Minilab 3a Worksheet

Pipes, Groups and Summaries

In the previous minilab we started looking at *data* wrangling (sometimes called *data munging*).

Image from https://timewellspent.kronos.com/2015/01/19/wrangling-big-data/

In particular, we looked at slicing and dicing a tidy (rectangular or structured) dataset, i.e., how to filter/arrange rows and select/mutate columns from a tibble using the dplyr package in the R tidyverse.

Did you notice how the dplyr functions behave in a similar way to SQL? SQL was designed for managing data in a relational database. But the R tidyverse is design for data analysis.

dplyr Function	Description	Equivalent SQL
select()	Selecting columns (variables)	SELECT
filter()	Filter (subset) rows	WHERE
arrange()	Sort the data	ORDER BY
mutate()	Creating new variables	COLUMN ALIAS

Throughout this minilab, you will find it useful to refer to the "Data Transformations with dplyr" cheat sheet (see https://rstudio.com/resources/cheatsheets/) and mentally tick off the functionality as we go.

1. Pipes

Remember the "200 countries, 200 years, in four minutes" video featuring Prof Hans Rosling. Reminder yourself by watching https://www.youtube.com/watch?v=ahp7QhbB8G4

(1) The data presented in the video is available in the R package "gapminder".

```
library(tidyverse)
install.packages("gapminder")    # run only once
library(gapminder)
names(gapminder)
ggplot(gapminder, aes(x=gdpPercap, y=lifeExp)) +
  geom_point(aes(colour=continent)) +
  scale_x_log10()
```

Notice that the scale on the x-axis reports numbers like "1e+03". This is scientific notation which means $1 \times 10^3 = 1000$. To force R to display large numbers in a more familiar form, just add the following command before you start to plot.

```
options(scipen=1000)
```

(2) In the last minilab, we saw that (in the R tidyverse) the various dplyr functions all take a tibble as the first input argument and output a tibble. *What if we want to apply a sequence of these functions?*

The first approach is to use intermediate variables to store the output of each function call, which can then be used as the input to the next function call. However, this approach creates a whole lot of variables in the workspace which we might not necessarily wish to keep.

```
# Approach 1: Intermediate variables
A = filter(gapminder, continent=="Europe", year==2007)
B = select(A,-continent,-year)
ggplot(B,aes(x=gdpPercap,y=lifeExp, size=pop)) +
  geom_point(alpha=0.5)
```

(3) Another approach is to use *nested function calls*, i.e., the first input argument is actually a function call. It is hard to see exactly what is going on at a glance, and it would make debugging difficult.

```
# Approach 2: Nested function calls
ggplot(select(filter(gapminder, continent=="Europe", year==2007),
              -continent,-year),aes(x=gdpPercap,y=lifeExp, size=pop)) +
  geom_point(alpha=0.5)
```

(4) A much better approach is to use the **pipe** operator (written as %>%), which takes the result from one function and passes it in as *the first argument* to the next function. Because all dplyr functions take as a first argument the tibble to manipulate, and then return a manipulated tibble, it is possible to "chain" together any of these functions using a pipe.

```
# Approach 3: Pipes
gapminder %>%
  filter(continent=="Europe", year==2007) %>%
  select(-continent,-year) %>%
  ggplot(aes(x=gdpPercap,y=lifeExp, size=pop)) +
    geom_point(alpha=0.5)
```

Notice how this approach makes it clear to see the *data flow* in our R script. So from now on, we will use pipes wherever possible. It also does not take up lots of memory storing all the intermediate data tables.

Useful tip. In RStudio ctrl-shift-m is the keyboard shortcut for %>% and alt-shift-k is the keyboard shortcut for telling you what the keyboard shortcuts are.

2. Groups and Summaries

We often wish to produce various counts and summaries of groups within a dataset.

(1) Firstly, we can easily produce summaries of variables in a tibble. A *summary function* takes a vector as an input and returns one value. The summary function n() counts the number of rows, n_distinct(variable) counts the number of distinct values in the variable, and mean(variable) calculates the mean of the values in the variable.

```
gapminder %>%
  filter(year==1952) %>%
  summarise(num_countries=n_distinct(country),
            mean_pop=mean(pop))
```

This dplyr pipe counts the number of distinct countries and calculates the mean population for the year 1952. *Notice that the output is a tibble with only one row.*

The back of the "Data Transformations with dplyr" Cheat Sheet lists a number of possible summary functions (see https://rstudio.com/resources/cheatsheets/). These include counts (n, n_distinct, sum), measures of locations (mean, median), current order (first, last, nth), rank (quantile, min, max) and spread (sd, var, IQR, mad). Most of these concepts are covered in the short lectures this week. It is also possible to write your own summary function.

