An Introduction to R for Quantitative Methods

Brian Habing, Haley Jeppson

2023-04-04

Contents

PREFACE

This book provides an introduction to R programming and serves as a companion to the data visualization workshop at NCME. It pulls from many sources, including An Introduction to R for Quantitative Methods, R for Data Science, and What They Forgot to Teach You About R.

Conventions

In this book, we alternate between regular text (like this), samples of code that you can type and run yourself, and the output of that code. In the main text, references to objects or other things that exist in the R language or in your R project—like tables of data, variables, functions, and so on—will also appear in `a monospaced` or `"typewriter" typeface`. Code you can type directly into R at the console will be in gray boxes, and also monospaced. Like this:

```
my_numbers <- c(1, 1, 4, 1, 1, 4, 1)
```

If you type that line of code into R's console it will create a thing called `my_numbers`. Doing this doesn't produce any output, however. When we write code that also produces output at the console, we will first see the code (in a gray box) and then the output in a monospaced font against a gray background. Here we add two numbers and see the result:

```
4 + 1
```

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

Two further notes about how to read this. First, by default in this book, anything that comes back to us at the console as the result of typing a command will be shown prefaced by two hash characters (`##`) at the beginning of each line of output. This is to help distinguish it from commands we type into the console. You will not see the hash characters at the console when you use R.

Second, both in the book *and* at the console, if the output of what you did results in a series of elements (like numbers, or observations from a variable, and so on) you will often see output that includes some number in square brackets at the beginning of the line. It looks like this: `[1]`. This is not part of the output itself, but just a counter or index keeping track of how many items have been printed out so far. In the case of adding `4 + 1` we got just one, or `[1]`, thing back—the number five. If there are more elements returned as the result of some instruction or command, the counter will keep track of that on each line. In this next bit of code we will tell R to show us the lower-case letters of the alphabet:

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

You can see the counter incrementing on each line as it keeps count of how many letters have been printed.

Chapter 1

An Introduction

Getting Started in R with RStudio

The book is designed for you to follow along in an active way, writing out the examples and experimenting with the code as you go. You will need to install some software first.

R

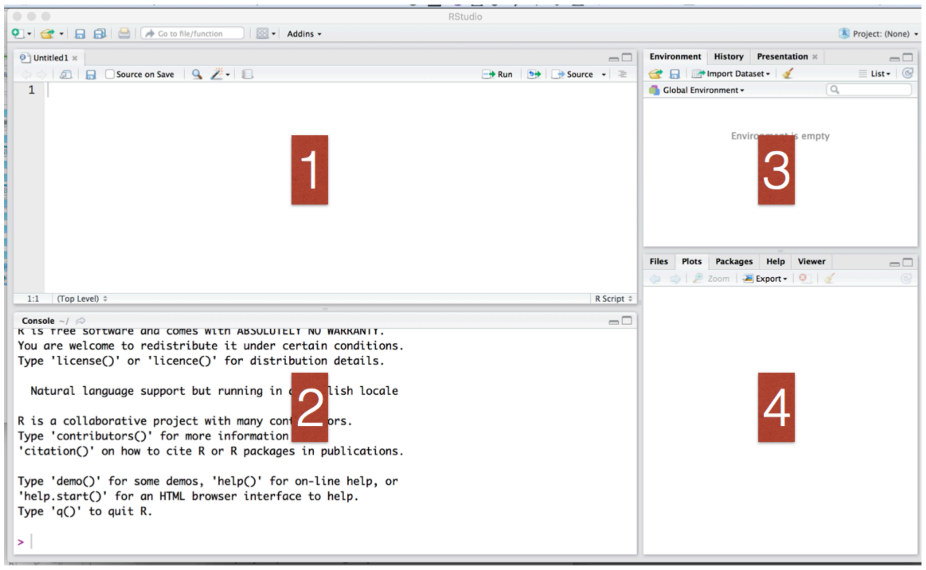
R is based on the statistical programming language S and can be downloaded for free from www.r-project.org. Currently, to do so, choose: CRAN, then a mirror site in the US, then Download R for Windows, then base, then "Download R 4.3.0 for Windows". This page also has links to FAQs and other information about R.

R itself is a relatively small application with next to no user interface. Everything works through a command line, or console. At its most basic, you launch it from your Terminal application (on a Mac) or Command Prompt (on Windows) by typing `R`. Once launched, R awaits your instructions at a command line of its own, denoted by the right angle bracket symbol, `>`. When you type an instruction and hit return, R interprets it and sends any resulting output back to the console. But although a plain text file and a command line is the absolute minimum you need to work with R, it is a rather spartan arrangement. We can make life easier for ourselves by using RStudio.

RStudio

RStudio is an IDE (integrated development environment), and a convenient interface for R. Think of R as like a car's engine and RStudio as a car's dashboard. You can download and install Rstudio from the official Rstudio website. When launched, it starts up an instance of R's console inside of itself. It also conveniently pulls together various other elements to help you get your work done. These include the document where you are writing your code, the output it produces, and R's help system. RStudio also knows about RMarkdown, and understands a lot about the R language and the organization of your project.

Once you have both R and Rstudio installed, open Rstudio. You should now see four panels: (1) Source editor, (2) Console window, (3) Environment pane, and (4) Other tabs/panes.



```
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.

  Natural language support but running in a   lish locale

R is a collaborative project with many cont   ors.
Type 'contributors()' for more information
'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.

>
```

Executing code and R script files

You can start coding by typing commands in the **Console panel (2)**. This window is also where the output will appear. For example if you were to type 2+2 and hit return in that window it will return the answer 4.

When using the Console window in RStudio or R, the up and down-arrow on the key-board can be used to scroll through previously entered lines. `history()` will open a window of previously entered commands (which we'll see below after entering some). If the font in this R Console is too small, or if you dislike the color or font, you can change it by selecting "Global Options" under the "Tools" menu and clicking on the "Appearance" tab in the pop up window.

Because this Console window is used for so many things it often fills up quickly — and so, if you are doing anything involving a large number of steps, it is often easiest to type them in a script first, which can be viewed in the **Source editor (1)**.

You can create a new script by clicking on the "File" menu and selecting "New File" then "R Script". A script is a collection of commands that you can save and run again later. To run a command, click in a given line or highlight the text and hit `Ctrl+Enter`, or click the "Run" button at the top of the script window. You can save your scripts for later use.

