Introduction to R

OSDC MiniSeries: Reproducible Research

These courses were created from curriculum originally developed by The Carpentries.

The aim of the OSDC MiniSeries workshops is to teach researchers basic concepts, skills, and tools for working with data so that they can get more done in less time, and with less pain. The lessons below were designed for those interested in working with social sciences data in R.

This is an introduction to R designed for participants with no programming experience. It begins with basic information about R syntax and the RStudio interface, and moves through how to import CSV files, the structure of data frames, how to deal with factors, how to add/remove rows and columns, and how to calculate summary statistics from a data frame.

Getting Started

The teaching style of these workshops is hands-on, so participants are encouraged to use their own computers to ensure the proper setup of tools for an efficient workflow.

These lessons assume no prior knowledge of the skills or tools. To get started, follow the directions in the "Setup" tab to download data to your computer and follow any installation instructions.

Prerequisites

This lesson requires a working copy of **R** and **RStudio**. To most effectively use these materials, please make sure to install everything before working through this lesson.

For Instructors

If you are teaching this lesson in a workshop, please see the Instructor notes.

Schedule

	Setup	Install R and RStudio Download files required for the lesson
00:00	Before we Start	How to find your way around RStudio? How to interact with R? How to manage your environment? How to install packages?
00:30	Introduction to R	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? How does R treat missing values? How can we deal with missing values in R?
01:20	Starting with Data	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?
02:10	Data Wrangling with dplyr	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 I create new columns or remove existing columns from a data frame
03:00	Finish	

(The actual schedule may vary slightly depending on the topics and exercises chosen by the instructor.)

Setup instructions

Overview

Questions

- How to install R and RStudio?

Objectives

- Install latest version of R.
- Install latest version of RStudio.

R and **RStudio** are separate downloads and installations. R is the underlying statistical computing environment, but using R alone is no fun. RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You need to install R before you install RStudio. After installing both programs, you will need to install the **tidyverse** package from within RStudio. Follow the instructions below for your operating system, and then follow the instructions to install **tidyverse**.

Windows

If you already have R and RStudio installed

- Open RStudio, and click on 'Help' > 'Check for updates'. If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check which version of R you are using, start RStudio and the first thing that appears in the console indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go to the CRAN website and check whether a more recent version is available. If so, please download and install it. You can check here for more information on how to remove old versions from your system if you wish to do so.

If you don't have R and RStudio installed

- Download R from the CRAN website.
- Run the .exe file that was just downloaded
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Windows Vista/7/8/10** (where x, y, and z represent version numbers)
- Double click the file to install it
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

MacOS

If you already have R and RStudio installed

- Open RStudio, and click on ‘Help’ > ‘Check for updates’. If a new version is available, quit RStudio, and download the latest version for RStudio.
- To check the version of R you are using, start RStudio and the first thing that appears on the terminal indicates the version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display which version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

If you don’t have R and RStudio installed

- Download R from the CRAN website.
- Select the .pkg file for the latest R version
- Double click on the downloaded file to install R
- It is also a good idea to install XQuartz (needed by some packages)
- Go to the RStudio download page
- Under *Installers* select **RStudio x.yy.zzz - Mac OS X 10.6+ (64-bit)** (where x, y, and z represent version numbers)
- Double click the file to install RStudio
- Once it’s installed, open RStudio to make sure it works and you don’t get any error messages.

Linux

- Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run sudo apt-get install r-base, and for Fedora sudo yum install R), but we don’t recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 3.3.1.
- Go to the RStudio download page
- Under *Installers* select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu sudo dpkg -i rstudio-x.yy.zzz-amd64.deb at the terminal).
- Once it’s installed, open RStudio to make sure it works and you don’t get any error messages.

Before We Start

Overview

Teaching: 25 min
Exercises: 5 min
Questions

- What is R? What is RStudio?
- Why Learn R?
- How do I interact with the RStudio environment?
- How do I install packages?

Objectives

- Learn the benefits of using R for data management.
- Tour the RStudio environment.
- Learn to install R packages.

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 Environment) for R. An IDE is a piece of software that provides tools to make programming easier.

Why learn R?

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

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

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

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

R code is great for reproducibility

Reproducibility 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.

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.

R works on data of all shapes and sizes

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

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

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

R produces high-quality graphics

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

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.

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.

A Tour of RStudio

We will use the RStudio 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.

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 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.

