggplot objects. This means you can easily set up plot templates and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
interviews_plot <- ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_ite

# Draw the plot
interviews_plot +
    geom_point()
```

7.2 Notes

- Anything you put in the ggplot() function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in aes().
- You can also specify mappings for a given geom independently of the mapping defined globally in the ggplot() function.
- The + sign used to add new layers must be placed at the end of the line containing the *previous* layer. If, instead, the + sign is added at the beginning of the line containing the new layer, **ggplot2** will not add the new layer and will return an error message. {: .callout}

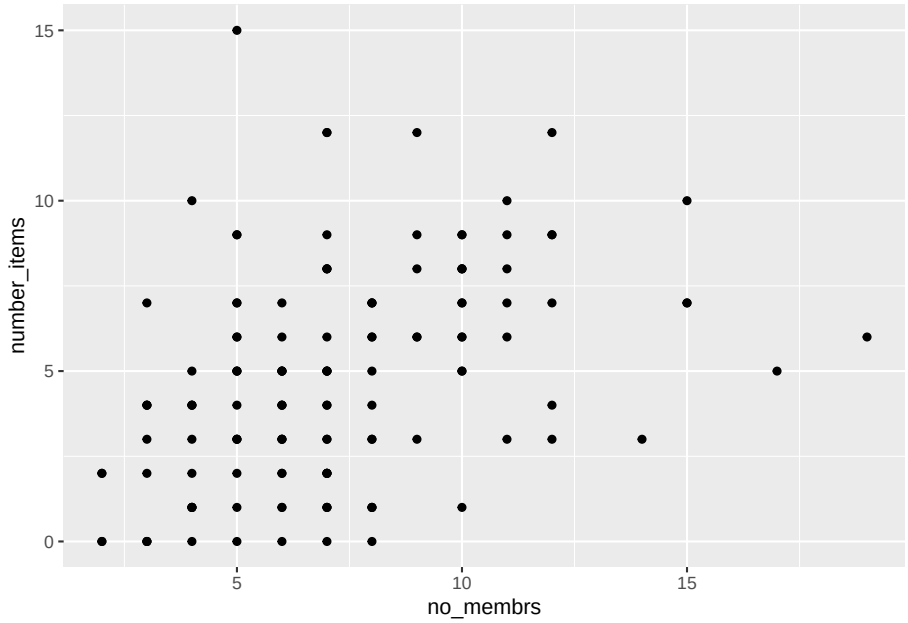
```
## This is the correct syntax for adding layers
interviews_plot +
    geom_point()

## This will not add the new layer and will return an error message
interviews_plot
+ geom_point()
```

7.3 Building your plots iteratively

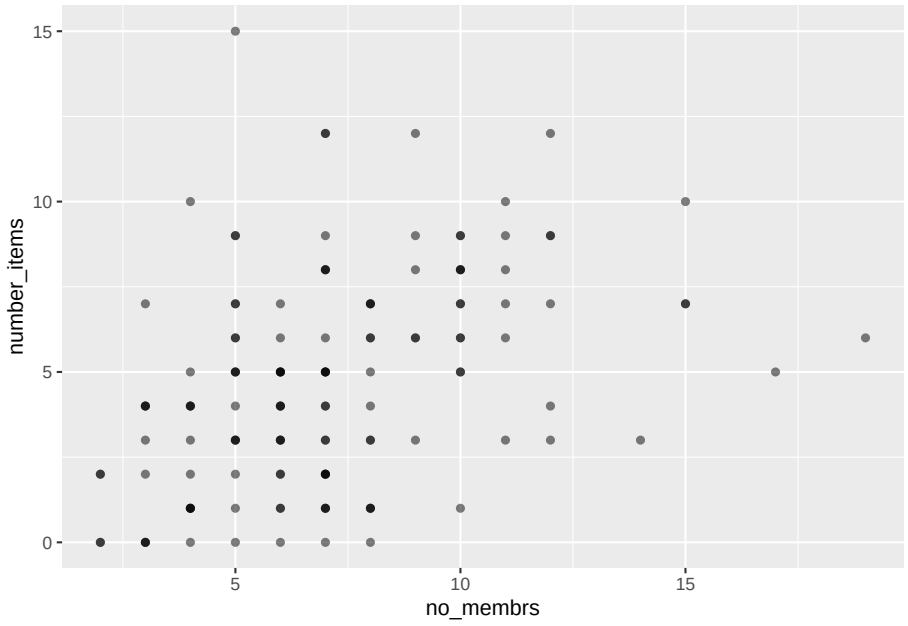
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_point()
```



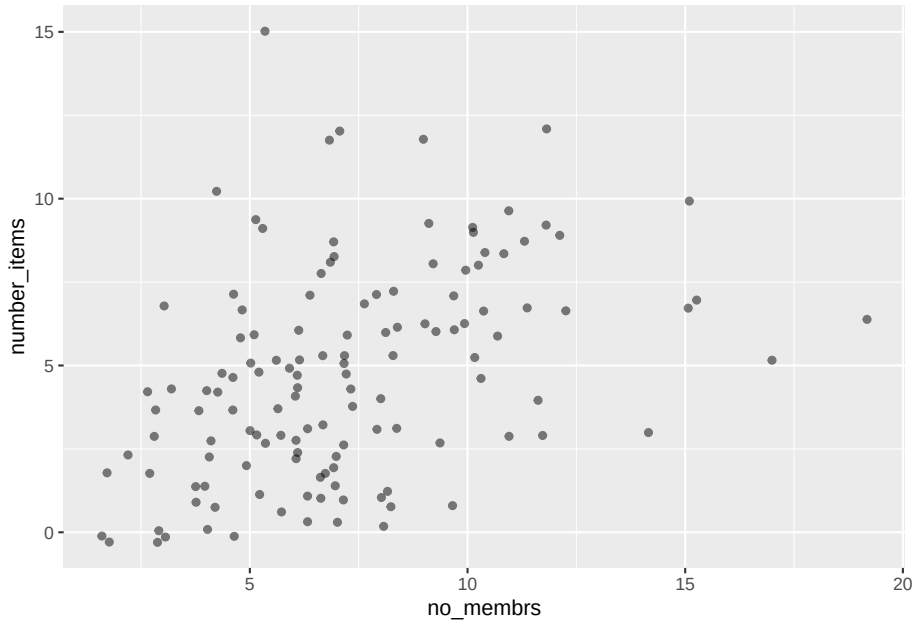
Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_point(alpha = 0.5)
```



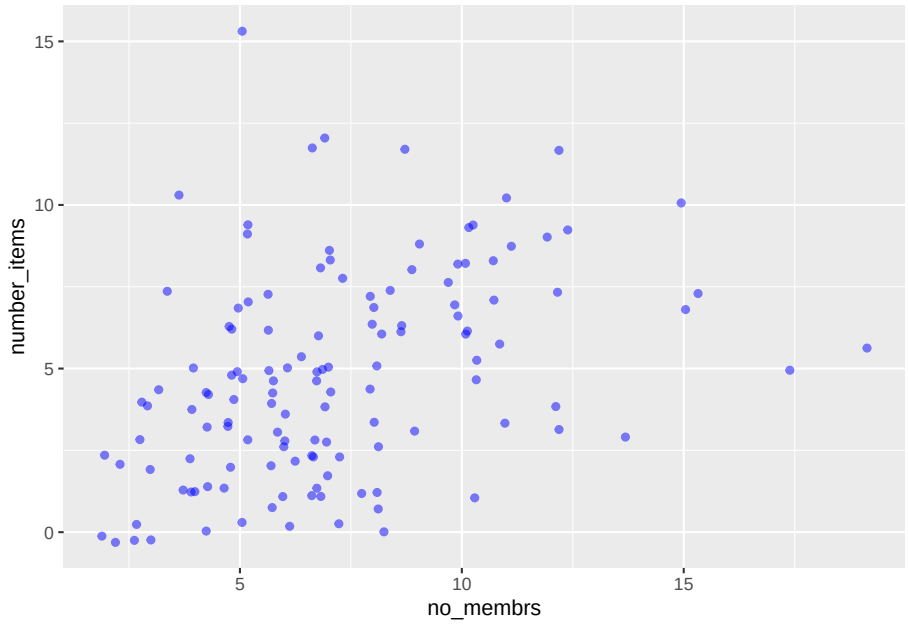
That only helped a little bit with the overplotting problem. We can also introduce a little bit of randomness into the position of our points using the `geom_jitter()` function.

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_jitter(alpha = 0.5)
```