1.1 Objects

At the heart of R are the various objects that you enter. An object could be data (in the form of a single value, a vector, a matrix, an array, a list, or a data frame) or a function that you created. Objects are created by assigning a value to the objects name using either `<-` or `=`. For example

```
x <- 3
```

All R statements where you create objects, **assignment statements**, have the same form:

```
object_name <- value
```

When reading that code say "object name gets value" in your head.

You will make lots of assignments, and `<-` is a pain to type. You can save time with **RStudio's keyboard shortcut: Alt + -** (the minus sign). Notice that RStudio automatically surrounds `<-` with spaces, which is a good code formatting practice. Code is miserable to read on a good day, so giveyoureyesabreak and use spaces.

If you run `x <- 3` in your local console (at the `>` prompt), R will only give you another prompt. This is because you merely assigned the value; you didn't ask R to do anything with it. Typing

```
x
```

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

will now return the number 3, the value of `x`. R is case sensitive, so entering `X <- 5` will create a separate object:

```
X <- 5
X
```

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

If you reassign an object, say `X <- 7`, the original value is over-written:

```
X <- 7
X
```

```
## [1] 7
```

If you attempt to use the name of a built in function or constant (such as `c()`, `t()`, `t.test()`, or `pi()`) for one of your variable names, you will likely find that future work you are trying to do gives unexpected results. Notice in the name of `t.test()` that periods are allowed in names of objects. Other symbols (except numbers and letters) are not allowed.

Your workspace

Note that the up-arrow and `history()` will now show us the commands we entered previously. This set of all of your created objects (your Workspace) is not saved by default when you exit R, and this is probably a good thing! Attachment to your workspace indicates that you have a **non-reproducible** workflow. Everything that really matters should be achieved through code that you save in your script, and so any individual R process and the associated workspace is disposable.

Data types in R

R has a veriety of data types:

- **logical**: boolean values
  - ex. `TRUE` and `FALSE`
- **double**: floating point numerical values (default numerical type)
  - ex. `1.335` and `7`
- **integer**: integer numerical values (indicated with an L)
  - ex. `7L` and `1:3`
- **character**: character string
  - ex. `"hello"`

- **lists**: 1d objects that can contain any combination of R objects
- & more, but we won't be focusing on those yet

1.2 Arithmetic and Parentheses

Using R can be a lot like using a calculator. All of the basic arithmetic operations work in R:

```
X - x
```

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

```
7 - 3
```

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

will both return the value 4, one by performing the arithmetic on the objects, and the other on the numbers. The other basic mathematical operators are:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

Figure 1.1: Image via Jenny Bryan's 'What They Forgot to Teach You About R'

- `^` exponentiation
- `%*%` matrix multiplication

R will often try to do the "common-sense" thing when using arithmetic arguments. For example, if `Y` is a vector or matrix of values, then `Y + 4` will add `4` to each of the values in `Y`. (So the vector `3, 2, 5` would become `7, 6, 9`).

Parentheses work as usual in mathematical statements, but they do not imply multiplication.

```
#X(x+5)
X*(x+5)
```

```
## [1] 56
```

Notice that the former returns an error about looking for a function called `X`, while the latter does the arithmetic to return the value `40`.

The other use of parentheses in R are to indicate that you attempting to run a function, and, if the function has any options it will contain those. The command:

```
rnorm(10)
```

```
##  [1]  0.2631387  1.8752579  1.1486833 -1.1221379 -0.4733657  0.4149713
##  [7]  0.9768609  0.3285278  0.2062373 -1.4611283
```

runs the function `rnorm()` with the argument `10`. In this case it is generating a random sample of 10 values from a normal distribution.

1.3 Help!

To see this, we could run the help function on that command.

```
help(rnorm)
```

A shortcut, `?rnorm`, would also work.

Every help file in R begins with a brief description of the function (or group of functions) in question, followed by all of the possible options (a.k.a. arguments) that you can provide. In this case the value `n` (the number of observations) is required. Notice that all of the other options are shown as being = some value – this indicates the defaults those values take if you do not enter them. The sample we generated above thus has mean 0 and standard deviation 1.

Below the list of arguments are a brief summary of what the function does (called the *Details*), a list of the *Value* (or values) returned by running the function, the *Source* of the algorithm, and general *References* on the topic. *See Also* is often the most useful part of the help for a function as it provides a list of related functions. Finally, there are some *Examples* that you can cut and paste in to observe the function in action.

1.4 Functions

Functions are (most often) verbs, followed by what they will be applied to in parentheses:

```
do_this(to_this)
do_that(to_this, to_that, with_those)
```

It is always safest to enter the values of functions using the names of the arguments:

```
rnorm(10, sd = 4)
```

```
##  [1]  1.79026562  2.37694147 -0.37064936 -3.03166536 -3.97063989  0.07295468
##  [7] -0.11129388  5.44160501 -0.11636767  1.02287144
```

rather than trusting that the argument you want happens to be first in the list:

```
rnorm(10, 4)
```

```
##  [1] 3.850148 3.315667 3.199099 3.291708 5.273520 3.143993 3.399510 3.976144
##  [9] 3.289327 3.777684
```

Notice the former puts 4 in for the standard deviation, while the latter is putting it in for the second overall argument, the mean (as seen in the help file).