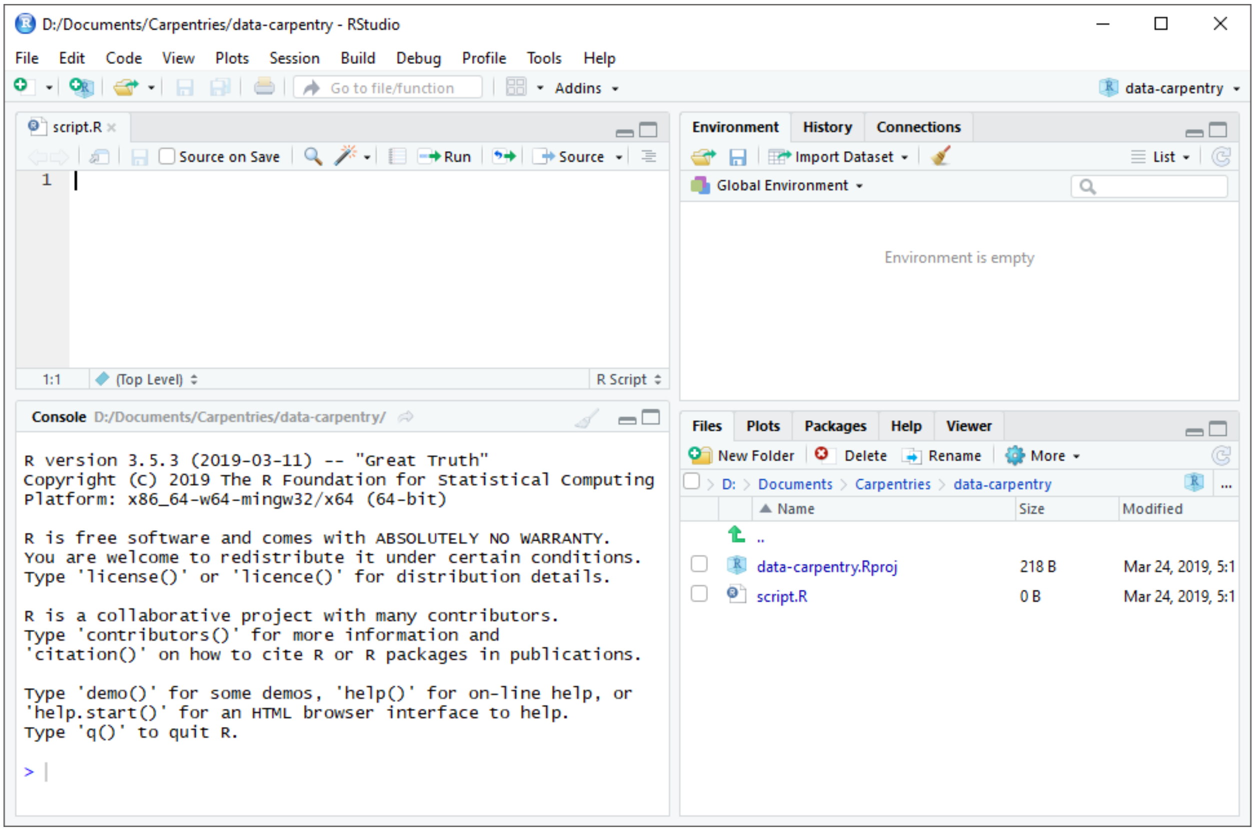
RStudio provides a helpful set of tools to do this through its “Projects” interface, which not only creates a working directory for you but also remembers its location (allowing you to quickly navigate to it). The interface also (optionally) preserves custom settings and open files to make it easier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

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., ~/intro-to-r)
- 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”.

The RStudio Interface

Let’s take a quick tour of RStudio.


D:/Documents/Carpentries/data-carpentry - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
data-carpentry
script.R
Source on Save
Run
Source
(Top Level)
R Script
Environment History Connections
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder Delete Rename More
D: > Documents > Carpentries > data-carpentry
Name Size Modified
data-carpentry.Rproj 218 B Mar 24, 2019, 5:1
script.R 0 B Mar 24, 2019, 5:1
Console D:/Documents/Carpentries/data-carpentry/

```
R version 3.5.3 (2019-03-11) -- "Great Truth"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

>
```

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 - **Environment/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