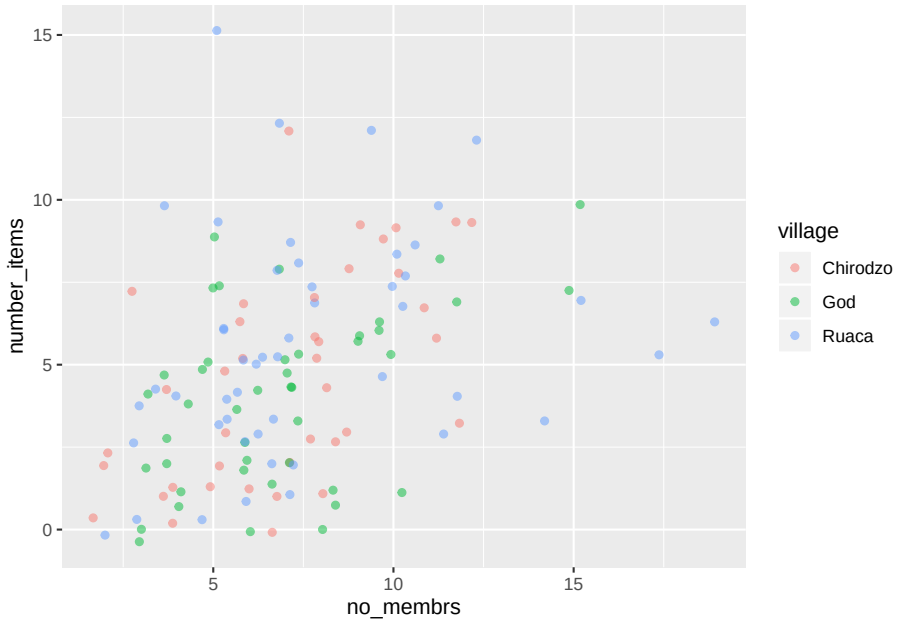
We can also add colors for all the points:

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_jitter(alpha = 0.5, color = "blue")
```



Or to color each species in the plot differently, you could use a vector as an input to the argument **color**. Because we are now mapping features of the data to a color, instead of setting one color for all points, the color now needs to be set inside a call to the **aes** function. **ggplot2** will provide a different color corresponding to different values in the vector. We set the value of **alpha** outside of the **aes** function call because we are using the same value for all points. Here is an example where we color by **village**:

```
ggplot(data = interviews_plotting, aes(x = no_membrs, y = number_items)) +
    geom_jitter(aes(color = village), alpha = 0.5)
```



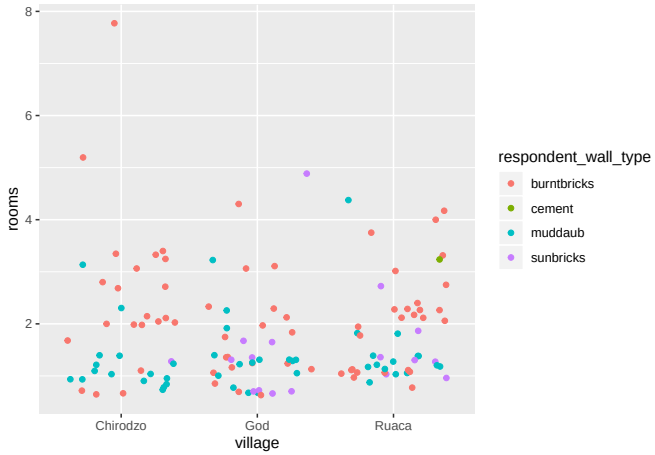
There appears to be a positive trend between number of household members and number of items owned (from the list provided). This trend does not appear to be different by village.

7.4 Exercise

Use what you just learned to create a scatter plot of `rooms` by `village` with the `respondent_wall_type` showing in different colors. Is this a good way to show this type of data?

7.5 Solution

```
ggplot(data = interviews_plotting, aes(x = village, y = rooms)) +
geom_jitter(aes(color = respondent_wall_type))
```

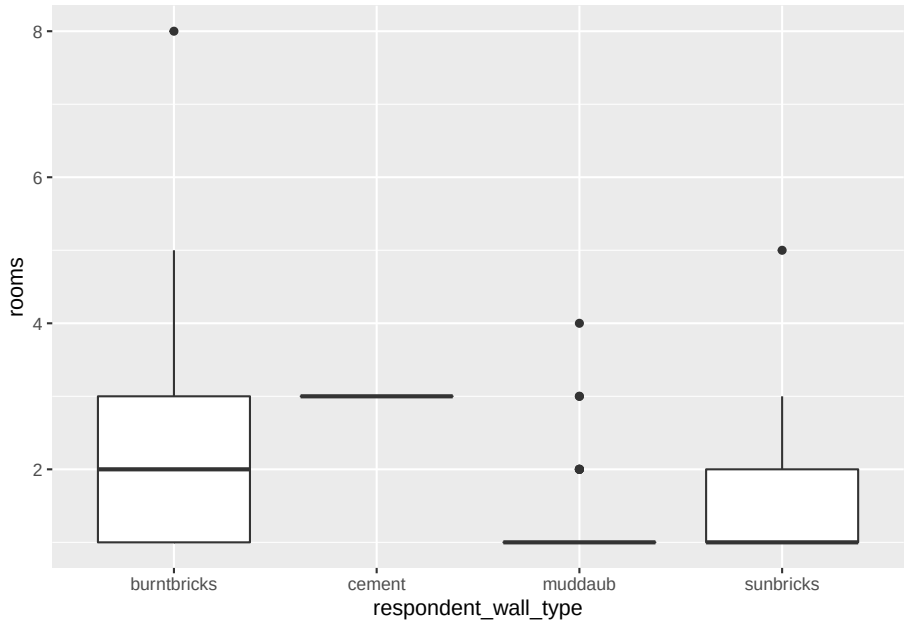


This is not a good way to show this type of data because it is difficult to distinguish between villages. {: .solution} {: .challenge}

7.6 Boxplot

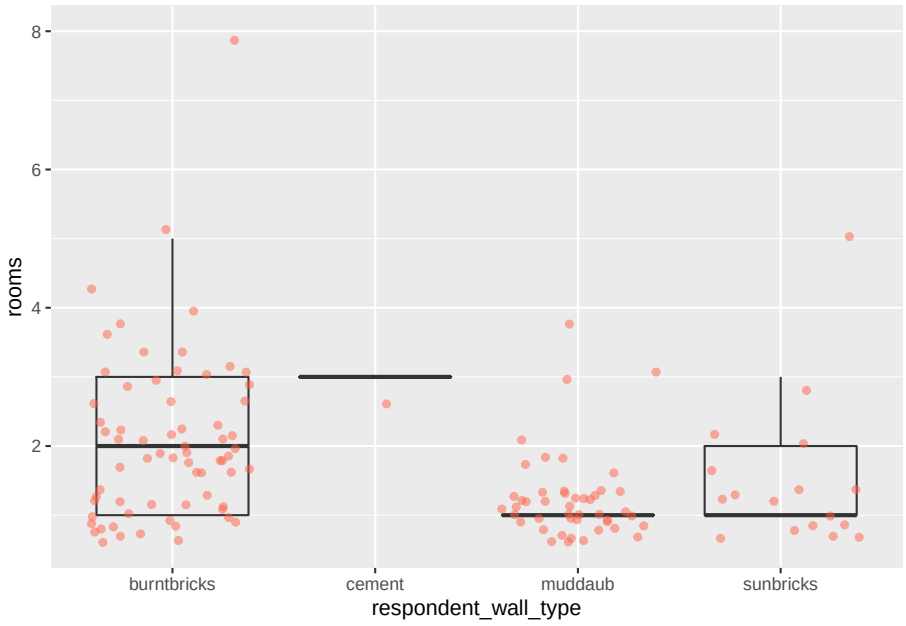
We can use boxplots to visualize the distribution of rooms for each wall type:

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type, y = rooms)) +
    geom_boxplot()
```