Note that these values we generated have not been saved as an object, and exist solely on the screen. To save the values we could have assigned the output of our function to an object.

```
normal.sample <- rnorm(50)
normal.sample
```

```
##  [1]  1.475007847 -0.046271038 -0.264772708 -2.337407124 -1.097348304
##  [6] -0.369543244 -0.623268491 -0.180579446 -0.414145243 -2.559928172
## [11]  0.640074953 -2.941827224 -0.774473114 -1.309581801  0.312178833
## [16]  0.639891373  0.431923522 -0.230329421  0.463993505  0.840445249
## [21]  0.162522267  1.712076312  0.035379280 -0.056334717 -0.364903523
## [26] -0.468979153 -1.169226700  1.770683649  0.344279934  0.953177692
## [31] -0.566627634  0.283804724 -0.053347178 -1.704057689 -1.092606093
## [36] -0.003955208 -0.038440241 -0.588619862  0.806658305 -1.853666310
## [41]  0.808169096 -2.448769237 -0.123230657  0.307448021  0.455226519
## [46] -0.140054024  0.235504886  0.989265715 -0.937726108 -1.017910756
```

Common statistical functions

A few common statistical functions include:

- `mean()` find the mean
- `median()` find the median
- `sd()` find the standard deviation

- `var()` find the variance
- `quantile()` find the quantiles (percentiles);
  - requires the data and the percentile you want
  - e.g. `quantile(normal.sample, .5)` is the median
- `max()` find the maximum
- `min()` find the minimum
- `summary()` find the 5-number summary
- `hist()` construct a histogram
- `boxplot()` construct a boxplot
- `qqnorm()` construct a normal quantile-quantile plot
- `qqline()` add the line to a normal quantile-quantile plot

Trying a few of these out (like `mean(normal.sample)`) will show us the descriptive statistics and basic graphs for a sample of size 50 from a normal population with mean 0 and standard deviation 1. (Using up arrow can make it quicker to try several in a row.)

As we will see in more detail later, it is possible to create your own functions by using the function function. This one creates a simple measure of skewness.

```r
Skew <- function(x){
    (mean(x) - median(x))/sd(x)}
```

Note that braces `{ }` in R are used to group several separate commands together, and also occur when using programming commands like loops or if-then statements. They work the same as parentheses in arithmetic expressions.

After entering or new function, it works like any built in function, except that it appears in our objects list.

```r
# Skew
# Skew()
Skew(normal.sample)
```

```
## [1] -0.1451548
```

Common mathematical functions

There are also a number of mathematical functions as well. Ones common in statistical applications include:

- `sqrt()` square root
- `exp()` exponent (e to the power)
- `log()` the natural logarithm by default
- `abs()` absolute values
- `floor()` round down
- `ceiling()` round up
- `round()` round to the nearest (even if .5)

1.5 Vectors, Matrices, and Arrays

Vectors

The output from `rnorm()` is different from the `X` and `x` we created as it contains more than just a single value - they are vectors. While we can think of them as vectors in the mathematical sense, we can also think of a vector as simply listing the values of a variable.

Vectors in R are created using the `c()` function (as in concatonate). Thus,

```
Y <- c(3, 2, 5)
Y
```

```
## [1] 3 2 5
```

```
Y + 4
```

```
## [1] 7 6 9
```

```
Y * 2
```

```
## [1]  6  4 10
```

creates a vector of length three (and we can verify that arithmetic works on it componentwise). Given two vectors arithmetic is also done componentwise:

```
Z <- c(1, 2, 3)
Y + Z
```

```
## [1] 4 4 8
```

```
Y * Z
```

```
## [1]  3  4 15
```

Other functions are also evaluated component-wise (if possible):

```
sqrt(Z)
```

```
## [1] 1.000000 1.414214 1.732051
```

Multiple vectors can be combined together by using the `c()` function:

```
YandZ <- c(Y, Z)
YandZ
```

```
## [1] 3 2 5 1 2 3
```

However, when asked to combine vectors of two different types, R will try to force them to be of the same type:

```
nums <- c(1, 2, 3)
nums
```

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

```
lttrs <- c("a", "b", "c")
lttrs
```

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

```
c(nums, lttrs)
```

```
## [1] "1" "2" "3" "a" "b" "c"
```

```
# c(nums, lttrs) + 2
```

Once we have our desired vector, an element in the vector can be referred to by using square brackets:

```
YandZ[2]
```

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

```
YandZ[2:4]
```

```
## [1] 2 5 1
```

```
YandZ[c(1, 4:6)]
```

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

By using the `c()` function, and the `:` to indicate a sequence of numbers, you can quickly refer to the particular portion of the data you are concerned with.

Matrices and arrays

Matrices (two-dimensional) and arrays (more than two dimensions) work similarly - they use brackets to find particular values, and all the values in an array or matrix must be of the same type (e.g. numeric, character, or factor). In the case of matrices, the first values in the brackets indicates the desired rows, and the ones after the comma indicate the desired columns.

```
Xmat <- matrix(c(3, 2, 5, 1, 2, 3),
               ncol = 3, byrow = TRUE)
Xmat
```

```
##      [,1] [,2] [,3]
## [1,]    3    2    5
## [2,]    1    2    3
```

```
Ymat <- rbind(Y, Z)
Ymat
```

```
##   [,1] [,2] [,3]
## Y    3    2    5
## Z    1    2    3
```

```
Xmat[1, 2:3]
```

```
## [1] 2 5
```

```
Zmat <- cbind(nums, lttrs)
Zmat
```

```
##      nums lttrs
## [1,] "1"  "a"
## [2,] "2"  "b"
## [3,] "3"  "c"
```

In the above code, `matrix()` is the function to form a vector into a matrix, `rbind()` places multiple vectors (or matrices) side-by-side as the rows of a new matrix (if the dimensions match), and `cbind()` does the same for columns.

1.6 Data Frames, Lists, and Attributes

Data frames

In many cases the data set we wish to analyze will not have all of the rows or columns in the same format. This type of data is stored in R as a **data frame**. A data frame is a rectangular collection of variables (in the columns) and observations (in the rows).

`scdata.txt` is one such data set, and it can be found in the course materials (its description is found in `scdata.pdf`). The data can be read in using code similar to the below (assuming a similar file structure).

```
sctable <- readr::read_table("data/scdata.txt")
```

The function `read_table()` reads in our file as a **tibble**, unlike `read.table()` which reads in the file as R's traditional `data.table`. Tibbles are data frames, but more opinionated, and they tweak some older behaviors to make working in the tidyverse a little easier.

There are a few good reasons to favor **{readr}** functions over the base equivalents:

- They are typically much faster than their base equivalents. Long running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try `data.table::fread()`. It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.
- They produce tibbles, they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
- They are more reproducible. Base R functions inherit some behavior from your operating system and environment variables, so import code that

works on your computer might not work on someone else's.

- Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames.

Some older functions don't work with tibbles. If you encounter one of these functions, use `as.data.frame()` to turn a tibble back to a data.frame. The main reason that some older functions don't work with tibble is the `[` function. With base R data frames, `[` sometimes returns a data frame, and sometimes returns a vector. With tibbles, `[` always returns another tibble.

