Various exercises on programming in R

Bram Kuijper

2022-01-28

Contents

Chapter 1

Introduction

Here a small selection of problems to get some practical exposure to R programming. Some of these exercises may be quite relevant to one's work, other may be less so. However, relevance to one's work is not a really good criterion here; we simply want to practice problem solving.

1.1 Prior knowledge about R

This set of exercises assumes you already have a reasonable understanding of the R basics. Here a bunch of different sources that provide an overview of the R basics:

- Chapters 1-8 of the book *The Book of R* (Davies, 2016), which is available in UExeter's library here
- An introduction to R freely available from the R website.
- Chapter 2 of the book *The R Book* (Crawley, 2013), which is available in UExeter's library here
- Chapter 1-7 of the book *The Art of R Programming* (Matloff, 2011), which is available in UExeter's library here

One should not search for pdfs of these books online.

1.1.1 Finding help

Next to googling everything you don't understand, you can also find help using the `?` command in R's built-in help pages. For example, you can use the `?` preceding a certain function or variable. For example, in case we want more information about R's `str()` function, we simply type

```
?str
```

Obviously, next to R's built-in help pages, search engines are your friend when you hunt for examples.

Pro-tip: use quotation marks "" when consulting help for operators and weird characters For operators and things with weird characters, using the `?` help function may not bring you very far. For example, let's say you are interested in finding out about R's `%in%` matching operator:

```
?%in%
## Error: <text>:1:2: unexpected SPECIAL
## 1: ?%in%
##      ^
```

Clearly, R chokes on the `%` character. What now? Easy, just use quotes:

```
?"%in%"
```

1.1.2 Different types of variables

You know about variables and different types of data such as `character`, `logical`. Also,you know about different data structures such as `c()` (vectors), `data.frame`s and `list`s. Finally, you know that the `typeof()` function can be used to find out what data is contained in a data type. Similarly, also the `str()` helps you to find out more about the structure of a data type. Please see here for a fantastic overview of the different data types and key data structures.

1.1.3 Operators

Please click the corresponding links if you do not know about arithmetic operators such as `*` (multiplication), `^` (power), `%%` (modulo). The same goes for logical operators such as `||` (which is the `OR` operator) and relational operators such as `>=` (which is the greater-than-or-equal-operator). See Chapter 4 for a brief overview.

1.1.4 Control flow

Things like `if-else` statements and `ifelse()` functions should be understood. Read the freely available Chapter 5 from (Wickham, 2019) for more information, or chapter 7 of (Matloff, 2011)

1.1.5 Sizes of things

To use loops, knowing the sizes of things in R is important. Hence, you know how to obtain the `length()` of a vector, or the number of characters in a string of text using `nchar()`. You also know how to obtain the number of rows and columns of a `data.frame`, using `nrow()`, `ncol()` or its dimensions using `dim()`.

1.2 Selecting subsets of your data

You know the difference between `iris[,1]`, `iris[1,]` and `iris[1,1]`? See Chapter 4 for a great overview of how to subset your data in different ways.

1.2.1 Loop structures

You will need to have read about `for`-loops and potentially also about `switch()` statements. Check out chapter 7 of (Matloff, 2011), chapter 21 of (Wickham and Grolemund, 2017) (available online) or check online for videos on these topics. For example, using the search terms `for loops in R` provides a bunch of great videos.

1.3 Some additional coding tips:

1.3.1 Don't ignore the links and hints

Sometimes there are hints provided in the exercises, for example: "You can do this by using the `paste()` function", which involves you reading the documentation on this function and trying some of the provided examples in the documentation. I typically provide hyperlinks to the documentation, which is why the `paste()` function here is given in blue. Click on them and read them. Please do.

1.3.2 Try out examples for yourself

When reading a book, try the code examples and snippets for yourself, by typing them out (don't copy-paste them, unless it is a massive amount of code). Only by typing out do you see what's going on and will you start to memorize the different coding constructs. Moreover, you are bound to make occasional mistakes! And mistakes are good, as the earlier you learn how to hunt for coding errors and solving mistakes the better.