By adding points to a boxplot, we can have a better idea of the number of measurements and of their distribution:

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type, y = rooms)) +
    geom_boxplot(alpha = 0) +
    geom_jitter(alpha = 0.5, color = "tomato")
```



We can see that muddaub houses and sunbrick houses tend to be smaller than burntbrick houses.

Notice how the boxplot layer is behind the jitter layer? What do you need to change in the code to put the boxplot in front of the points such that it's not hidden?

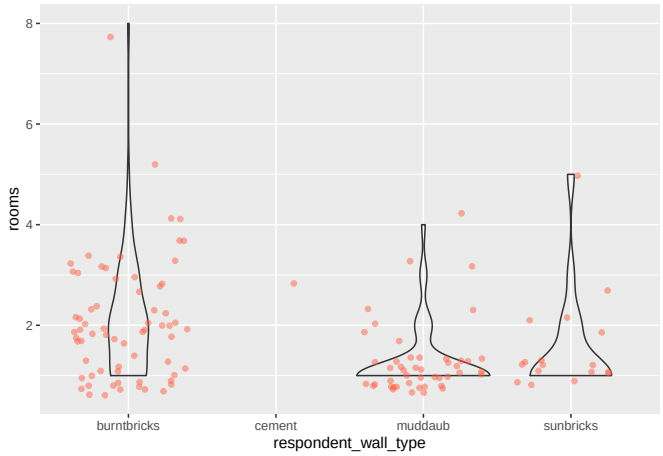
7.7 Exercise

Boxplots are useful summaries, but hide the *shape* of the distribution. For example, if the distribution is bimodal, we would not see it in a boxplot. An alternative to the boxplot is the violin plot, where the shape (of the density of points) is drawn.

- Replace the box plot with a violin plot; see `geom_violin()`.

7.8 Solution

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type, y = rooms)) +
  geom_violin(alpha = 0) +
  geom_jitter(alpha = 0.5, color = "tomato")
```



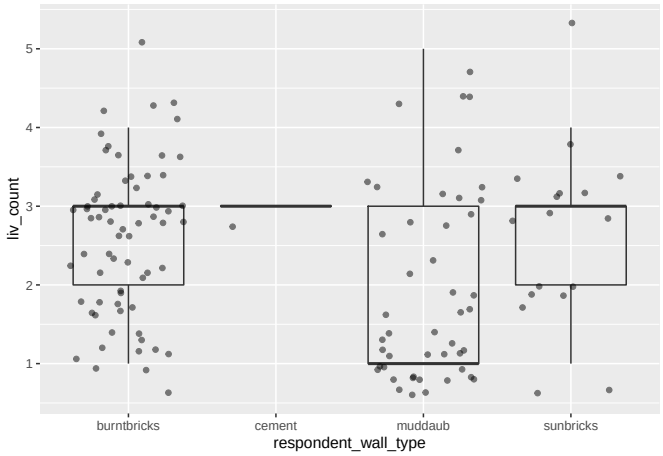
{: .solution}

So far, we've looked at the distribution of room number within wall type. Try making a new plot to explore the distribution of another variable within wall type.

- Create a boxplot for `liv_count` for each wall type. Overlay the boxplot layer on a jitter layer to show actual measurements.

7.9 Solution

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type, y = liv_count
  geom_boxplot(alpha = 0) +
  geom_jitter(alpha = 0.5)
```

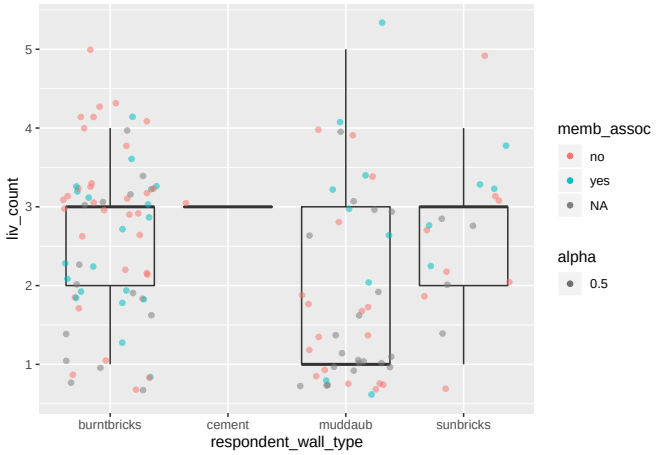


{: .solution}

- Add color to the data points on your boxplot according to whether the respondent is a member of an irrigation association (`memb_assoc`).

7.10 Solution

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type, y = liv_count)) +
  geom_boxplot(alpha = 0) +
  geom_jitter(aes(alpha = 0.5, color = memb_assoc))
```

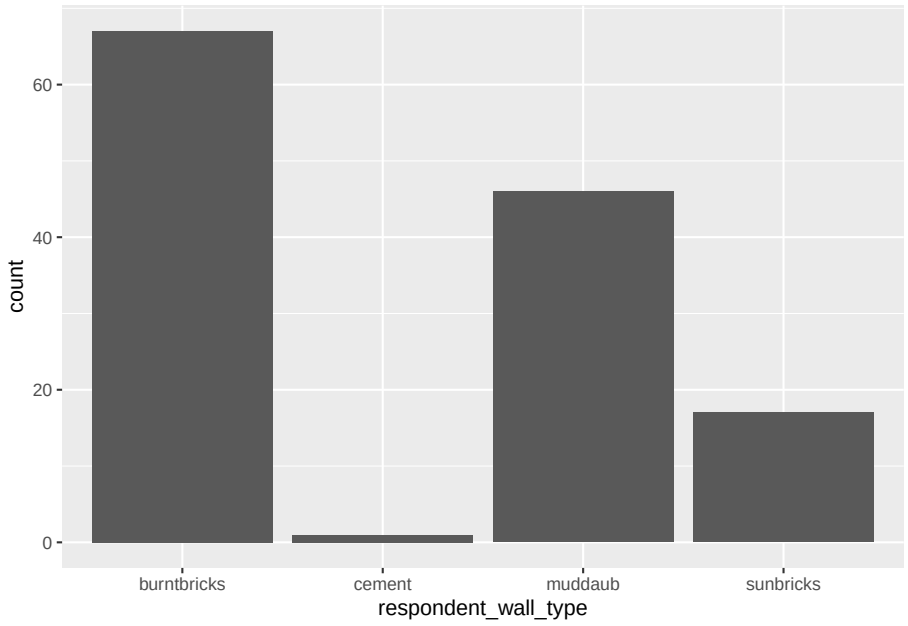


{: .solution} {: .challenge}

7.11 Barplots

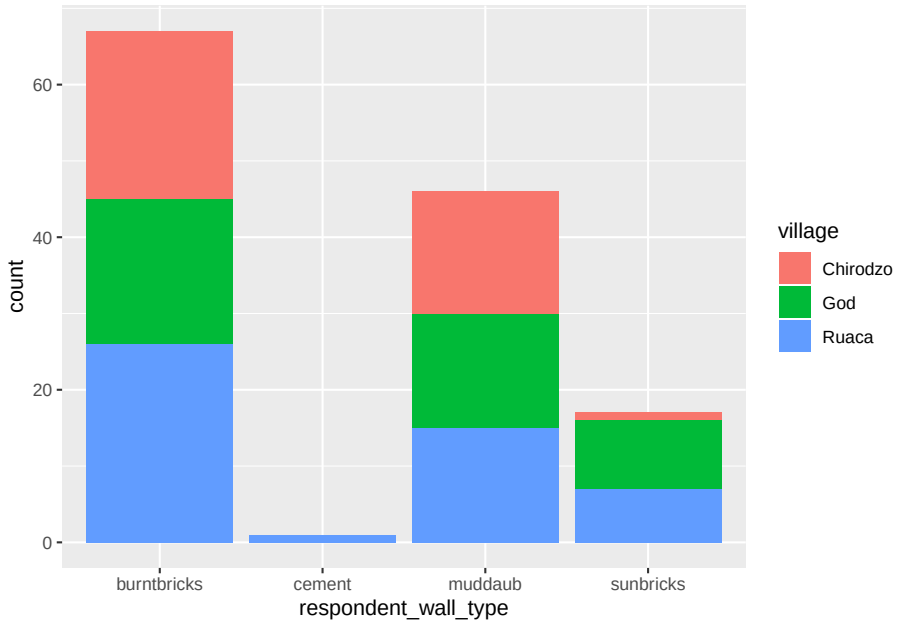
Barplots are also useful for visualizing categorical data. By default, `geom_bar` accepts a variable for x, and plots the number of instances each value of x (in this case, wall type) appears in the dataset.

