Data manipulation

Contents

Introduction

This is the first programming practical. If you haven't yet done so, open the project file `02_Data_manipulation.Rproj` in RStudio. You can choose to write the answers to your exercises in either an `.R` file or in an `.Rmd` file. Example answer files are provided in the project directory (`example_answers.Rmd` and `example_answers.R`). You can open these from the files pane and use them as a starting point. While working through the exercises, write down your code in one of these files. Use proper style and provide comments so you can read it back later and still understand what is happening.

The practicals always start with the packages we are going to use. Be sure to run these lines in your session to load their functions before you continue. If there are packages that you have not yet installed, first install them with `install.packages()`.

```
library(ISLR)
library(tidyverse)
library(haven)
library(readxl)
```

Data types

There are several data types in `R`. Here is a table with the most common ones:

Type	Short	Example
Integer	int	0, 1, 2, 3, -4, -5

Type	Short	Example
Numeric / Double	dbl	0.1, -2.5, 123.456
Character	chr	"dav is a cool course"
Logical	lgl	TRUE / FALSE
Factor	fctr	low, medium, high

The `class()` function can give you an idea about what type of data each variable contains.

---

1. **Run the following code in R and inspect their data types using the `class()` function. Try to guess beforehand what their types will be!**

```
object_1 <- 1:5
object_2 <- 1L:5L
object_3 <- "-123.456"
object_4 <- as.numeric(object_2)
object_5 <- letters[object_1]
object_6 <- as.factor(rep(object_5, 2))
object_7 <- c(1, 2, 3, "4", "5", "6")
```

the factor data type is special to R and uncommon in other programming languages. It is used to represent categorical variables with fixed possible values. For example, when there is a multiple choice question with 5 possible choices (a to e) and 10 students answer the question, we may get a result as in `object_6`.

Vectors can have only a single data type. Note that the first three elements in `object_7` have been converted. We can convert to different data types using the `as.<class>()` functions.

---

2. **Convert `object_7` back to a vector of numbers using the `as.numeric()` function**

Lists

A list is a collection of objects. The elements may have names, but it is not necessary. Each element of a list can have a different data type, unlike vectors.

---

3. **Make a list called `objects` containing object 1 to 7 using the `list()` function.**

You can select elements of a list using its name (`objects$elementname`) or using its index (`objects[[1]]` for the first element).

A special type of list is the `data.frame`. It is the same as a list, but each element is forced to have the same length and a name. The elements of a `data.frame` are the columns of a dataset. In the tidyverse, `data.frames` are called `tibbles` and they are printed in a nice way, as we will see later.

---

4. **Make a data frame out of `object_1`, `object_2`, and `object_5` using the `data.frame()` function**

Just like a list, te columns in a data frame (the variables in a dataset) can be accessed using their name `df$columnname` or their index `df[[1]]`. Additionally, the tenth row can be selected using `df[10, ]`, the second column using `df[, 2]` and cell number 10, 2 can be accessed using `df[10, 2]`. This is because data frames also behave like the `matrix` data type in addition to the `list` type.

Loading, viewing, and summarising data

We are going to use a dataset from Kaggle - the Google play store apps data by user `lava18`. We have downloaded it into the data folder already from https://www.kaggle.com/lava18/google-play-store-apps (downloaded on 2018-09-28).

Tidyverse contains many data loading functions – each for their own file type – in the packages `readr` (default file types), `readxl` (excel files), and `haven` (external file types such as from SPSS or Stata). The most common file type is `csv`, which is what we use here.

---

5. **Use the function `read_csv()` to import the file "data/googleplaystore.csv" and store it in a variable called `apps`.**

If necessary, use the help files. These import functions from the tidyverse are fast and safe: they display informative errors if anything goes wrong. `read_csv()` also displays a message with information on how each column is imported: which variable type each column gets.

---

6. **Did any column get a variable type you did not expect?**

---

7. **Use the function `head()` to look at the first few rows of the `apps` dataset**

---

8. **Repeat steps 5, 6, and 7 but now for "data/students.xlsx" (NB: You'll need a function from the package `readxl`). Also try out the function `tail()` and `View()` (with a capital V).**

9. **Create a summary of the three columns in the students dataset using the `summary()` function. What is the range of the grades achieved by the students?**

Data transformation with `dplyr` verbs

The tidyverse package `dplyr` contains functions to transform, rearrange, and filter data frames.