1.3.3 Search enginges are your friend

Search engines are your friend. If some function or construct is used that is new to one, google what it is about. Say, one encounters the statement

```
means <- apply(X=cars,MARGIN=2,FUN=mean)
```

and one does not know what the `apply` statement is about or what the `cars` variable is about, I'd immediately search for `apply r` or `cars r`.

1.3.4 Comment your code

Always try to briefly comment each statement (or group of statements), using `#`.

```
# select even numbers
even_odd <- c(1,2,3,4,5,6,7,8) %% 2 == 0
```

1.3.5 Keep an error record

Try to keep a record of all the main error messages you encounter, so that you build a record of different error messages and how they have eventually been resolved. This serves as a great lookup tool for recurring errors.

An entry in my error record reads, for example:

```
apply(X=cars,MARGIN=2,FUN=mean())
## Error in mean.default(): argument "x" is missing, with no default
```

This error occurs because the supplied `mean()` function used to calculate means of the different columns is provided with parentheses `()`. However, in order for this function to actually calculate means of each column, you simply need to provide the function name `mean` without the parentheses, or:

```
apply(X=cars,MARGIN=2,FUN=mean)
## speed  dist
## 15.40 42.98
```

1.3.6 Find out the root of errors using `traceback()`

If you stumble upon an error, use R's `traceback()` function to find the root cause. As `traceback()` can produce a lot of garbage, it may help to reduce the amount of output, for example by stating `traceback(max.lines=5)`.

1.3.7 Inspect values using `print()` or use R's debugger

When you have just started to code in R, use as many `print()` statements as possible, to obtain information about the state of your program and the values of your variables. When you become more experienced, you can start to inform yourself about R's very handy debugging tools that can do all the variable inspection work for you. Read more about R debugging in the freely available chapter 22 of (Wickham, 2019).

Chapter 2

Using loops to calculate sums and sums of squares

We often use loops to go over series of values and perform operations on them. While more advanced R users typically use techniques such as vectorization, or functions such as `lapply()`, using basic `for` loops is often the best thing to use when going over a series of values. Here we practice this by performing operations over a simple list of values.

2.1 Exercise: printing elements from a vector

Here we have the following vector of values:

```
# a list of values
my.list <- c(5,10,19,22,3,40,48)
```

Produce a `for` loop that uses the `print()` function to print each value to the console, resulting in the following output:

```
## [1] 5
## [1] 10
## [1] 19
## [1] 22
## [1] 3
## [1] 40
## [1] 48
```

2.2 Exercise: printing every nth row from a `data.frame`

Let us use the built-in `cars` dataset. Use a `for` loop to go over the rows of this `data.frame` and then use the `print()` function to print every 10th row, as in

```
##   speed dist
## 1     4    2
##    speed dist
## 11    11   28
##    speed dist
## 21    14   36
##    speed dist
## 31    17   50
##    speed dist
## 41    20   52
```

2.3 Exercise: summing a vector

Produce a single `for` loop to calculate the sum of `my.list` in exercise 2.1. This is chiefly for the sake of practice, as in reality we would use the `sum()` function for this. The only output of your script should be:

```
## [1] "The sum of the list is 147"
```

which can be done using the functions `print()` and, in particular, `paste()` which allows you to concatenate strings (pieces of text) and numbers into one sentence.

2.4 Sums of squares

Now copy-paste and modify the code from exercise 2.3 to calculate both the sum and the sum of squares within one and the same `for` loop. Calculating the sum of squares is the same as first squaring each numbers and then summing these squares, i.e., $5 \times 5 + 10 \times 10 + \cdots + 48 \times 48$.

The program should have no output, other than the following message at the end:

```
## [1] "The sum is 147 and the sum of squares is 4883"
```

and again, this can be realized using the functions `print()` and `paste()`.