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type)) +
    geom_bar()
```



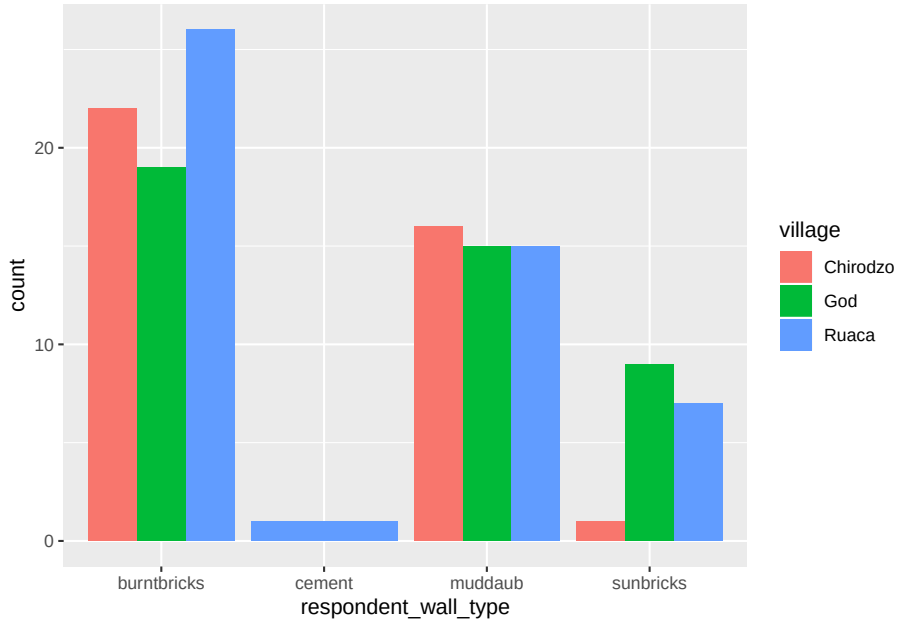
We can use the `fill` aesthetic for the `geom_bar()` geom to color bars by the portion of each count that is from each village.

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type)) +
    geom_bar(aes(fill = village))
```



This creates a stacked bar chart. These are generally more difficult to read than side-by-side bars. We can separate the portions of the stacked bar that correspond to each village and put them side-by-side by using the `position` argument for `geom_bar()` and setting it to "dodge".

```
ggplot(data = interviews_plotting, aes(x = respondent_wall_type)) +
    geom_bar(aes(fill = village), position = "dodge")
```

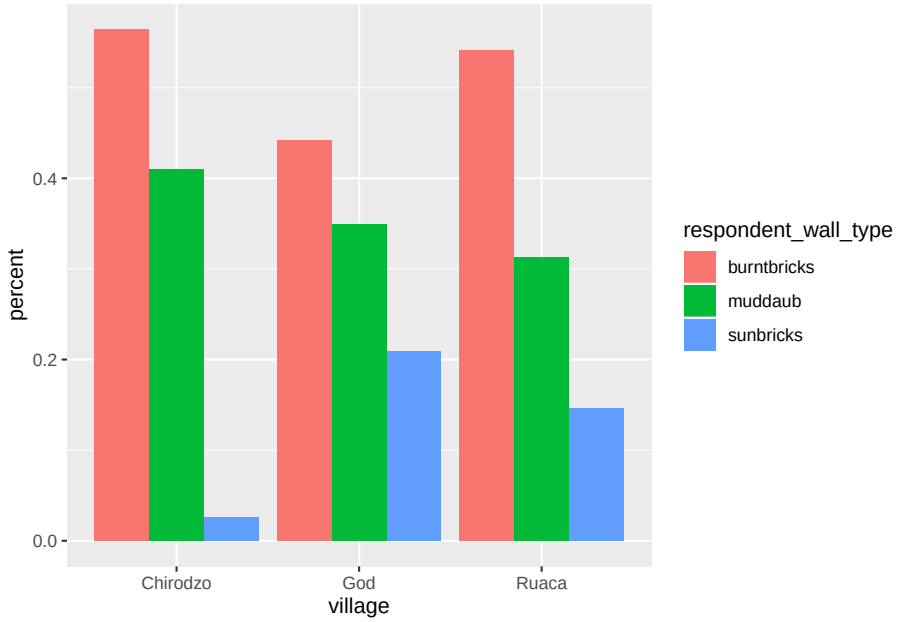


This is a nicer graphic, but we're more likely to be interested in the proportion of each housing type in each village than in the actual count of number of houses of each type (because we might have sampled different numbers of households in each village). To compare proportions, we will first create a new data frame (`percent_wall_type`) with a new column named "percent" representing the percent of each house type in each village. We will remove houses with cement walls, as there was only one in the dataset.

```
percent_wall_type <- interviews_plotting %>%
    filter(respondent_wall_type != "cement") %>%
    count(village, respondent_wall_type) %>%
    group_by(village) %>%
    mutate(percent = n / sum(n)) %>%
    ungroup()
```

Now we can use this new data frame to create our plot showing the percentage of each house type in each village.

```
ggplot(percent_wall_type, aes(x = village, y = percent, fill = respondent_wall_type))
    geom_bar(stat = "identity", position = "dodge")
```



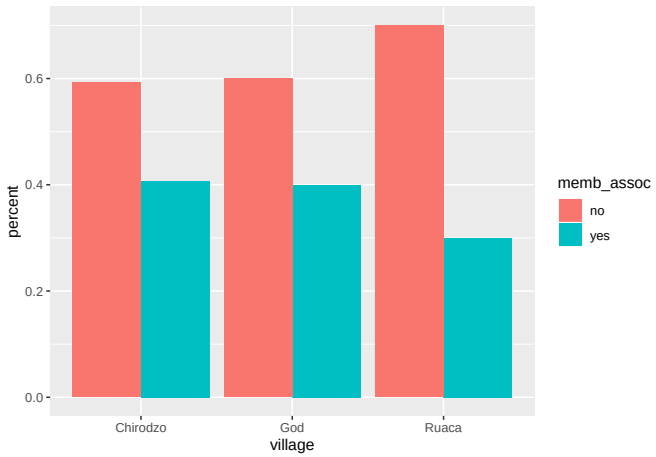
7.12 Exercise

Create a bar plot showing the proportion of respondents in each village who are or are not part of an irrigation association (`memb_assoc`). Include only respondents who answered that question in the calculations and plot. Which village had the lowest proportion of respondents in an irrigation association?