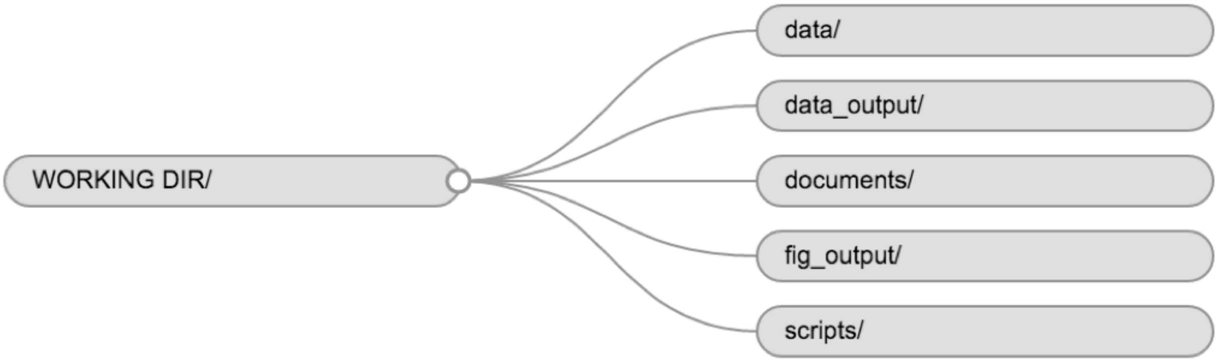
Organizing your working directory

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. 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.



The working directory

The working directory is an important concept to understand. It is the place where R will look for and save files.

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.

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")
```

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 (bottom left panel in RStudio) 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.

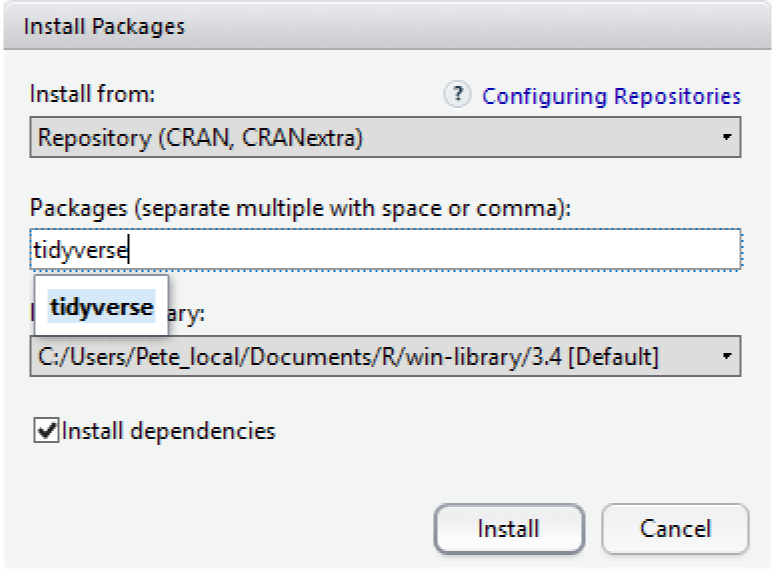
RStudio allows you to execute commands directly from the script editor by using the `Ctrl` + `Enter` shortcut (on Mac, `Cmd` + `Return` will work). The command on the current line in the script (indicated by the cursor) or all of the commands in selected text will be sent to the console and executed when you press `Ctrl` + `Enter`.

Installing additional packages using the packages tab

In addition to the core R installation, there are more than 10,000 additional packages that 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 for anyone to download and install into their own R environment.

You can see if you have a package installed by looking in the packages tab (on the lower-right by default). You can also type the command `installed.packages()` into the console and examine the output.

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.



Install from: Repository (CRAN, CRANextra)

Packages (separate multiple with space or comma): tidyverse

ary: C:/Users/Pete_local/Documents/R/win-library/3.4 [Default]

☑ Install dependencies

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.

Exercise

Use both the Console and the Packages tab to confirm that you have the tidyverse installed.

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

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.

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

Key Points

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

Introduction to R

Overview

Teaching: 30 min
Exercises: 20 min
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?
- 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.

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`.

Note: 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. More generally we prefer the `<-` syntax over `=` because it increases the readability of the code.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. However, there are some general rules and best practices concerning naming.

Naming Rules

- Can't 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`).

Naming Best Practices

- Object names should be explicit and not too long.
- Use nouns for object names, and verbs for function names.

- 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.
- 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.

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.

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 parentheses 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
```

Exercise

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

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
```

Exercise

Create two variables `length` and `width` and assign them values. It should be noted that, because `length` and `width` are built-in R functions, R Studio might add "`()`" after `length` and `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`.

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 later 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 used to alter the way the function operates, such as whether it ignores certain values, or what symbol to use in a plot.

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.

Exercise

Type `?round` into 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?

Vectors and data types

A vector is the most common and basic data type in R. A vector is composed of 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 the number 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`. Since 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 also 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. 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`).

Exercise 1

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?

Exercise 2

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

```
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?