2.5 Sum only if larger than x

Copy-paste the code from exercise 2.4 and use an `if`-statement to calculate the sum and the sum of squares for those numbers in the list above that are greater

or equal than $x = 10$. Other numbers can be ignored. The resulting message should now be:

```
## [1] "The sum is 139 and the sum of squares is 4849"
```

2.6 Separate sums for odd and even numbers

Copy-paste the code from exercise 2.4 and modify it to calculate sums and sums of squares for odd and even numbers separately. Do this using an `if-else` statement. The resulting code should print the following two messages:

```
## [1] "Sum of even numbers is 120. Sum of even squares is 4488."
## [1] "Sum of odd numbers is 27. Sum of odd squares is 395."
```

Note the punctuation in the output above, which was not there before and requires some changes in the `paste()` statement.

Hint: to find out whether a number is odd or even, use the modulo operator `a %% b`. This gives the remainder of the division `a/b`. For example,

```
33 %% 3
## [1] 0
33 %% 2
## [1] 1
33 %% 5
## [1] 3
33 %% 10
## [1] 3
```

2.7 Produce a vector of cumulative values

Say, you have a vector of values, called `a.vec`. From this vector, one can produce another vector, called a cumulative vector, which is often used to develop sampling distributions. The ith index of such a vector contains the sum of elements from element 1 to i. For example, for the following vector

```
a.vec <- c(5,9,1,3,12,8,50,82,1,9,2,7)
```

the corresponding cumulative vector should contain the following sequence of values:

```
##  [1]   5  14  15  18  30  38  88 170 171 180 182 189
```

1. Develop code that produces a cumulative vector for the vector `a.vec` above, where the cumulative vector should be contained in the variable `cumul.vec`. The variable `cumul.vec` should be initialized before the start of the `for`-loop as an `NA`-filled vector of the same length as `a.vec`.

2. Now, we assign `a.vec <- iris[,1]`. Without any further changes to your code, it should now fill `cumul.vec` with cumulative values for this new version of `a.vec`. Output should now be:

```
##   [1]   5.1  10.0  14.7  19.3  24.3  29.7  34.3  39.3  43.7  48.6  54.0  58.8
##  [13]  63.6  67.9  73.7  79.4  84.8  89.9  95.6 100.7 106.1 111.2 115.8 120.9
##  [25] 125.7 130.7 135.7 140.9 146.1 150.8 155.6 161.0 166.2 171.7 176.6 181.6
##  [37] 187.1 192.0 196.4 201.5 206.5 211.0 215.4 220.4 225.5 230.3 235.4 240.0
##  [49] 245.3 250.3 257.3 263.7 270.6 276.1 282.6 288.3 294.6 299.5 306.1 311.3
##  [61] 316.3 322.2 328.2 334.3 339.9 346.6 352.2 358.0 364.2 369.8 375.7 381.8
##  [73] 388.1 394.2 400.6 407.2 414.0 420.7 426.7 432.4 437.9 443.4 449.2 455.2
##  [85] 460.6 466.6 473.3 479.6 485.2 490.7 496.2 502.3 508.1 513.1 518.7 524.4
##  [97] 530.1 536.3 541.4 547.1 553.4 559.2 566.3 572.6 579.1 586.7 591.6 598.9
## [109] 605.6 612.8 619.3 625.7 632.5 638.2 644.0 650.4 656.9 664.6 672.3 678.3
## [121] 685.2 690.8 698.5 704.8 711.5 718.7 724.9 731.0 737.4 744.6 752.0 759.9
## [133] 766.3 772.6 778.7 786.4 792.7 799.1 805.1 812.0 818.7 825.6 831.4 838.2
## [145] 844.9 851.6 857.9 864.4 870.6 876.5
```

Chapter 3

Slightly more advanced for loops

In this set of exercises we will look at slightly more complicated for loops.