7.13 Solution

```
percent_memb_assoc <- interviews_plotting %>%
  filter(!is.na(memb_assoc)) %>%
  count(village, memb_assoc) %>%
  group_by(village) %>%
  mutate(percent = n / sum(n)) %>%
  ungroup()

ggplot(percent_memb_assoc, aes(x = village, y = percent, fill = memb_assoc)) +
geom_bar(stat = "identity", position = "dodge")
```

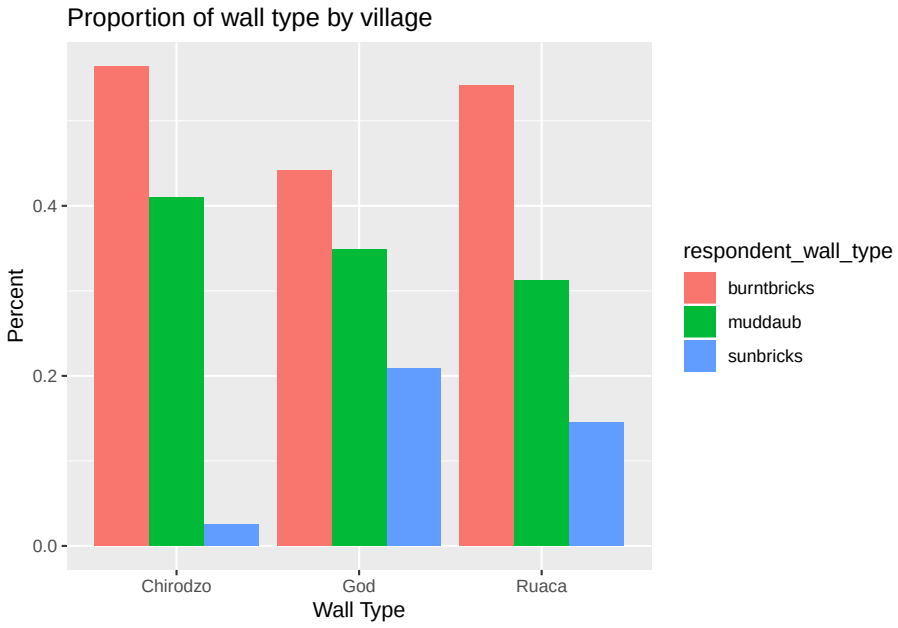


Ruaca had the lowest proportion of members in an irrigation association. {: .solution} {: .challenge}

7.14 Adding Labels and Titles

By default, the axes labels on a plot are determined by the name of the variable being plotted. However, **ggplot2** offers lots of customization options, like specifying the axes labels, and adding a title to the plot with relatively few lines of code. We will add more informative x and y axis labels to our plot of proportion of house type by village and also add a title.

```
ggplot(percent_wall_type, aes(x = village, y = percent, fill = respondent_wall_type)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title="Proportion of wall type by village",
         x="Wall Type",
         y="Percent")
```

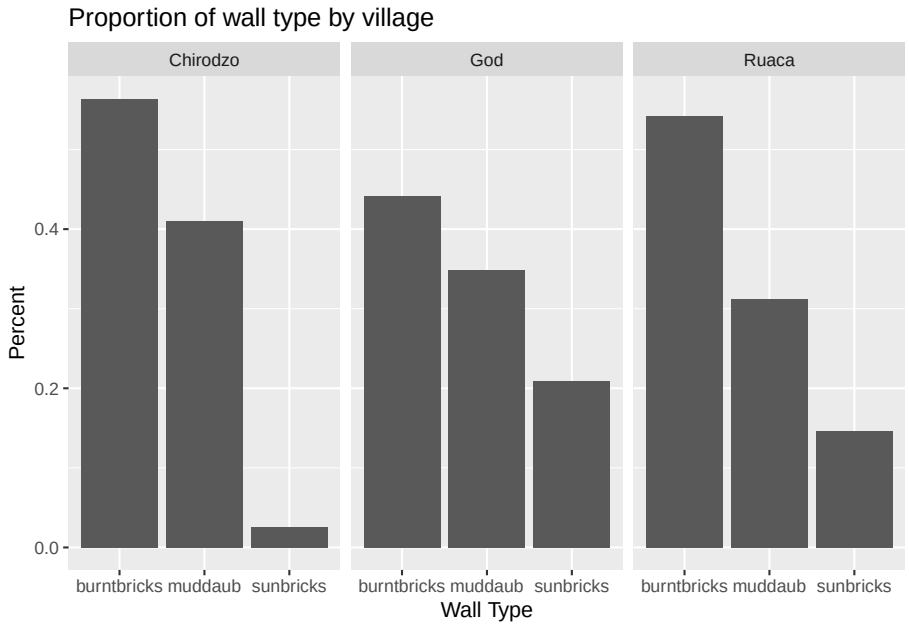


7.15 Faceting

Rather than creating a single plot with side-by-side bars for each village, we may want to create multiple plot, where each plot shows the data for a single village. This would be especially useful if we had a large number of villages that we had sampled, as a large number of side-by-side bars will become more difficult to read.

`ggplot2` has a special technique called *faceting* that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to split our barplot of housing type proportion by village so that each village has it's own panel in a multi-panel plot:

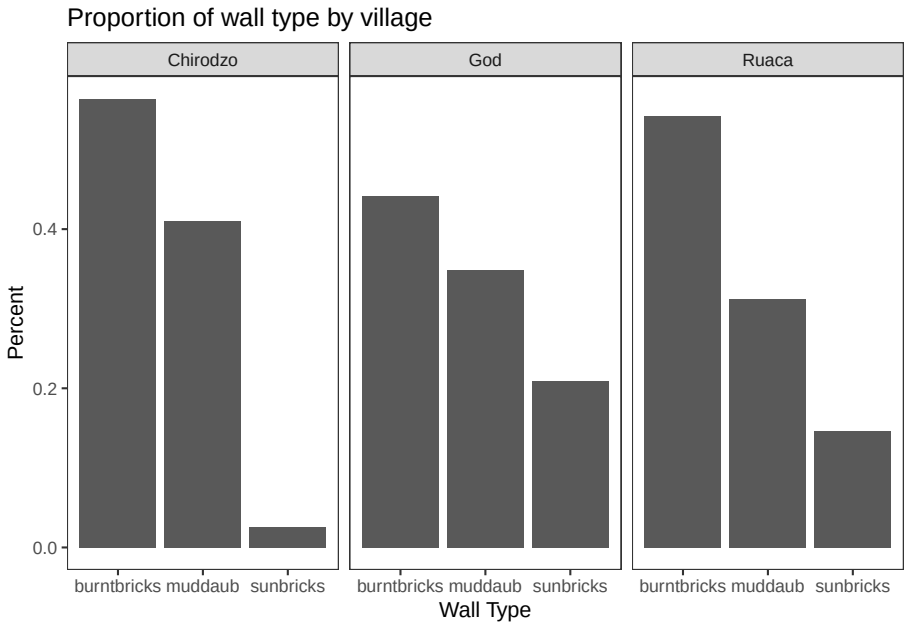
```
ggplot(percent_wall_type, aes(x = respondent_wall_type, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title="Proportion of wall type by village",
         x="Wall Type",
         y="Percent") +
    facet_wrap(~ village)
```



Click the "Zoom" button in your RStudio plots pane to view a larger version of this plot.

Usually plots with white background look more readable when printed. We can set the background to white using the function `theme_bw()`. Additionally, you can remove the grid:

```
ggplot(percent_wall_type, aes(x = respondent_wall_type, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    labs(title="Proportion of wall type by village",
         x="Wall Type",
         y="Percent") +
    facet_wrap(~ village) +
    theme_bw() +
    theme(panel.grid = element_blank())
```



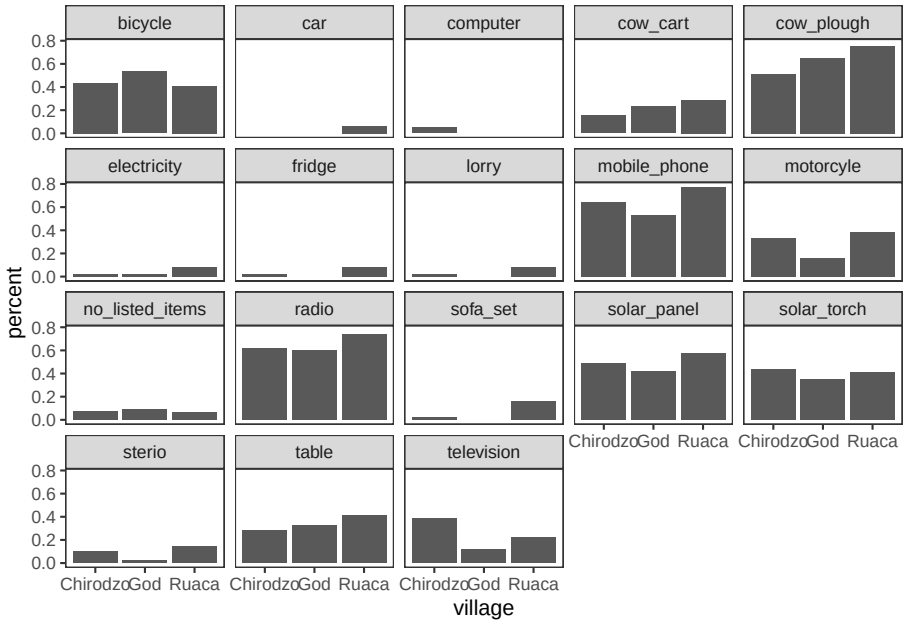
What if we wanted to see the proportion of respondents in each village who owned a particular item? We can calculate the percent of people in each village who own each item and then create a faceted series of bar plots where each plot is a particular item. First we need to calculate the percentage of people in each village who own each item:

```
percent_items <- interviews_plotting %>%
    gather(items, items_owned_logical, bicycle:no_listed_items) %>%
    filter(items_owned_logical) %>%
    count(items, village) %>%
    ## add a column with the number of people in each village
    mutate(people_in_village = case_when(village == "Chirodzo" ~ 39,
                                         village == "God" ~ 43,
                                         village == "Ruaca" ~ 49)) %>%
    mutate(percent = n / people_in_village)
```