Exercise 3

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)
```

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.

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.

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 < 4 | hh_members > 7]
[1]  3 10
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.

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

The function `%in%` allows you to test if any of the elements of a search vector (on the left hand side) are found in the target vector (on the right hand side):

```
possessions %in% c("car", "bicycle")
[1]  TRUE  TRUE FALSE FALSE FALSE
```

Note that the output is the same length as the search vector on the left hand side, because `%in%` checks whether each element of the search vector is found somewhere in the target vector. Thus, you can use `%in%` to select the elements in the search vector that appear in your target vector:

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

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
## Count the number of missing values.
sum(is.na(rooms))
[1] 1
## Returns the object with incomplete cases removed. The returned
object is an atomic vector of type `"numeric"` (or `"double"`).
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 vector of type `"numeric"` (or `"double"`).
rooms[complete.cases(rooms)]
[1] 2 1 1 4
```

Exercise

1. Using this vector of `rooms`, create a new vector with the `NA`s 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.

Key Points

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

Starting with Data

Overview

Teaching: 30 min
Exercises: 20 min
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?

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.
- Subset and extract values from data frames.
- 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.

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. Data frames are analogous to the more familiar spreadsheet in programs such as Excel, with one key difference. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors).

A data frame can be created manually, 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.

The base R function `read.csv()` stores data as a data frame, where `read_csv()` from the **tidyverse** package stores data as a tibble. We prefer tibbles because they have nice printing properties among other desirable qualities. Read more about tibbles here.

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

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)
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 to load the data in R's memory using the function `read_csv()` from the **readr** package, which is part of the **tidyverse**. (Learn more about the **tidyverse** collection of packages here.) **readr** 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, including **readr**.

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 R in the function to automatically convert all the "NULL" entries in the dataset into `NA`.

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

The second statement in the code above creates a data frame but doesn't output any data because, as you might recall, assignments (`<-`) don't display anything. If we want to check that our data has been loaded, we can see the contents of the data frame by typing its name: `interviews` in the console.

```
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>
```

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 as the first element, and the number of columns as the second element (the dimensions 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," i.e., they can be used on other types of objects besides data frames or tibbles.

Indexing and subsetting data frames

Our `interviews` data frame has rows and columns (it has 2 dimensions). In practice, we may not need the entire data frame. For instance, we may only be interested in a subset of the observations (the rows) or a particular set of variables (the columns). 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.

Tip

Indexing a tibble with `[` or `[[` or `$` always results in a tibble. However, note this is not true in general for data frames, so be careful! Different ways of specifying these coordinates can lead to results with different classes.

```
## first element in the first column of the data frame (as a data.frame)
interviews[1, 1]
# A tibble: 1 x 1
  key_ID
   <dbl>
1      1
## first element in the 6th column (as a data.frame)
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 data.frame)
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_type 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 Chirodzo 2016-11-16 00:00:00        12        70 burntbricks              3
 9 Chirodzo 2016-11-16 00:00:00         8         6 burntbricks              1
10 Chirodzo 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
```

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. Using the number of rows in the `interviews` dataset that you found in question 2, extract the row that is in the middle of the dataset. Store the content of this middle row in an object named `interviews_middle`. (hint: This dataset has an odd number of rows, so finding the middle is a bit trickier than dividing `n_rows` by `2`. Use the `median()` function and what you’ve learned about sequences in R to extract the middle row!)
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.

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 represent categorical data. They are stored as integers associated with labels and they can be ordered (ordinal) or unordered (nominal). Factors create a structured relation between the different levels (values) of a categorical variable, such as days of the week or responses to a question in a survey. This can make it easier to see how one element relates to the other elements in a column. 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 predefined 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 may improve your visualization, or it may be 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`. 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
```

So far, your factor is unordered, like a nominal variable. R does not know the difference between a nominal and an ordinal variable. You make your factor an ordered factor by using the `ordered=TRUE` option inside your factor function. Note how the reported levels changed from the unordered factor above to the ordered version below. Ordered levels use the less than sign `<` to denote level ranking.