3.1 Exercise: a list of random letters

The `LETTERS`

3.2 Exercise: changes in state over time

In many biological models, we model environments that change between two states, `0` and `1`. For example, these environments may represent a dry vs a hot environment. Let ϕ reflect the probability that an environment changes from one state to another.

3.3 Exercise: select columns of a `data.frame` containing floating-point numbers

Data types like `data.frames` and `tibbles` are collections of variables ordered in rows and columns, like a spreadsheet. Take for example the `Cars93` `data.frame` from R's `MASS` package, which we inspect with the `str()` command:

```
library("MASS")
str(Cars93)
## 'data.frame':    93 obs. of  27 variables:
##  $ Manufacturer      : Factor w/ 32 levels "Acura","Audi",..: 1 1 2 2 3 4 4 4 4 5 ...
##  $ Model             : Factor w/ 93 levels "100","190E","240",..: 49 56 9 1 6 24 54 74 73 35 .
```

```
##  $ Type              : Factor w/ 6 levels "Compact","Large",..: 4 3 1 3 3 3 2 2 3 2
##  $ Min.Price         : num  12.9 29.2 25.9 30.8 23.7 14.2 19.9 22.6 26.3 33 ...
##  $ Price             : num  15.9 33.9 29.1 37.7 30 15.7 20.8 23.7 26.3 34.7 ...
...
```

Sometimes, we would like to select certain columns from such a `data.frame`, dependent on a particular condition. Hence the current exercises:

Select only those columns from `Cars93` which contain floating-point numbers, such as `6.5` or `1e-03`. Use a `for`-loop to loop over the columns of `Cars93` to select those columns

To progress with the exercise, here a couple of hints:

3.3.1 Hint 1: data types

The way to find out whether a column of a `data.frame` contains floating-point numbers is to use the `typeof()` function. Let us compare two different columns from the `Cars93` dataset, for example:

```
typeof(Cars93[,"Min.Price"])
## [1] "double"
typeof(Cars93[,"MPG.city"])
## [1] "integer"
```

where `double` reflects that the `Min.Price` column contains floating-point numbers, whereas `integer` reflects that the `MPG.city` column contains integers.

3.3.2 Hint 2: data.frame dimensions

To loop through the columns of the `data.frame`, you need to find out the dimensions. The total number of columns (or rows) can be obtained by `ncol()` (or `ncol()`).

3.3.3 Hint 3: select columns based on a vector

To select multiple columns in a `data.frame` (if you forgot about column and row selection, see page 66 and beyond in (Matloff, 2011)), you can use a vector with numbers:

```
# make a vector containing the column numbers that you want to select
# in this case: column 1, column 2, column 4
column.select <- c(1,2,4)
Cars93[,column.select]
##     Manufacturer          Model Min.Price
## 1          Acura        Integra      12.9
## 2          Acura         Legend      29.2
## 3           Audi             90      25.9
```

```
## 4           Audi            100      30.8
...
```

3.4 Nested for loops

In this part of the exercises, we will be looking at nested `for` loops

3.5 Parameter combinations

Chapter 4

Looping with datasets

In this set of exercises, we will use existing datasets and use looping techniques to read in the data. For example, let's say you have 10 datasets which you need to read into R into one big dataframe. How to go about this? Along the way you will encounter a bunch of functions from various packages that are helpful when working with data files.

4.1 Preliminaries

When generating large databases, we typically have to join together multiple files. Let us download and save a zip archive containing a bunch of Excel files from here. Unzip the folder in some directory.

There are also some other files in the zip file, they should be maintained there for the sake of the exercise.

4.2 Obtaining a list of files

Use the function `list.files()` to return a list of files starting with `sheet_` and with the extension `.xls`. The list should return no other files, and hence you will have to make use of the `pattern` argument of the `list.files()` function to only list the desired files. The resulting output should look like the following:

```
##  [1] "sheet_1.xls"  "sheet_10.xls" "sheet_2.xls"  "sheet_3.xls"  "sheet_4.xls"
##  [6] "sheet_5.xls"  "sheet_6.xls"  "sheet_7.xls"  "sheet_8.xls"  "sheet_9.xls"
```

Hence, other files that are contained in the zip folder, like `another_excel_file.xls` or `sheet_other.txt`, should not feature in the resulting list.