To calculate this percentage data frame, we needed to use the `case_when()` parameter within `mutate()`. In our earlier examples, we knew that each house was one and only one of the types specified. However, people can (and do) own more than one item, so we can't use the sum of the count column to give us the denominator in our percentage calculation. Instead, we need to specify the number of respondents in each village. Using this data frame, we can now create a multi-paneled bar plot.

```
ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
```

```
    facet_wrap(~ items) +
    theme_bw() +
    theme(panel.grid = element_blank())
```



7.16 ggplot2 themes

In addition to `theme_bw()`, which changes the plot background to white, **ggplot2** comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at http://docs.ggplot2.org/current/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The ggthemes package provides a wide variety of options (including an Excel 2003 theme). The **ggplot2** extensions website provides a list of packages that extend the capabilities of **ggplot2**, including additional themes.

7.17 Exercise

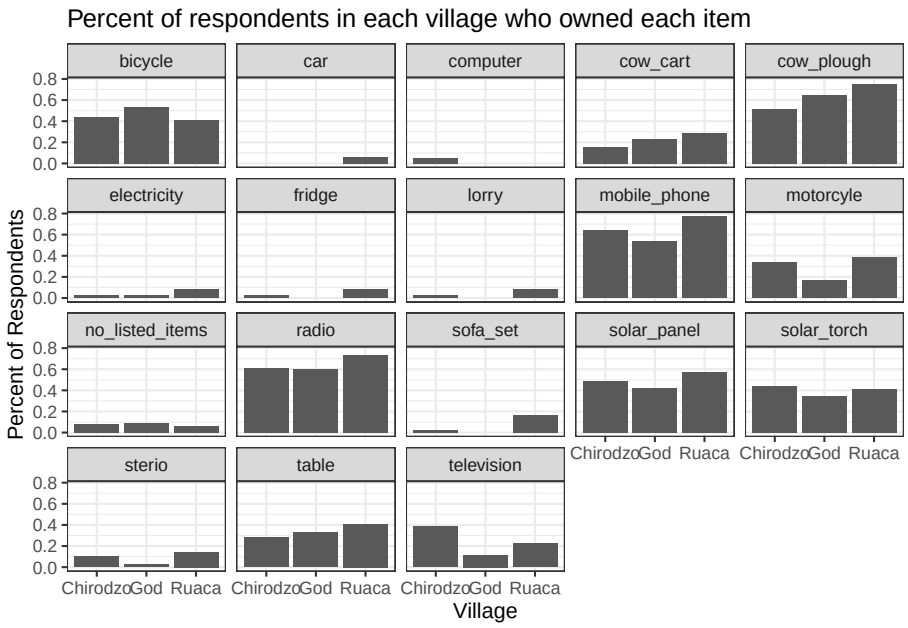
Experiment with at least two different themes. Build the previous plot using each of those themes. Which do you like best? {: .challenge}

7.18 Customization

Take a look at the **ggplot2** cheat sheet, and think of ways you could improve the plot.

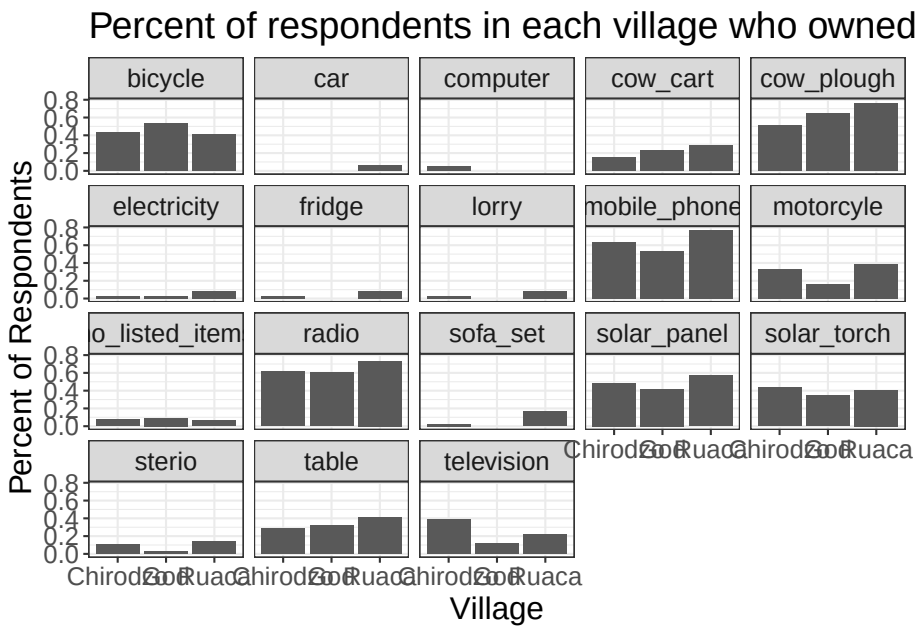
Now, let's change names of axes to something more informative than 'village' and 'percent' and add a title to the figure:

```
ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village who owned each item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw()
```



The axes have more informative names, but their readability can be improved by increasing the font size:

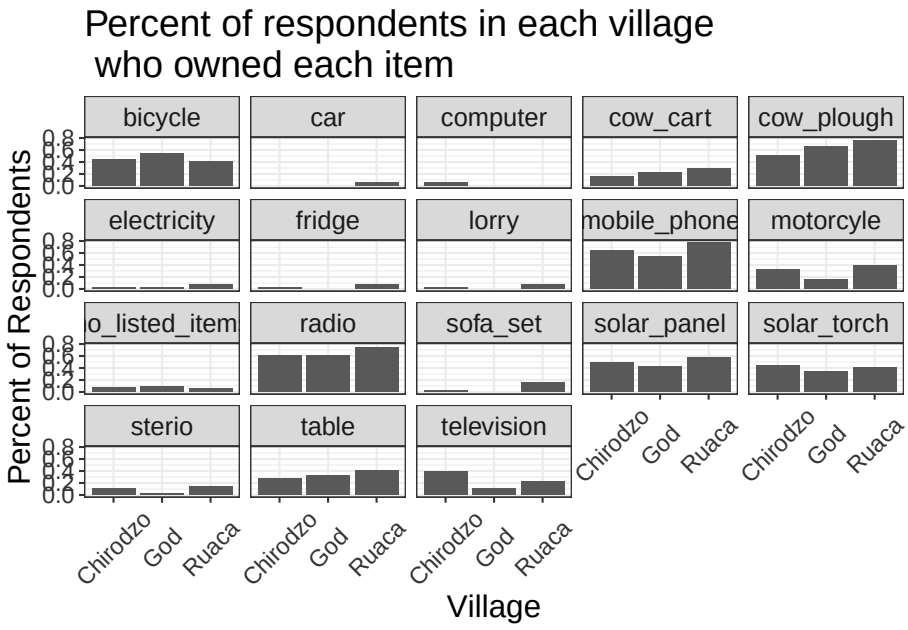
```
ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village who owned each item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw() +
    theme(text=element_text(size = 16))
```



Note that it is also possible to change the fonts of your plots. If you are on Windows, you may have to install the **extrafont** package, and follow the instructions included in the README for this package.