(2) Secondly, we can group the rows of a tibble by some logical criterion (using group_by()) and then apply the summary functions to each group of rows separately.

```
gapminder %>%
  group_by(year) %>%
  summarise(num_countries=n_distinct(country),
            mean_pop=mean(pop))
```

This dplyr pipe counts the number of distinct countries and calculates the mean population for each year in the gapminder dataset. *Notice that the output is a tibble with one row for each group, i.e., each year in this case. So the result of using group_by() and summarise() is a table of summary values.*

(3) We can then pipe this summary dataset (tibble) directly into ggplot to draw a basic line graph of the results.

```
gapminder %>%
  group_by(year) %>%
  summarise(num_countries=n_distinct(country),
            mean_pop=mean(pop)) %>%
  ggplot(aes(x=year,y=mean_pop)) +
    geom_line()
```

If you are seeing the labels on the y-axis in "scientific notation", e.g., "3.5e+07" which means $3.5 \times 10^7 = 35000000$, remember you can disable this as follows.

```
options(scipen=1000)
```

Together, group_by() and summarise() provide the mostly commonly used tools when working with dplyr. If you apply summarise() without first defining the groups it will have one group including all rows.

Challenge (optional). Adapt the dplyr pipe above to use gapminder data to plot a line graph of the total population of each continent over the years. *Hint:* group by both continent and year and then use continent to colour each line.

Exercise. Experiment with the dplyr functions introduced above on the *starwars* dataset (built in to the tidyverse).

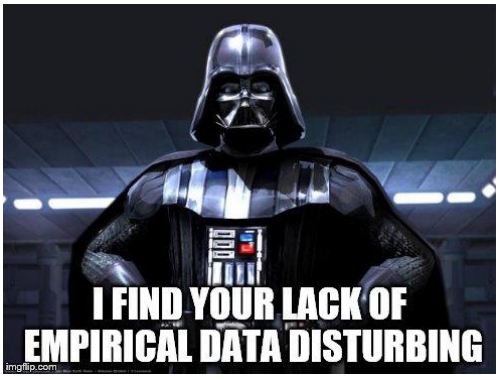

Image from https://i.imgflip.com/keajw.jpg

```
starwars
names(starwars)
```

(a) Convert each of these commands (separately) into a dplyr pipe.

```
filter(starwars, species=="Droid")
arrange(starwars, desc(mass))
```

(b) Write a dplyr pipe to group the starwars dataset by species and count the number of characters of each species and the mean mass of each species.

Exercise. We can import football data directly from http://www.football-data.co.uk/. Firstly, select one of the following leagues: E0 (English Premier League), E1 (Championship), E2 (League 1), E3 (League 2), EC (Conference), or you can try a different league from around Europe such as SC0 (Scottish Premier League), D1 (Bundesliga, Germany), I1 (Serie A, Italy), and SP1 is La Liga (Spain). You can work out some other options from carefully looking at https://www.football-data.co.uk/data.php.

```
# Choose which football league
myleague = "E0"
# Get data from www.football-data.co.uk
url = paste0("http://www.football-data.co.uk/mmz4281/1920/",myleague,".csv")
football = read_csv(url)
football
distinct(football, HomeTeam)
```

If this R code does not work it could be that the website is being blocked by your firewall.

See https://www.football-data.co.uk/notes.txt for a description of the variables (columns).

(a) Pick your favourite team in this league, and filter out only those matches (rows) that your team is involved in (either as the home team or the away team).

(b) Write a dplyr pipe to find the mean (average) number of Full Time Home Goals scored by each team over the league season.

Summary

In this minilab, we have looked at pipes (fantastic for mapping the flow of data through an analysis), and seen how summaries by group are very useful for analysing data. Overall, we have been transforming data that are stored in rectangular tables (tibbles) so that we can answer a particular query. Notice how useful pipes and group summaries are. Many of these queries would be very difficult to conduct in SQL. Hopefully, you are starting to get comfortable with manipulating datasets.

Looking at the data science workflow below, and the corresponding R tidyverse packages, which of these have we seen so far? Which are you most looking forward to exploring further?

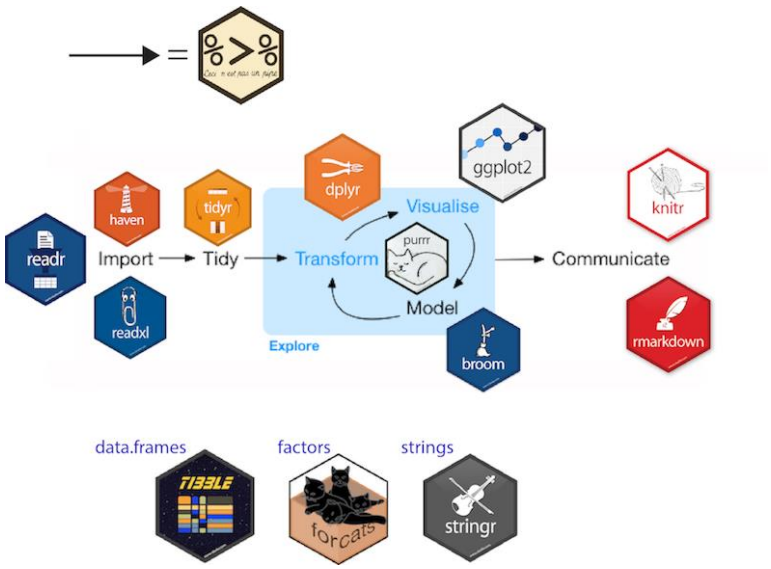

Image from: https://rworkshop.uni.lu/lectures/lecture07_plotting.html#4