Inspecting objects

To inspect the data without needing to print out the entire data set, we can try out the following commands:

- `head()`
- `tail()`
- `summary()`
- `str()`
- `dim()`

For example, use `head()` and `tail()` to see the first and last rows. This lets you check the variable names as well as the number of observations successfully read in.

```
head(sctable)
```

```
## # A tibble: 6 x 27
##   County Region Births Death InfMort Minor~1 Over65 PopChng PopDens Urban Income
##   <chr>  <chr>   <dbl> <dbl>   <dbl>   <dbl>  <dbl>   <dbl>   <dbl> <dbl>  <dbl>
## 1 Abbev~ Upsta~   12.5  10.3    15.1    31.7   14.7     9.7    51.5  23.4  32635
## 2 Aiken  Midla~   12.1   9.8     9.6    28.6   12.8    17.8   133.   60.9  37889
## 3 Allen~ LowCo~   14.8  12.1    18.5    72.6   12.7    -4.4    27.5  59    20898
## 4 Ander~ Upsta~   13    10.7    10.3    18.4   13.7    14.2   231.   58.3  36807
## 5 Bambe~ Midla~   11.5   9.9    21.7    63.5   13.9    -1.4    42.4  45.7  24007
## 6 Barnw~ Midla~   14    10.9    21.4    44.8   12.6    15.7    42.8  14.9  28591
## # ... with 16 more variables: ConsInc <dbl>, FarmInc <dbl>, ManIncom <dbl>,
## #   RetInc <dbl>, FdStmps <dbl>, MoblHms <dbl>, NoCar <dbl>, PlumProb <dbl>,
## #   PoorChild <dbl>, Unemp <dbl>, Coll4 <dbl>, Crime <dbl>, HSGrad <dbl>,
## #   JuvDel <dbl>, MVDeath <dbl>, SchlSpnd <dbl>, and abbreviated variable name
## #   1: Minority
```

```
tail(sctable)
```

```
## # A tibble: 6 x 27
##   County Region Births Death InfMort Minor~1 Over65 PopChng PopDens Urban Income
##   <chr>  <chr>   <dbl> <dbl>   <dbl>   <dbl>  <dbl>   <dbl>   <dbl> <dbl>  <dbl>
## 1 Saluda Midla~   11.2  11.3     4.7    34.2   14.5    16.7    42.5  18.7  35774
## 2 Spart~ Upsta~   13.1   9.8     7.9    24.9    12.5     11.9   313.   64.8  37579
## 3 Sumter Midla~   15.6   8.9     8.5    50      11.2      3.3   157.   62.1  33278
## 4 Union  Upsta~   10.7  13.2    12.8    32.2    15.6     -1.5    58.1  35.7  31441
## 5 Willi~ Peedee   12.6  11.8     8.8    67.3    13        1.1    39.8  15.1  24214
## 6 York   Upsta~   13.4   7.7     7.6    22.7    10.4     25.2   241.   64.3  44539
## # ... with 16 more variables: ConsInc <dbl>, FarmInc <dbl>, ManIncom <dbl>,
## #   RetInc <dbl>, FdStmps <dbl>, MoblHms <dbl>, NoCar <dbl>, PlumProb <dbl>,
## #   PoorChild <dbl>, Unemp <dbl>, Coll4 <dbl>, Crime <dbl>, HSGrad <dbl>,
## #   JuvDel <dbl>, MVDeath <dbl>, SchlSpnd <dbl>, and abbreviated variable name
## #   1: Minority
```

Extracting parts of objects

For object `x`, we can extract parts in the following manner (`rows` and `columns` are vectors of indices):

```
x$variable
x[, "variable"]
x[rows, columns]
x[1:5, 2:3]
x[c(1,5,6), c("County", "Region")]
x$variable[rows]
```

Many of these extraction methods access the rows and columns of a data frame by treating it similarly to a matrix:

```
County1 <- sctable[1, ]
Birth.Death <- sctable[ , 3:4]
```

This simplicity sometimes causes trouble though. While `Birth.Death` may look on the screen like it is a matrix, it is still a data frame and many functions which use matrix operations (like matrix multiplication) will give an error. The `attributes()` function will show us the true status of our object (it returns `NULL` for a numeric vector and the dimensions if a matrix):

```
Birth.Death
```

```
## # A tibble: 46 x 2
##    Births Death
##     <dbl> <dbl>
##  1   12.5  10.3
##  2   12.1   9.8
##  3   14.8  12.1
##  4   13    10.7
##  5   11.5   9.9
##  6   14    10.9
##  7   15.8   7.8
##  8   14.4   6.6
##  9   11.5  11.1
## 10   14.5   8.5
## # ... with 36 more rows
```

```
attributes(Birth.Death)
```

```
## $names
## [1] "Births" "Death"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
BD.matrix <- as.matrix(Birth.Death)
attributes(BD.matrix)
```

```
## $dim
## [1] 46  2
##
## $dimnames
## $dimnames[[1]]
## NULL
##
## $dimnames[[2]]
## [1] "Births" "Death"
```

The `$` is used to access whatever corresponds to an entry in the names attribute:

```
Birth.Death$Births
```

```
##  [1] 12.5 12.1 14.8 13.0 11.5 14.0 15.8 14.4 11.5 14.5 13.3 12.3 12.1 12.3 13.0
## [16] 12.4 15.6 12.9 10.3 12.8 15.3 12.7 14.0 12.6 13.9 12.7 14.3 12.8 12.2 11.6
## [31] 12.9 14.0  7.6 13.8 11.9 12.5 11.7 13.8 11.2 13.3 11.2 13.1 15.6 10.7 12.6
## [46] 13.4
```

This is particularly useful when trying to access a portion of the output of a function for later use. For example, later we will see a method of doing statistical inference called the t-test. In R, this is performed by the function `t.test()` which can create a great deal of output on the screen.

```
t.test(normal.sample)
```

```
##
##  One Sample t-test
##
## data:  normal.sample
## t = -1.631, df = 49, p-value = 0.1093
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.5406289  0.0562202
## sample estimates:
##  mean of x
## -0.2422044
```

If you only want the part called the “p-value” for later use we can pull that out of the output.

```
t.out <- t.test(normal.sample)
attributes(t.out)
```

```
## $names
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
##
## $class
## [1] "htest"
```

```
t.out$p.value
```

```
## [1] 0.1093039
```

We could then save the resulting value as part of a vector or matrix of other p-values, for example.