```
respondent_floor_type_ordered <- factor(respondent_floor_type,
ordered=TRUE)
respondent_floor_type_ordered # after setting as ordered factor
[1] earth brick brick earth
Levels: earth < brick
```

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:

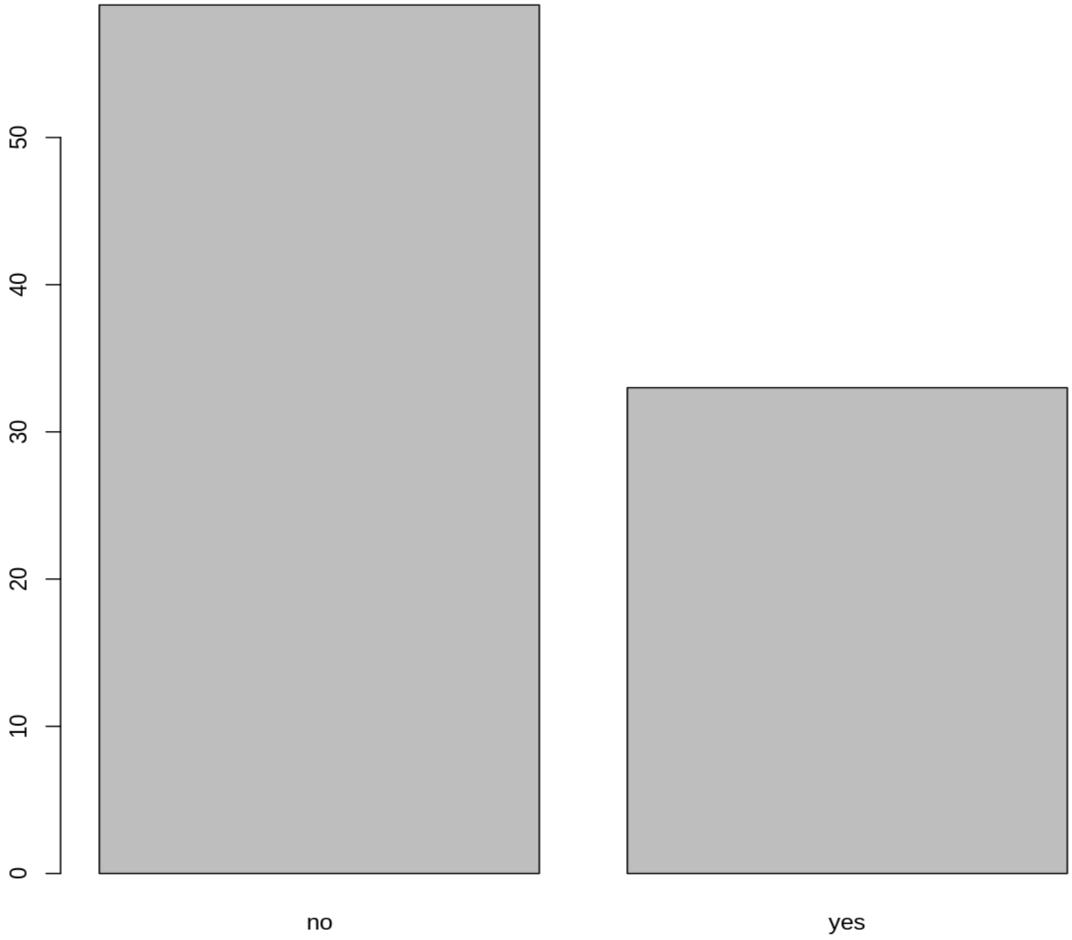
- We obtain all the factor levels using `levels(year_fct)`
- We convert these levels to numeric values using `as.numeric(levels(year_fct))`
- We then access these numeric values using the underlying integers of the vector `year_fct` inside the square brackets

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>
```