4.3 Obtaining a list of files with full path names

Again, use the function `list.files()` from above, but make sure that it returns the complete path names, rather than just only the names of the sheets. Read the documentation of `list.files()` to see which argument(s) you will have to tweak in order for it to return full path names. Output should look like the following, except that your file system is different:

```
##  [1] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_1.xls"
##  [2] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_10.xls"
##  [3] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_2.xls"
##  [4] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_3.xls"
##  [5] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_4.xls"
##  [6] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_5.xls"
##  [7] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_6.xls"
##  [8] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_7.xls"
##  [9] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_8.xls"
## [10] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_9.xls"
```

In future uses of `list.files()`, retaining the full path name saves a lot of hassle, as it means that any future script can use the output of `list.files()` without having the locate the actual directory of the excel sheets.

We store the list of files that you obtained using the `list.files()` function above for future use, in a variable `xls_sheet_list`.

4.4 Open a single file

Use the `read_excel()` function from the readxl package to read the first excel file of the list (i.e., `xls_sheet_list[[1]]`) into a variable. Specifically, `read_excel()` reads data from the excel sheet into a so-called `tibble`, which is `tidyverse` 's more modern version of a `data.frame`.

Subsequently, use `print()` to display the file's path name and then, below, print the first five lines of the resulting tibble. One can do so by using the `slice_head()` function from the `dplyr` package. The resulting output should look like the following:

```
## [1] "/Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_1.xls"
```

```
## # A tibble: 5 x 3
##   treatment individual  size
##       <dbl>      <dbl> <dbl>
## 1         1          1  3.27
## 2         2          1  5.87
## 3         3          1  9.25
## 4         1          2  2.93
## 5         2          2  5.25
```

4.5 Getting information about each file within a `for`-loop

Now we will use a `for`-loop to go through the list of files and read each file into a `tibble` using `read_excel()`. Subsequently, we can then request some information about the current excel sheet by interrogating the resulting `tibble`. Here, we merely print the path name of each file (which you already have done previously), the number of rows contained in the `tibble` (using either `nrow()` or `tidyverse`'s `tally()`) and the names of all the columns using the `names()` function. Hence, output should look like the following:

```
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_1.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_10.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_2.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_3.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_4.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_5.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_6.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_7.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_8.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
## [1] "path name: /Users/bram/Projects/exercises_r/r_exercises_bookdown/code//sheet_9.xls"
## [1] "rows: 30"
## [1] "column names: treatment,individual,size"
```

4.6 Accumulating all `tibble`s into a single big `tibble`

Now try to accumulate all `tibble`s into a big one. Before we start our `for` loop, we assign an empty `tibble` (i.e., `tibble()`) to a variable, say `super_tbl`. Starting with an empty `tibble` makes sure that when we run our script again, `super_tbl` does not retain the contents of any previous runs of the same script.

Then, during each iteration of the `for`-loop, we append the contents of each excel sheet to this `super_tbl` variable by using the function `bind_rows()`. We then assign the result of this function back to `super_tbl`, so that `super_tbl` grows during each iteration with the contents of each excel sheet.