The `$` is also used to access named parts of lists, which we will see can be used to store a variety of kinds of information in a single object.

1.7 Packages

The ability to “easily” write functions in R has lead to an explosion of procedures that researchers have written and made available to the public, typically as an R package. An R package is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R. For example, the **MASS** package contains all of the functions corresponding to the Springer text *Modern Applied Statistics with S* by Venables and Ripley.

While some of these are automatically included with the basic installation of R, most are not and can installed with the `install.packages()` function.

```
install.packages("package_name")
```

When you run the code to install a package on your own computer, R will download the packages from CRAN and install them on to your computer.

If you have problems installing, make sure that you are connected to the internet, and that https://cloud.r-project.org/ isn’t blocked by your firewall or proxy.

You will not be able to use the functions, objects, and help files in a package until you load it with `library()`.

```
library(package_name)
```

The command `library()` is used to activate a downloaded package and give access to all of its functions and must be done once per session.

```
# help(fractions)
library(MASS)
help(fractions)
fractions(0.5)
```

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

```
fractions(pi)
```

```
## [1] 4272943/1360120
```

```
4272943/1360120
```

```
## [1] 3.141593
```

Required packages

The remainder of this ebook (and the workshop) requires that you install the tidyverse library and several other add-on packages for R. These libraries provide useful functionality that we will take advantage of throughout the book. You can learn more about the tidyverse's family of packages at its website.

To install the necessary packages, type the following lines of code at R's command prompt, located in the console window, and hit return.

```
course_packages <- c("tidyverse", "GGally", "ggrepel", "MASS")

install.packages(course_packages)
```

R should then download and install these packages for you. It may take a little while to download everything.

1.8 RMarkdown

Beyond data analysis, coding, and creating graphics, R and RStudio also allow for the creation of documents using Markdown. Markdown is a particular type of markup language. Markup languages are designed to produce documents from plain text. Some of you may be familiar with LaTeX, another (less human friendly) markup language for creating pdf documents. LaTeX gives you much greater control, but it is restricted to pdf and has a much greater learning curve.

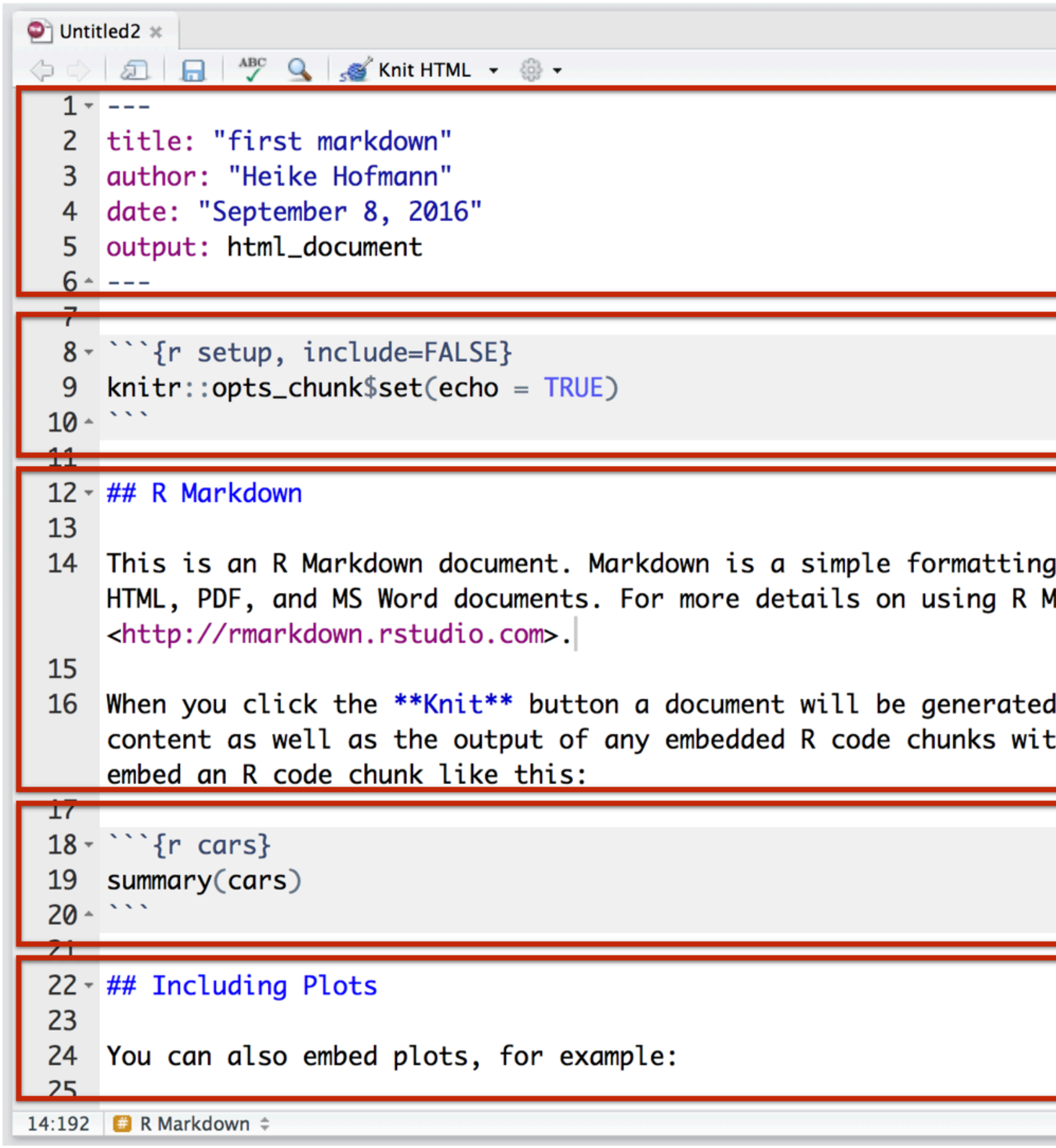
Markdown is becoming a standard and many websites will generate HTML from Markdown (e.g. GitHub, Stack Overflow, reddit, …). It is also relatively easy:

```
*italic*
**bold**
# Header 1
## Header 2
### Header 3
- List item 1
- List item 2
    - item 2a
    - item 2b
1. Numbered list item 1
1. Numbered list item 2
    - item 2a
    - item 2b
```

Have a look at RStudio's RMarkdown cheat sheet.