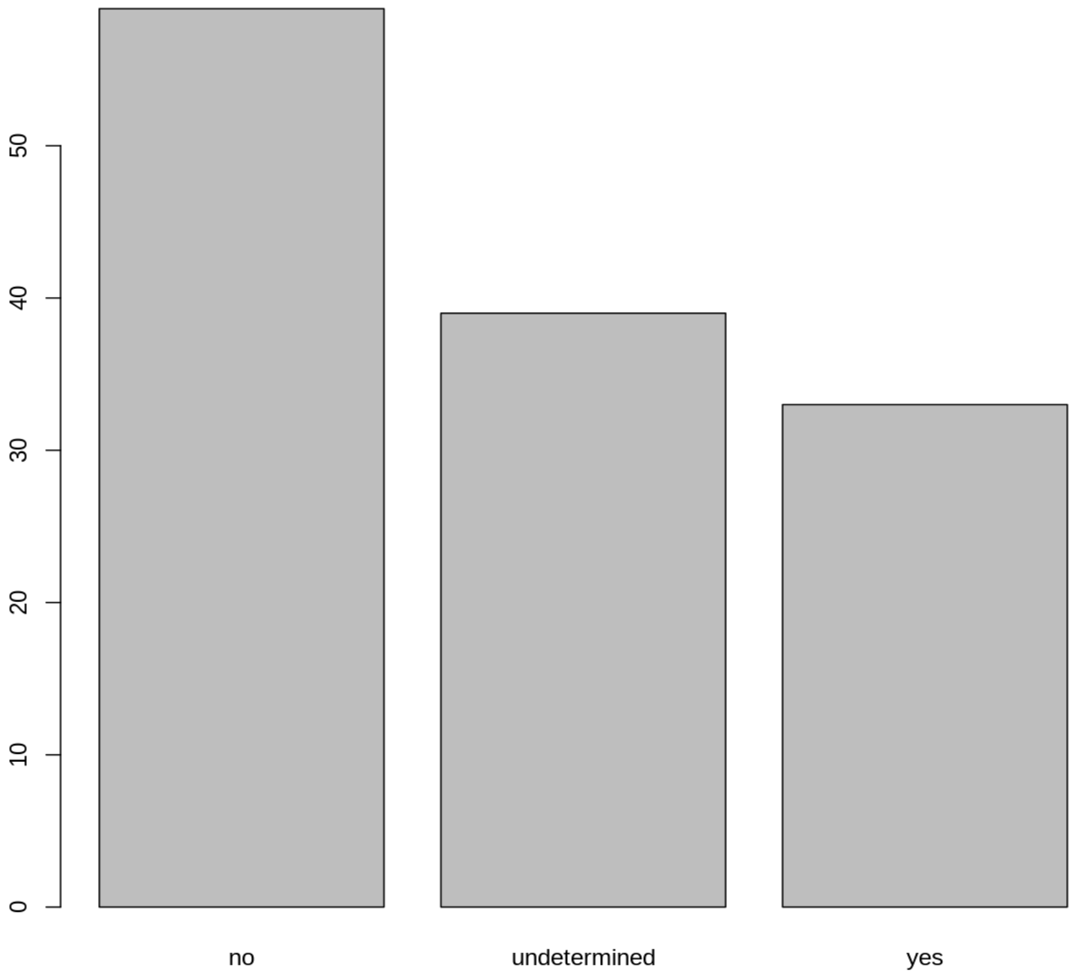
```
 [76] <NA> yes  no   no   yes  no   no   yes  no   yes  no   no  
<NA> yes  yes 
 [91] yes  yes  yes  no   no   no   no   yes  no   no   yes  yes  no  
<NA> no  
[106] no   <NA> no   no   <NA> no   <NA> <NA> no   no   no   no   yes 
no   no  
[121] no   no   no   no   no   no   no   no   no   yes  <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 in addition to “no”s and “yes”s, there are 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 be 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)
```



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”)?

Key Points

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

Data Wrangling with dplyr

Overview

Teaching: 30 min
Exercises: 20 min
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 I create new columns or remove existing columns from a data frame?

Objectives

- Describe the purpose of the **`dplyr`** package.
- 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.
- Export a data frame to a csv file.

Learning dplyr

dplyr is a package for making tabular data wrangling easier by using a limited set of functions that can be combined to extract and summarize insights from your data. Like **readr**, **dplyr** is also part of the **tidyverse**. These packages were loaded in R's memory when we called `library(tidyverse)` earlier.

The package **dplyr** provides easy tools for the most common data wrangling tasks. It is built to work directly with data frames. To learn more about **dplyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet.