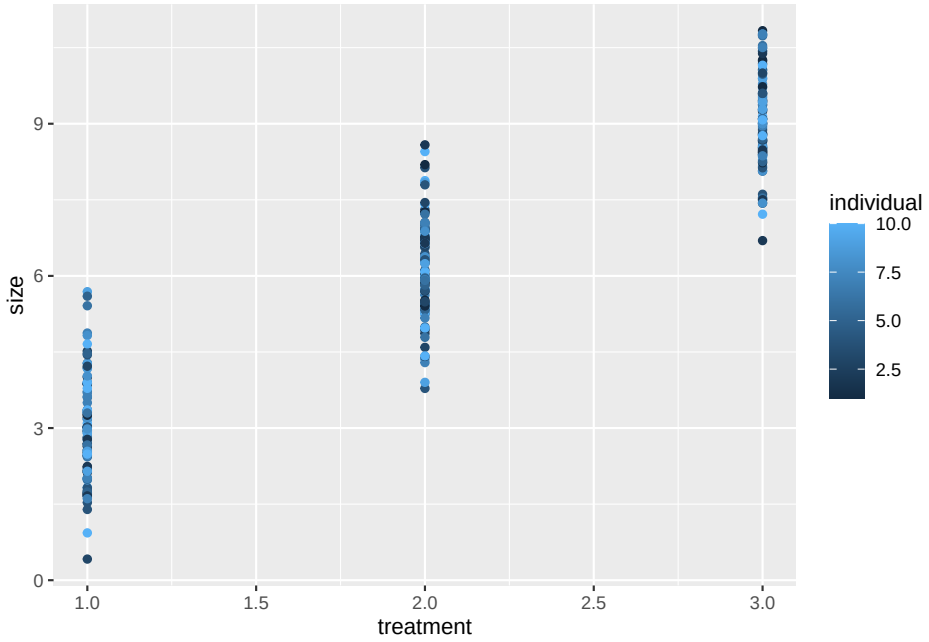
Finally, we then just use `print(super_tbl)` and the `summary()` function to inspect the result

```
## # A tibble: 300 x 3
##    treatment individual  size
##        <dbl>      <dbl> <dbl>
##  1         1          1  3.27
##  2         2          1  5.87
##  3         3          1  9.25
##  4         1          2  2.93
##  5         2          2  5.25
##  6         3          2  9.58
##  7         1          3  4.18
##  8         2          3  6.27
##  9         3          3  9.37
## 10         1          4  1.40
## # ... with 290 more rows
```

```
##    treatment   individual        size
##  Min.   :1   Min.   : 1.0   Min.   : 0.4171
##  [ reached getOption("max.print") -- omitted 5 rows ]
```

4.7 Plotting the data

The next task is to use `ggplot2` to plot the contents of `super_tbl`. Specifically, we plot the size on the y-axis and the different treatments on the x-axis, while the colors of the points should reflect the different individuals. We use points, rather than lines, by using `geom_point()`. On `ggplot2`'s website you can also download an elegant cheat sheet that is very helpful in achieving this.



4.8 Data modification / conversion

As you can see when you `print()` the contents of `super_tbl`, all data types have the type of a `<dbl>` (double), which stands for double-precision floating point value. Such floating point values are typically used for data which varies in a continuous fashion, as is the case for the `size` column in `super_tbl`.