RMarkdown is an authoring format that enables easy creation of dynamic documents, presentations, and reports from R. It combines markdown syntax with embedded R code chunks that are run so their output can be included in the final document.

Figure 1.2: Artwork by allison horst



~~~
---
title: "first markdown"
author: "Heike Hofmann"
date: "September 8, 2016"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

This is an R Markdown document. Markdown is a simple formatting
HTML, PDF, and MS Word documents. For more details on using R M
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated
content as well as the output of any embedded R code chunks wit
embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

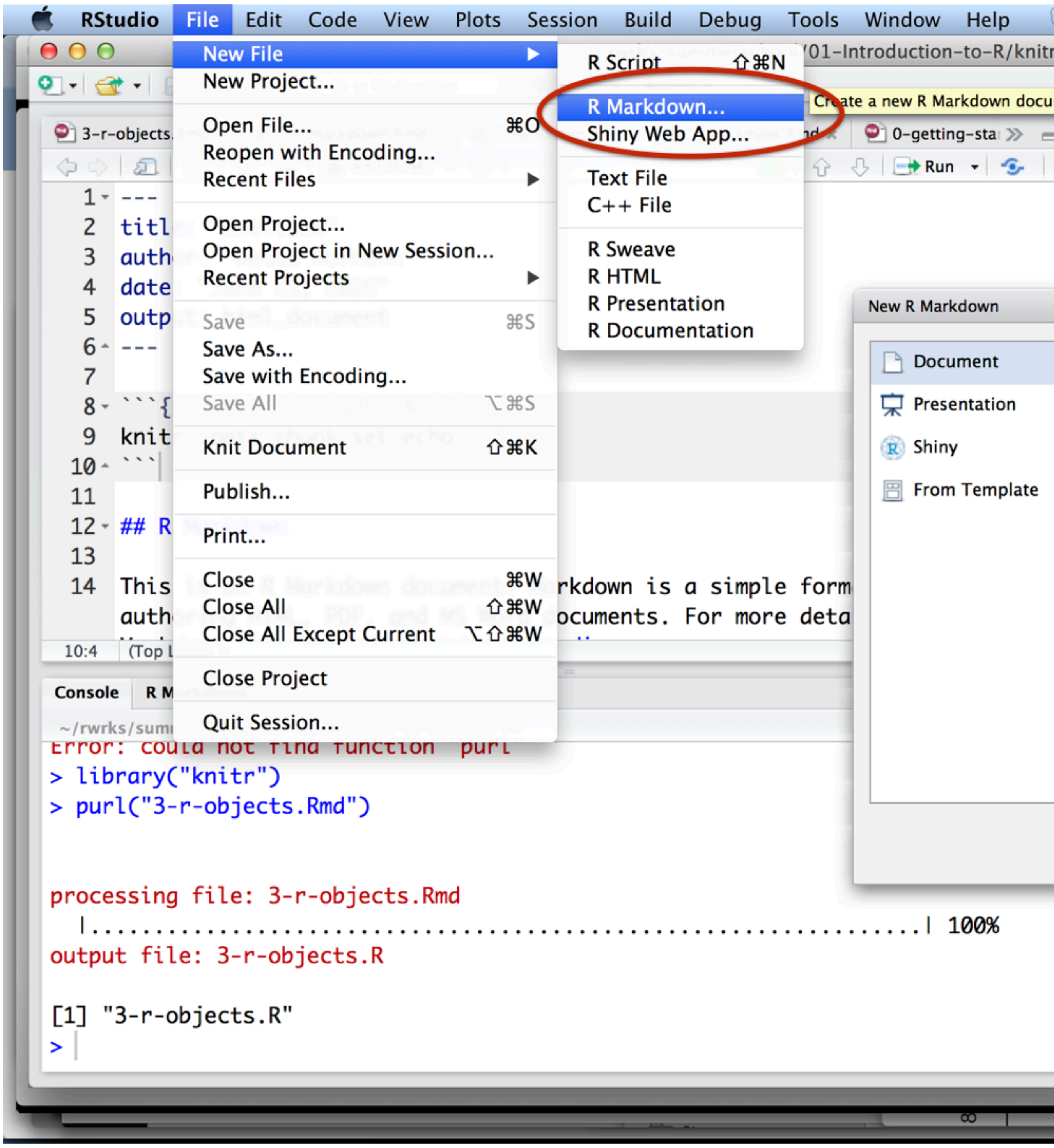
You can also embed plots, for example:

```
```

Most importantly, RMarkdown creates fully reproducible reports since each time you knit the analysis is run from the beginning, encouraging transparency. Collaborators (including your future self) will thank you for integrating your analysis and report.

Exercise: Create your first Rmarkdown.

1. Open RStudio, create a new project.
2. Create a new RMarkdown file and knit it.
3. Make changes to the markdown formatting and knit again (use the RMarkdown cheat sheet)
4. If you feel adventurous, change some of the R code and knit again.