Filter

The first verb is `filter()`, which selects rows from a data frame. Chapter 5 of R4DS states that to use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: >, >=, <, <=, != (not equal), and == (equal).

When you're starting out with R, the easiest mistake to make is to use = instead of == when testing for equality.

10. **Look at the help pages for `filter()` (especially the examples) and show the students with a grade lower than 5.5**

11. **Show only the students with a grade higher than 8 from programme A**

If you are unsure how to proceed, read Section 5.2.2 from R4DS.

Arrange

The second verb is `arrange()`, which sorts a data frame by one or more columns.

12. **Sort the students dataset such that the students from programme A are on top of the data frame and within the programmes the highest grades come first.**

Select

The third verb is `select()`, which selects columns of interest.

---

13. **Show only the `student_number` and `programme` columns from the students dataset**

Mutate

With `mutate()` you can compute new columns and transform existing columns as functions of the columns in your dataset. For example, we may create a new logical column in the students dataset to indicate whether a student has passed or failed:

```
students <- mutate(students, pass = grade > 5.5)
students
```

```
## # A tibble: 37 x 4
##    student_number grade programme pass
##             <dbl> <dbl> <chr>     <lgl>
##  1        5117250  6.54 A         TRUE
##  2        6562582  7.57 A         TRUE
##  3        6000241  6.08 B         TRUE
##  4        4862862  7.71 A         TRUE
##  5        6561723  6.57 B         TRUE
##  6        5625916  7.90 B         TRUE
##  7        4096023  5.92 A         TRUE
##  8        6114656  5.16 A         FALSE
##  9        5265402  5.49 B         FALSE
## 10        5977188  7.26 B         TRUE
## # ... with 27 more rows
```

Now, the students dataset has an extra column named “pass”.

You can also transform existing columns with the `mutate()` function. For example, we may want to transform the programme column to an actual programme name according to this table:

Code	Name
A	Science
B	Social Science

---

14. **Use `mutate()` and `recode()` to change the codes in the programme column of the students dataset to their names. Store the result in a variable called `students_recoded`**

Chapter 5 of R4DS neatly summarises the five key dplyr functions that allow you to solve the vast majority of your data manipulation challenges:

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).

Cleaning data files and extracting the most useful information is essential to any downstream steps such as plotting or analysis. Make sure you know exactly which variable types are in your tibbles / data frames!

Data processing pipelines

A very useful feature in tidyverse is the pipe `%>%`. The pipe inputs the result from the left-hand side as the first argument of the right-hand side function: `filter(students, grade > 5.5)` becomes `students %>% filter(grade > 5.5)`. With the pipe, a set of processing steps becomes a neatly legible data processing pipeline!

Different tasks we have performed on the students dataset can be done in one pipeline like so:

```
students_dataset <-
  read_xlsx("data/students.xlsx") %>%
  mutate(prog = recode(programme, "A" = "Science", "B" = "Social Science")) %>%
  filter(grade > 5.5) %>%
  arrange(programme, -grade) %>%
  select(student_number, prog, grade)

students_dataset
```

```
## # A tibble: 34 x 3
##    student_number prog    grade
##             <dbl> <chr>   <dbl>
##  1        4011659 Science  8.94
##  2        4133949 Science  8.40
##  3        6553913 Science  8.24
##  4        6352581 Science  8.09
##  5        6165611 Science  8.02
##  6        6997130 Science  7.75
##  7        4862862 Science  7.71
##  8        6562582 Science  7.57
```

```
## 9         4483974 Science  7.46
## 10        5128923 Science  7.26
## # ... with 24 more rows
```

In one statement, we have loaded the dataset from disk, recoded the programme variable, filtered only students that pass, reordered the rows and selected the relevant columns only. We did not need to save intermediate results or nest functions deeply.

Create a data processing pipeline that loads the apps dataset, parses the number of downloads using `mutate` and `parse_number()`, shows only apps with more than 500 000 000 downloads, orders them by rating (best on top), and shows only the relevant columns (you can choose which are relevant).

If you find duplicates, you may need to use `distinct(App, .keep_all = TRUE)` as the last step in your pipeline to remove duplicate app names.

Grouping and summarisation

We have now seen how we can transform and clean our datasets. The next step is to start exploring the dataset by computing relevant summary statistics, such as means, ranges, variances, differences, etc. We have already used the function `summary()` which comes with `R`, but `dplyr` has extra summary functionality in the form of the `summarise()` (or `summarize()`) verb.