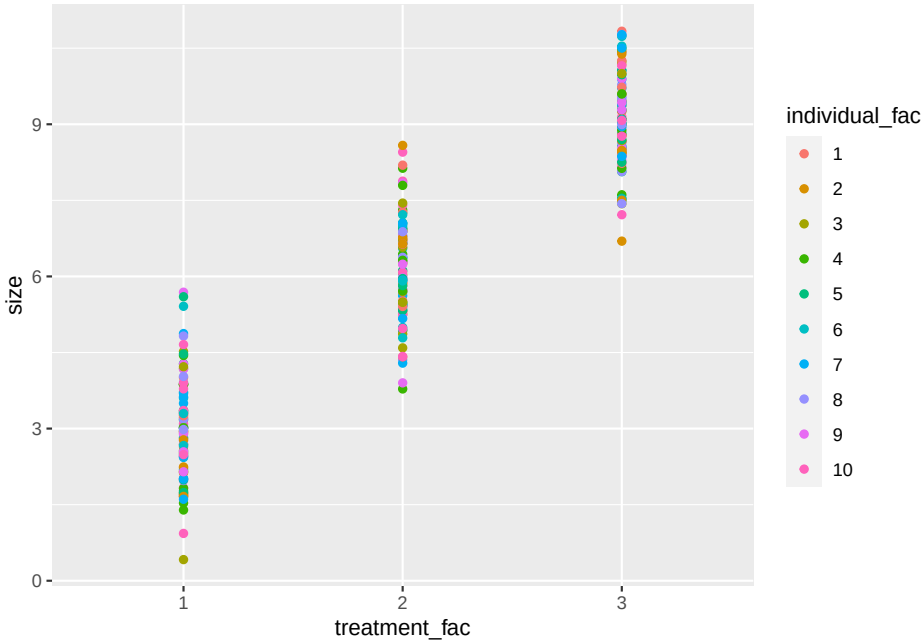
However, we often code treatments, and sometimes also individuals (if there are only a few, like here), as discrete factors at a nominal scale, meaning that number '1' is not necessarily lower than '2', it is merely an identifier of that treatment or individual. To accomplish this, we convert the `individual` and `treatment` column from `<dbl>` to `<fct>`, which stands for factor.

To convert values into factors, we might either use the old-style function `as.factor()` or we might use the more modern `as_factor()` function from the haven package. For this example there is no difference between both functions, but in general `as_factor()` is slightly more generic.

How to change our data using the `as_factor()` function? What we can do is apply the function to the columns in question, by using the `mutate()` function from the `dplyr()` package. Using this `mutate()` function (you have been checking the examples on the manual page, right?), we can then add two new columns to our data frame, say `individual_fac` and `treatment_fac` which reflect our new factor data.

If we now plot our new `individual_fac` and `treatment_fac` columns, our plot

will look slightly different:



Can you describe what is different between this plot and the previous one and why?

4.9 A further aside: regular expressions

The next exercise focuses on changing the data during the `for` loop that we used in section @ref{forLoop}.

Consider the following scenario: your supervisor tells you (belatedly, as always) that they have omitted some relevant detail in the previous analysis. It turns out that the size measurements in the first 4 excel sheets (i.e., `sheet_1.xls`, `sheet_2.xls`, `sheet_3.xls` and `sheet_4.xls`) were performed on a different breed of individuals (breed A), while measurements from other excel sheets used animals of breed B.

Let us visually inspect whether the relationship between treatment and size is different for the two different breeds of animals. To this end, we do not modify the underlying excel sheets, rather we add the column `breed` to the `tibbles` when reading in the data from each excel sheet. The `breed` column should have the values `A` for the first four sheets and the value `B` for the remaining sheets respectively.

A first suggestion may be to simply keep track of at which of the excel sheets we are when looping over the `xls_sheet_list` variable which contains all the files

(see section @ref{forLoop}), and then set the `breed` variable according to an `if else` statement (i.e., first 4 files `A`, remaining files `B`). However, if you look at the listing of files in section @ref{fileList} above, you will see that `sheet_10.xls` (breed `B`) is at the second position in the variable `xls_sheet_list`, so using a simple rule like that is not going to work.

4.9.1 Regular expressions to the rescue

Hence, we need to inspect the filenames themselves and sort out `sheet_1.xls`, `sheet_2.xls`, `sheet_3.xls` and `sheet_4.xls` from the rest. How to do this? Regular expressions are a great way to filter numbers from textual data.

While we are not delving into regular expressions now,

Bibliography

Crawley, M. J. (2013). *The R Book*. John Wiley & Sons, Chichester, UK.

Davies, T. M. (2016). *The Book of R*. No Starch Press, San Francisco, CA.

Matloff, N. (2011). *The art of R programming*. no starch press, San Francisco.

Wickham, H. (2019). *Advanced R*. CRC Press, Boca Raton, FL.

Wickham, H. and Grolemund, G. (2017). *R for Data Science*. O'Reilly, Sebastopol, CA.