~~~
---
titl
auth
date
outp
---

```{
knit
```


## R

This                                   rkdown is a simple form
auth                                   ocuments. For more deta
~~~

```
Error: could not find function "purl"
> library("knitr")
> purl("3-r-objects.Rmd")


processing file: 3-r-objects.Rmd
  |.................................................................| 100%
output file: 3-r-objects.R

[1] "3-r-objects.R"
>
```

Chapter 2

Graphics

Outline

1. Introduction to ggplot2

- setup
- why ggplot2
- the evolution of a ggplot
- create your first ggplot

2. ggplot2 concepts

- the grammar of ggplot2
- geometrical layers
- statistical layers
- facets
- ggplots as objects

3. Advanced customization

- scales
- coordinates
- labels
- annotations
- themes
- legends
- fonts

4. Extensions

- patchwork
- gganimate

- ggplotly
- ggiraph

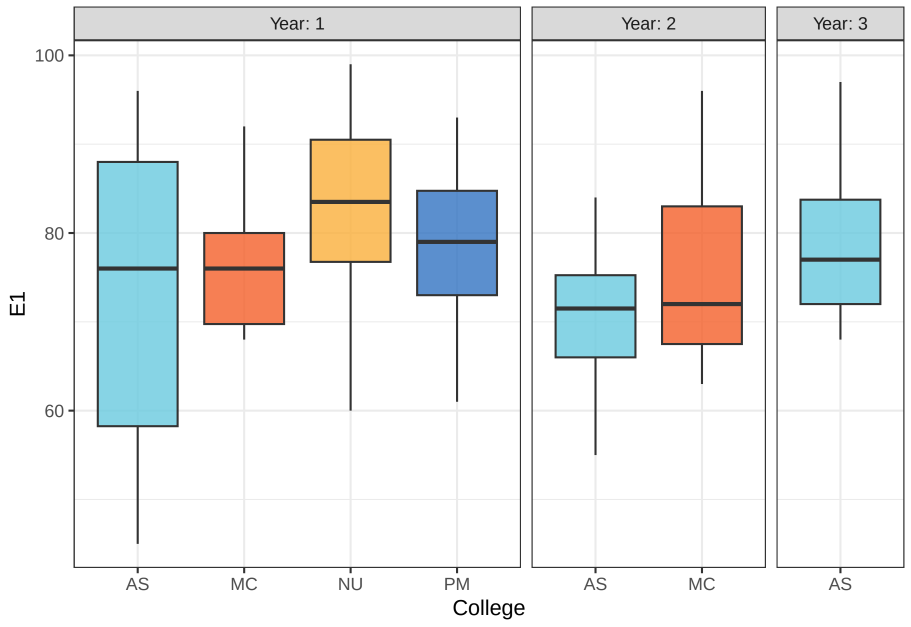
2.1 An example

The data is a stratified sample of 70 students from a large section course. The strata were based on the college the students belonged to (AS = Arts & Sciences, PM = Professional Management, MC = Mass Communications, and NU=Nursing) and their year in school (ranging from 1st to 3rd based on credit hours, and limited based on expectation of having at least 10 students from that college at that grade level). The response variables are their Hmwk = Homework Average and E1 to E3 = their grades on the first three exams.

```r
library(ggplot2)
students <- readr::read_table("data/CourseData.txt")[, -1]
students
```

```
## # A tibble: 70 x 6
##    College  Year  Hmwk    E1    E2    E3
##    <chr>   <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 NU          1  83.8    79    89    59
##  2 NU          1  77.1    60    97    63
##  3 NU          1  94.5    83    88    69
##  4 NU          1  84.3    91    77    72
##  5 NU          1  73.4    68    79    76
##  6 NU          1  94.7    89    96    79
##  7 NU          1  77.7    76    43    84
##  8 NU          1  92.9    92    81    84
##  9 NU          1  95.9    84    85    84
## 10 NU          1  98.2    99    84    85
## # ... with 60 more rows
```

```r
ggplot(students, aes(College, E1)) +
  geom_boxplot(aes(fill = stage(College, after_scale = alpha(fill, 0.8))), show.legend
  facet_grid(~Year, scales = "free", space = "free_x", labeller = labeller(Year = label
  theme_bw() +
  scale_fill_manual(values = c("#69cadf", "#f4602a", "#fbb03b", "#3273c2"))
```


Year: 1
Year: 2
Year: 3
100
80
60
E1
AS
MC
NU
PM
AS
MC
AS
College

Chapter 3

Transforming Data

{dplyr} basics

The book R for Data Science is a *very* helpful reference guide. Chapter 5 covers many of the topics covered in this section, and may be useful as a resource later or to dive deeper into a topic.

Topics:

- Filter rows
  - Comparisons
  - Logical operators
  - Missing values
- Arrange rows
- Select columns
- Add new variables
- Grouped summaries
  - Combining multiple operations with the pipe
  - Missing values
  - Counts
  - Useful summary functions
  - Grouping by multiple variables
  - Ungrouping
- Grouped mutates (and filters)
- Extra: Working with factor variables

3.1 An example

The following example ties several of the above ideas together. Imagine that we have a set of grades from a course that we would like to convert to letter grades using a particular weighting and letter-grade cut-offs.

```
students <- readr::read_table("data/CourseData.txt")[, -1]
head(students)
```

```
## # A tibble: 6 x 6
##   College  Year  Hmwk    E1    E2    E3
##   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 NU          1  83.8    79    89    59
## 2 NU          1  77.1    60    97    63
## 3 NU          1  94.5    83    88    69
## 4 NU          1  84.3    91    77    72
## 5 NU          1  73.4    68    79    76
## 6 NU          1  94.7    89    96    79
```

```
tail(students)
```

```
## # A tibble: 6 x 6
##   College  Year  Hmwk    E1    E2    E3
##   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 MC          2  83.2    69    77    93
## 2 MC          2  95.7    88    89    92
## 3 MC          2  79.3    80    61    97
## 4 MC          2  78.1    69    89    97
## 5 MC          2  99.1    84    89    96
## 6 MC          2  95.8    96   100   100
```

In this particular case we want the weighting to be:

- 20% to Hmwk (col 3)
- 25% to E1 (col 4)
- 25% to E2 (col 5)
- 30% to E3 (col 6)

But those weights could change later. A function can be written to take the data, and the weights and calculate the weighted average. We can then add that score to the students data frame, and add a final column with the actual letter grades on a 90-80-70-60 scale.