After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90-degree angle, or experiment to find the appropriate angle for diagonally oriented labels. With a larger font, the title also runs off. We can `\n` in the string for the title to insert a new line:

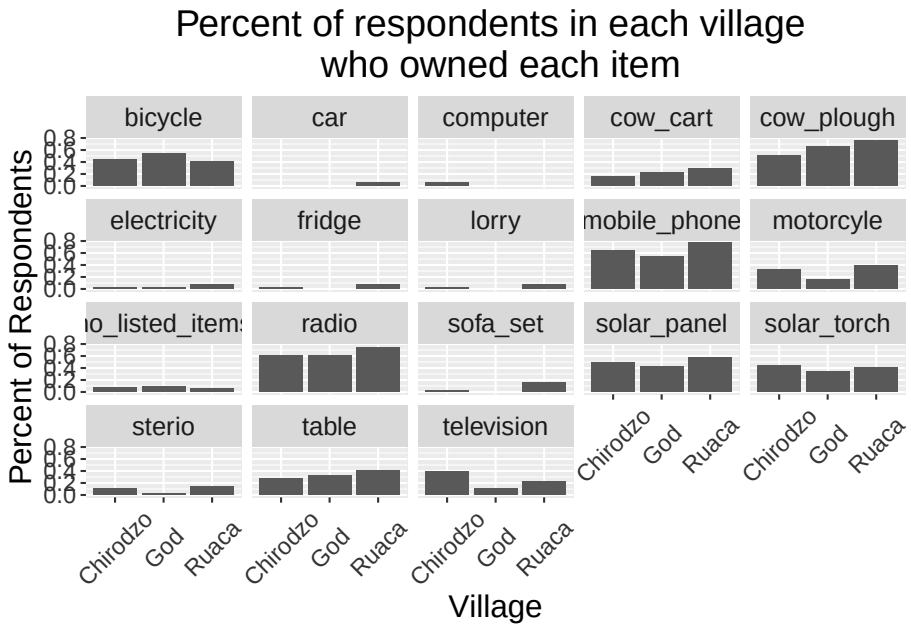
```
ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned each item",
         x = "Village",
         y = "Percent of Respondents") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 45, hjust = 0.5, vjust
          axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16))
```



If you like the changes you created better than the default theme, you can save them as an object to be able to easily apply them to other plots you may create. We can also add `plot.title = element_text(hjust = 0.5)` to center the title:

```
grey_theme <- theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 45
                    axis.text.y = element_text(colour = "grey20", size = 12),
                    text = element_text(size = 16),
                    plot.title = element_text(hjust = 0.5))


ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned each item",
         x = "Village",
         y = "Percent of Respondents") +
    grey_theme
```



7.19 Exercise

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio **ggplot2** cheat sheet for inspiration. Here are some ideas:

- See if you can make the bars white with black outline.
- Try using a different color palette (see http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/). {: .challenge}

After creating your plot, you can save it to a file in your favorite format. The Export tab in the **Plot** pane in RStudio will save your plots at low resolution, which will not be accepted by many journals and will not scale well for posters.

Instead, use the `ggsave()` function, which allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`).

Make sure you have the `fig_output/` folder in your working directory.

```
my_plot <- ggplot(percent_items, aes(x = village, y = percent)) +
    geom_bar(stat = "identity", position = "dodge") +
    facet_wrap(~ items) +
    labs(title = "Percent of respondents in each village \n who owned each item",
         x = "Village",
```

```
         y = "Percent of Respondents") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 45, hjust =
          axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16),
          plot.title = element_text(hjust = 0.5))

ggsave("fig_output/name_of_file.png", my_plot, width = 15, height = 10)
```

Note: The parameters `width` and `height` also determine the font size in the saved plot.

Chapter 8

Advanced variable creation with `forcats`

teaching: 60
exercises: 15
adapted from:
questions:

- "How can I easily create new categorical variables?"

objectives:

-

keypoints:

-

8.1 ForCats

Chapter 9

Producing Reports With `knitr`

teaching: 60
exercises: 15
adapted from: http://swcarpentry.github.io/r-novice-gapminder/15-knitr-markdown/index.html
questions:

- "How can I integrate software and reports?"

objectives:

- Understand the value of writing reproducible reports
- Learn how to recognise and compile the basic components of an R Markdown file
- Become familiar with R code chunks, and understand their purpose, structure and options
- Demonstrate the use of inline chunks for weaving R outputs into text blocks, for example when discussing the results of some calculations
- Be aware of alternative output formats to which an R Markdown file can be exported

keypoints:

- "Mix reporting written in R Markdown with software written in R."

- "Specify chunk options to control formatting."
- "Use `knitr` to convert these documents into PDF and other formats."

9.1 Data analysis reports

Data analysts tend to write a lot of reports, describing their analyses and results, for their collaborators or to document their work for future reference.

Many new users begin by first writing a single R script containing all of the work. Then simply share the analysis by emailing the script and various graphs as attachments. But this can be cumbersome, requiring a lengthy discussion to explain which attachment was which result.

Writing formal reports with Word or LaTeX can simplify this by incorporating both the analysis report and output graphs into a single document. But tweaking formatting to make figures look correct and fix obnoxious page breaks can be tedious and lead to a lengthly "whack a mole" game of fixing new mistakes resulting from a single formatting change.

Creating a web page (as an html file) by using R Markdown makes things easier. The report can be one long stream, so tall figures that wouldn't ordinary fit on one page can be kept full size and easier to read, since the reader can simply keep scrolling. Formatting is simple and easy to modify, allowing you to spend more time on your analyses instead of writing reports.

9.2 Literate programming

Ideally, such analysis reports are *reproducible* documents: If an error is discovered, or if some additional subjects are added to the data, you can just re-compile the report and get the new or corrected results (versus having to reconstruct figures, paste them into a Word document, and further hand-edit various detailed results).

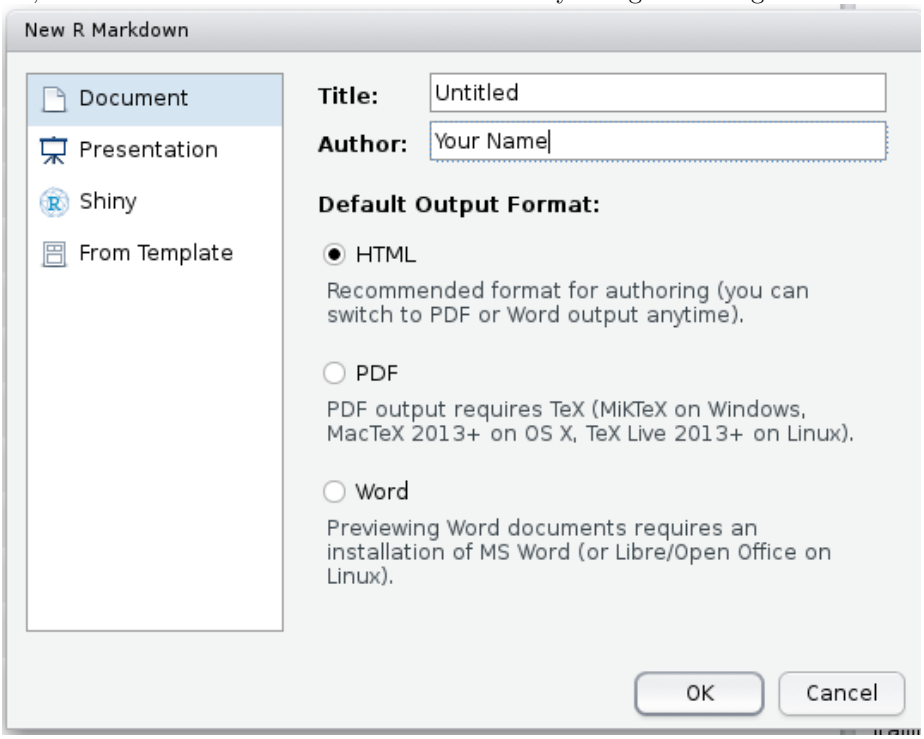
The key R package is `knitr`. It allows you to create a document that is a mixture of text and chunks of code. When the document is processed by `knitr`, chunks of code will be executed, and graphs or other results inserted into the final document.

This sort of idea has been called "literate programming".