To make sure that everyone is using 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

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, separated by commas. Alternatively, if you are selecting columns adjacent to each other, you can use a `:` to select a range of columns, read as "select columns from __ *to* __."

```
# to select columns throughout the data frame
select(interviews, village, no_membrs, months_lack_food)
# to select a series of connected columns
select(interviews, village:respondent_wall_type)
```

To choose rows based on specific criteria, we can use the `filter()` function. The arguments after the data frame specify the condition(s) we want for our final data frame to adhere to (e.g., village name is Chirodzo). We can chain a series of conditions together using commas between each condition.

```
# one condition
filter(interviews, village == "Chirodzo")
# A tibble: 39 x 14
   key_ID village interview_date      no_membrs years_liv respondent_wall…
rooms
    <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>
<dbl>
 1      8 Chirod… 2016-11-16 00:00:00        12        70 burntbricks        3
 2      9 Chirod… 2016-11-16 00:00:00         8         6 burntbricks        1
 3     10 Chirod… 2016-12-16 00:00:00        12        23 burntbricks        5
 4     34 Chirod… 2016-11-17 00:00:00         8        18 burntbricks        3
 5     35 Chirod… 2016-11-17 00:00:00         5        45 muddaub            1
 6     36 Chirod… 2016-11-17 00:00:00         6        23 sunbricks          1
 7     37 Chirod… 2016-11-17 00:00:00         3         8 burntbricks        1
 8     43 Chirod… 2016-11-17 00:00:00         7        29 muddaub            1
 9     44 Chirod… 2016-11-17 00:00:00         2         6 muddaub            1
10     45 Chirod… 2016-11-17 00:00:00         9         7 muddaub            1
# … with 29 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>
# multiple conditions
filter(interviews, village == "Chirodzo", rooms > 1, no_meals > 2)
# A tibble: 10 x 14
   key_ID village interview_date      no_membrs years_liv respondent_wall…
rooms
    <dbl> <chr>   <dttm>                  <dbl>     <dbl> <chr>          <dbl>
 1     10 Chirod… 2016-12-16 00:00:00        12        23 burntbricks        5
 2     49 Chirod… 2016-11-16 00:00:00         6        26 burntbricks        2
 3     52 Chirod… 2016-11-16 00:00:00        11        15 burntbricks        3
 4     56 Chirod… 2016-11-16 00:00:00        12        23 burntbricks        2
 5     65 Chirod… 2016-11-16 00:00:00         8        20 burntbricks        3
 6     66 Chirod… 2016-11-16 00:00:00        10        37 burntbricks        3
 7     67 Chirod… 2016-11-16 00:00:00         5        31 burntbricks        2
 8     68 Chirod… 2016-11-16 00:00:00         8        52 burntbricks        3
 9    199 Chirod… 2017-06-04 00:00:00         7        17 burntbricks        2
10    200 Chirod… 2017-06-04 00:00:00         8        20 burntbricks        2
# … 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>
```

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 == "Chirodzo")
interviews_ch <- select(interviews2, village:respondent_wall_type)
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually.

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

```
interviews_ch <- select(filter(interviews, village == "Chirodzo"),
                        village:respondent_wall_type)
```

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**.

```
interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)
# A tibble: 39 x 5
   village  interview_date      no_membrs years_liv
respondent_wall_type
   <chr>    <dttm>                  <dbl>     <dbl> <chr>
 1 Chirodzo 2016-11-16 00:00:00        12        70 burntbricks
 2 Chirodzo 2016-11-16 00:00:00         8         6 burntbricks
 3 Chirodzo 2016-12-16 00:00:00        12        23 burntbricks
 4 Chirodzo 2016-11-17 00:00:00         8        18 burntbricks
 5 Chirodzo 2016-11-17 00:00:00         5        45 muddaub
 6 Chirodzo 2016-11-17 00:00:00         6        23 sunbricks
 7 Chirodzo 2016-11-17 00:00:00         3         8 burntbricks
 8 Chirodzo 2016-11-17 00:00:00         7        29 muddaub
 9 Chirodzo 2016-11-17 00:00:00         2         6 muddaub
10 Chirodzo 2016-11-17 00:00:00         9         7 muddaub
# … with 29 more rows
```

In the above code, we use the pipe to send the `interviews` dataset first through `filter()` to keep rows where `village` is “Chirodzo”, 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 == "Chirodzo"`, *then* we `select` columns `no_membrs` and `years_liv`.