```
students <- students %>%
  mutate(Final = .2*Hmwk + .25*E1 + .25*E2 + .3*E3,
         Grade = case_when(
           Final < 60 ~ "F",
           Final < 69 ~ "D",
           Final < 79 ~ "C",
           Final < 89 ~ "B",
           Final >= 90 ~ "A"))

head(students)
```

```
## # A tibble: 6 x 8
##   College  Year  Hmwk    E1    E2    E3 Final Grade
##   <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
## 1 NU          1  83.8    79    89    59  76.5 C
## 2 NU          1  77.1    60    97    63  73.6 C
## 3 NU          1  94.5    83    88    69  82.4 B
## 4 NU          1  84.3    91    77    72  80.5 B
## 5 NU          1  73.4    68    79    76  74.2 C
## 6 NU          1  94.7    89    96    79  88.9 B
```

Chapter 4

Reshaping Data

{tidyr} basics

The book R for Data Science is a *very* helpful reference guide. Chapter 12 covers many of the topics covered in this section, and may be useful as a resource later or to dive deeper into a topic.

Topics:

- Tidy data
- Pivoting
    - Longer
    - Wider
- Separate & unite

4.1 An example

Reshape data into long form:

```r
## load data
library(psych)
data(bfi)
bfi_trim <- bfi %>%
  dplyr::select(matches('^A[1-5]|^N'))

## Compare the 1 to 3 factor solutions
model1 = fa(bfi_trim, 1)
model2 = fa(bfi_trim, 2)
model3 = fa(bfi_trim, 3)

## as a matrix
```

```r
obs = cor(bfi_trim, use = "pairwise.complete.obs")
obs
```

```
##             A1          A2          A3           A4          A5          N1
## A1  1.00000000 -0.34019325 -0.26524705 -0.146424511 -0.18143827  0.16647717
## A2 -0.34019325  1.00000000  0.48509804  0.335087208  0.39008358 -0.08767439
## A3 -0.26524705  0.48509804  1.00000000  0.360428299  0.50414114 -0.08410612
## A4 -0.14642451  0.33508721  0.36042830  1.000000000  0.30753726 -0.09893616
## A5 -0.18143827  0.39008358  0.50414114  0.307537265  1.00000000 -0.19573473
## N1  0.16647717 -0.08767439 -0.08410612 -0.098936157 -0.19573473  1.00000000
## N2  0.13931876 -0.05049870 -0.08830390 -0.143785958 -0.18807886  0.70698096
## N3  0.10269550 -0.03549215 -0.04147399 -0.070287040 -0.13549734  0.55642509
## N4  0.05220365 -0.08971128 -0.12824047 -0.166311475 -0.20227344  0.39780585
## N5  0.01667965  0.01916850 -0.03693212 -0.007078645 -0.07612539  0.37812644
##            N2          N3          N4           N5
## A1  0.1393188  0.10269550  0.05220365  0.016679649
## A2 -0.0504987 -0.03549215 -0.08971128  0.019168496
## A3 -0.0883039 -0.04147399 -0.12824047 -0.036932115
## A4 -0.1437860 -0.07028704 -0.16631148 -0.007078645
## A5 -0.1880789 -0.13549734 -0.20227344 -0.076125391
## N1  0.7069810  0.55642509  0.39780585  0.378126441
## N2  1.0000000  0.54910308  0.39157038  0.350612639
## N3  0.5491031  1.00000000  0.51950432  0.428418658
## N4  0.3915704  0.51950432  1.00000000  0.397696146
## N5  0.3506126  0.42841866  0.39769615  1.000000000
```

```r
## in long form
obs_long <- obs %>%
  as_tibble() %>%
  mutate(i = names(.)) %>%
  pivot_longer(cols = -i, names_to = "j", values_to = "obs")
obs_long
```

```
## # A tibble: 100 x 3
##    i     j         obs
##    <chr> <chr>   <dbl>
##  1 A1    A1     1
##  2 A1    A2    -0.340
##  3 A1    A3    -0.265
##  4 A1    A4    -0.146
##  5 A1    A5    -0.181
##  6 A1    N1     0.166
##  7 A1    N2     0.139
##  8 A1    N3     0.103
##  9 A1    N4     0.0522
## 10 A1    N5     0.0167
## # ... with 90 more rows
```

```
## add model residuals
fa_models <- obs_long %>%
  mutate(resid1 = c(model1$residual),
         resid2 = c(model2$residual),
         resid3 = c(model3$residual))
fa_models
```

```
## # A tibble: 100 x 6
##    i     j         obs  resid1  resid2     resid3
##    <chr> <chr>   <dbl>   <dbl>   <dbl>      <dbl>
##  1 A1    A1     1       0.942   0.856   0.826
##  2 A1    A2    -0.340  -0.280  -0.0899 -0.0782
##  3 A1    A3    -0.265  -0.196   0.0122  0.00741
##  4 A1    A4    -0.146  -0.0820  0.0377  0.0328
##  5 A1    A5    -0.181  -0.0919  0.0573  0.0522
##  6 A1    N1     0.166  -0.0138  0.0469 -0.00378
##  7 A1    N2     0.139  -0.0369  0.0235 -0.0136
##  8 A1    N3     0.103  -0.0638  0.0130  0.0335
##  9 A1    N4     0.0522 -0.0919 -0.0588  0.0000186
## 10 A1    N5     0.0167 -0.0977 -0.0309  0.000117
## # ... with 90 more rows
```

```
## pivot to long form
fa_models_long <- fa_models %>%
  pivot_longer(cols = resid1:resid3, names_to = "model", values_to = "resid")
fa_models_long
```

```
## # A tibble: 300 x 5
##    i     j        obs model     resid
##    <chr> <chr>  <dbl> <chr>     <dbl>
##  1 A1    A1     1     resid1  0.942
##  2 A1    A1     1     resid2  0.856
##  3 A1    A1     1     resid3  0.826
##  4 A1    A2    -0.340 resid1 -0.280
##  5 A1    A2    -0.340 resid2 -0.0899
##  6 A1    A2    -0.340 resid3 -0.0782
##  7 A1    A3    -0.265 resid1 -0.196
##  8 A1    A3    -0.265 resid2  0.0122
##  9 A1    A3    -0.265 resid3  0.00741
## 10 A1    A4    -0.146 resid1 -0.0820
## # ... with 290 more rows
```

Packages

```
subset(data.frame(sessioninfo::package_info()), attached==TRUE, c(package, loadedversi
```

```
##               package loadedversion
## dplyr           dplyr        1.0.10
## forcats       forcats         0.5.1
## ggplot2       ggplot2         3.4.0
## MASS             MASS      7.3-58.3
## psych           psych         2.2.9
## purrr           purrr         1.0.1
## readr           readr         2.1.2
## stringr       stringr         1.4.1
## tibble         tibble         3.1.8
## tidyr           tidyr         1.2.1
## tidyverse tidyverse         1.3.1
```