`knitr` allows you to mix basically any sort of text with code from different programming languages, but we recommend that you use `R Markdown`, which mixes Markdown with R. Markdown is a light-weight mark-up language for creating web pages.

9.3 Creating an R Markdown file

Within RStudio, click File → New File → R Markdown and you'll get a dialog box like this:



You can stick with the default (HTML output), but give it a title.

9.4 Basic components of R Markdown

The initial chunk of text (header) contains instructions for R to specify what kind of document will be created, and the options chosen. You can use the header to give your document a title, author, date, and tell it that you're going to want to produce html output (in other words, a web page).

```
---
title: "Initial R Markdown document"
author: "Karl Broman"
date: "April 23, 2015"
output: html_document
---
```

You can delete any of those fields if you don't want them included. The double-quotes aren't strictly *necessary* in this case. They're mostly needed if you want to include a colon in the title.

RStudio creates the document with some example text to get you started. Note below that there are chunks like

These are chunks of R code that will be executed by `knitr` and replaced by their results. More on this later. Also note the web address that's put between angle brackets (`< >`) as well as the double-asterisks in `**Knit**`. This is Markdown.

9.5 Markdown

Markdown is a system for writing web pages by marking up the text much as you would in an email rather than writing html code. The marked-up text gets *converted* to html, replacing the marks with the proper html code.

For now, let's delete all of the stuff that's there and write a bit of markdown.

You make things **bold** using two asterisks, like this: `**bold**`, and you make things *italics* by using underscores, like this: `_italics_`.

You can make a bulleted list by writing a list with hyphens or asterisks, like this:

```
* bold with double-asterisks
* italics with underscores
* code-type font with backticks
```

or like this:

```
- bold with double-asterisks
- italics with underscores
- code-type font with backticks
```

Each will appear as:

- bold with double-asterisks
- italics with underscores
- code-type font with backticks

You can use whatever method you prefer, but *be consistent*. This maintains the readability of your code.

You can make a numbered list by just using numbers. You can even use the same number over and over if you want:

```
1. bold with double-asterisks
1. italics with underscores
1. code-type font with backticks
```

This will appear as:

1. bold with double-asterisks
2. italics with underscores
3. code-type font with backticks

You can make section headers of different sizes by initiating a line with some number of `#` symbols:

```
# Title
## Main section
### Sub-section
#### Sub-sub section
```

You *compile* the R Markdown document to an html webpage by clicking the "Knit" button in the upper-left.

9.6 Challenge 1

Create a new R Markdown document. Delete all of the R code chunks and write a bit of Markdown (some sections, some italicized text, and an itemized list).

Convert the document to a webpage. > ## Solution to Challenge 1 > > In RStudio, select File > New file > R Markdown… > > Delete the placeholder text and add the following: > > `> # Introduction > > ## Background on Data > > This report uses the *gapminder* dataset, which has columns that include: > > * country > * continent > * year > * lifeExp > * pop > * gdpPercap > > ## Background on Methods > > >` Then click the 'Knit' button on the toolbar to generate an html document (webpage). {: .solution} {: .challenge}

9.7 A bit more Markdown

You can make a hyperlink like this: `[text to show](http://the-web-page.com)`.

You can include an image file like this: `![caption](http://url/for/file)`

You can do subscripts (e.g., F_2) with `F~2~` and superscripts (e.g., F^2) with `F^2^`.

If you know how to write equations in LaTeX, you can use `$ $` and `$$ $$` to insert math equations, like `$E = mc^2$` and

```
$$y = \mu + \sum_{i=1}^p \beta_i x_i + \epsilon$$
```

You can review Markdown syntax by navigating to the "Markdown Quick Reference" under the "Help" field in the toolbar at the top of RStudio.

9.8 R code chunks

The real power of Markdown comes from mixing markdown with chunks of code. This is R Markdown. When processed, the R code will be executed; if they produce figures, the figures will be inserted in the final document.

The main code chunks look like this:

That is, you place a chunk of R code between “‘{r chunk_name} and “‘. You should give each chunk a unique name, as they will help you to fix errors and, if any graphs are produced, the file names are based on the name of the code chunk that produced them.

9.9 Challenge 2

Add code chunks to:

- Load the ggplot2 package
- Read the gapminder data
- Create a plot

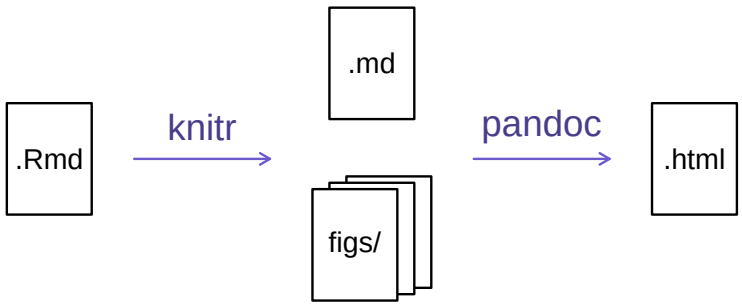
9.10 Solution to Challenge 2

{: .solution} {: .challenge}

9.11 How things get compiled

When you press the “Knit” button, the R Markdown document is processed by `knitr` and a plain Markdown document is produced (as well as, potentially, a set of figure files): the R code is executed and replaced by both the input and the output; if figures are produced, links to those figures are included.

The Markdown and figure documents are then processed by the tool `pandoc`, which converts the Markdown file into an html file, with the figures embedded.



9.12 Chunk options

There are a variety of options to affect how the code chunks are treated. Here are some examples:

- Use `echo=FALSE` to avoid having the code itself shown.
- Use `results="hide"` to avoid having any results printed.
- Use `eval=FALSE` to have the code shown but not evaluated.
- Use `warning=FALSE` and `message=FALSE` to hide any warnings or messages produced.
- Use `fig.height` and `fig.width` to control the size of the figures produced (in inches).

So you might write:

Often there will be particular options that you'll want to use repeatedly; for this, you can set *global* chunk options, like so:

The `fig.path` option defines where the figures will be saved. The `/` here is really important; without it, the figures would be saved in the standard place but just with names that begin with `Figs`.

If you have multiple R Markdown files in a common directory, you might want to use `fig.path` to define separate prefixes for the figure file names, like `fig.path="Figs/cleaning-"` and `fig.path="Figs/analysis-"`.

9.13 Challenge 3

Use chunk options to control the size of a figure and to hide the code.

9.14 Solution to Challenge 3

{: .solution} {: .challenge}

You can review all of the `R` chunk options by navigating to the "R Markdown Cheat Sheet" under the "Cheatsheets" section of the "Help" field in the toolbar at the top of RStudio.

9.15 Inline R code

You can make *every* number in your report reproducible. Use ‘r and ‘ for an in-line code chunk, like so: ‘r round(some_value, 2)‘. The code will be executed and replaced with the *value* of the result.

Don't let these in-line chunks get split across lines.

Perhaps precede the paragraph with a larger code chunk that does calculations and defines variables, with `include=FALSE` for that larger chunk (which is the same as `echo=FALSE` and `results="hide"`).

Rounding can produce differences in output in such situations. You may want `2.0`, but `round(2.03, 1)` will give just `2`.

The `myround` function in the R/broman package handles this.

9.16 Challenge 4

Try out a bit of in-line R code.

9.17 Solution to Challenge 4

Here's some inline code to determine that 2 + 2 = `` `r 2+2` ``.

{: .solution} {: .challenge}

9.18 Other output options

You can also convert R Markdown to a PDF or a Word document. Click the little triangle next to the "Knit" button to get a drop-down menu. Or you could put `pdf_document` or `word_document` in the initial header of the file.

9.19 Tip: Creating PDF documents

Creating .pdf documents may require installation of some extra software. If required this is detailed in an error message.

- TeX installers for Windows.
- TeX installers for macOS. {: .callout}

9.20 Resources

- Knitr in a knutshell tutorial
- Dynamic Documents with R and knitr (book)
- R Markdown documentation
- R Markdown cheat sheet
- Getting started with R Markdown
- R Markdown: The Definitive Guide (book by Rstudio team)
- Reproducible Reporting
- The Ecosystem of R Markdown
- Introducing Bookdown