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

```
interviews_ch <- interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)

interviews_ch
# A tibble: 39 x 5
   village  interview_date      no_membrs years_liv
respondent_wall_type
   <chr>    <dttm>                  <dbl>     <dbl> <chr>
```

```
 1 Chirodzo 2016-11-16 00:00:00        12          70 burntbricks
 2 Chirodzo 2016-11-16 00:00:00         8           6 burntbricks
 3 Chirodzo 2016-12-16 00:00:00        12          23 burntbricks
 4 Chirodzo 2016-11-17 00:00:00         8          18 burntbricks
 5 Chirodzo 2016-11-17 00:00:00         5          45 muddaub
 6 Chirodzo 2016-11-17 00:00:00         6          23 sunbricks
 7 Chirodzo 2016-11-17 00:00:00         3           8 burntbricks
 8 Chirodzo 2016-11-17 00:00:00         7          29 muddaub
 9 Chirodzo 2016-11-17 00:00:00         2           6 muddaub
10 Chirodzo 2016-11-17 00:00:00         9           7 muddaub
# … with 29 more rows
```

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`.

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 of the `is.na()` function. Thus, if `is.na()` returns a value of `TRUE` (because the `memb_assoc` is missing), the `!` symbol negates this and says we only want values of `FALSE`, where `memb_assoc` **is not** missing.

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!

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.

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))
`summarise()` has grouped output by 'village'. You can override using
the `.groups` argument.
# 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
```

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

```
interviews %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs)) %>%
    arrange(mean_no_membrs)
`summarise()` has grouped output by 'village'. You can override using
the `.groups` argument.
# A tibble: 9 x 3
# Groups:   village [3]
  village  memb_assoc mean_no_membrs
  <chr>    <chr>               <dbl>
1 Chirodzo NA                   5.08
2 God      NA                   6
3 Ruaca    NA                   6.22
4 God      no                   7.13
5 Ruaca    no                   7.18
6 Chirodzo yes                  7.82
7 God      yes                  8
8 Chirodzo no                   8.06
9 Ruaca    yes                  9.5
```

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 %>%
    group_by(village, memb_assoc) %>%
    summarize(mean_no_membrs = mean(no_membrs)) %>%
    arrange(desc(mean_no_membrs))
`summarise()` has grouped output by 'village'. You can override using
the `.groups` argument.
# A tibble: 9 x 3
# Groups:   village [3]
  village  memb_assoc mean_no_membrs
  <chr>    <chr>               <dbl>
1 Ruaca    yes                  9.5 
2 Chirodzo no                   8.06
3 God      yes                  8   
4 Chirodzo yes                  7.82
5 Ruaca    no                   7.18
6 God      no                   7.13
7 Ruaca    NA                   6.22
8 God      NA                   6   
9 Chirodzo NA                   5.08
```

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
```

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?
2. Use `group_by()` and `summarize()` to find the mean, min, and max number of household members for each village.
3. What was the largest household interviewed in each month?

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.

Imagine you wanted to use the reduced size dataset from the piping exercise above in a new project. If you remember, we used the following code to filter only those records where the village name was "Chirodzo" and select only the variables `village` through `respondent_wall_type`, and stored the results in a new object called `interviews_ch`.

```
interviews_ch <- interviews %>%
    filter(village == "Chirodzo") %>%
    select(village:respondent_wall_type)
interviews_ch
# A tibble: 39 x 5
   village  interview_date      no_membrs years_liv
respondent_wall_type
   <chr>    <dttm>                  <dbl>     <dbl> <chr>
 1 Chirodzo 2016-11-16 00:00:00        12        70 burntbricks
 2 Chirodzo 2016-11-16 00:00:00         8         6 burntbricks
 3 Chirodzo 2016-12-16 00:00:00        12        23 burntbricks
 4 Chirodzo 2016-11-17 00:00:00         8        18 burntbricks
 5 Chirodzo 2016-11-17 00:00:00         5        45 muddaub
 6 Chirodzo 2016-11-17 00:00:00         6        23 sunbricks
 7 Chirodzo 2016-11-17 00:00:00         3          8 burntbricks
 8 Chirodzo 2016-11-17 00:00:00         7         29 muddaub
 9 Chirodzo 2016-11-17 00:00:00         2          6 muddaub
10 Chirodzo 2016-11-17 00:00:00         9          7 muddaub
# … with 29 more rows
```

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

```
write_csv(interviews_ch, file = "data_output/interviews_ch.csv")
```

Key Points

- Use the `dplyr` package to manipulate data frames.
- Use `select()` to choose variables from a data frame.
- 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.

Additional Resources

1. Keyboard shortcuts, etc. - RStudio cheatsheet about the RStudio IDE
2. R Style Guides
   a. Google's Style Guide
   b. Jean Fan's Style Guide
   c. Tidyverse's Style Guide
3. Object-oriented programming.
   https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects
4. Data transformation with dplyr cheatsheet.