PRACTICAL 2

Aim:

Introducing you to data wrangling / manipulation in R, specifically using the `tidyverse` group of packages. We will mostly be using the package `dplyr`, which is part of the `tidyverse`.

Mode of study:

These instructions should enable you to work through the required new material on your own. However, it is **strongly** recommended that you attend your assigned practical lecture slot, where the lecturer will go through these steps and also provide more detail than can be found in these instructions.

When attending a practical lecture, you should follow along on your own laptop or on one of the venue computers; the aim of a practical lecture is not to watch the lecturer complete the steps, but to learn by doing.

Before you start:

Remember from Practical 1 that a *package* in R is a collection of functions, data and documentation. The use of packages makes R very powerful, as it avoids you having to program everything from scratch.

Before using a package for the first time, you must install it.

In every R session where you want to use an installed package, you must load the package.

Installing and loading packages can be done in two ways: using the bottom right-hand window in RStudio, or typing directly in the console.

Instructions:

This document contains discussions of some basic concepts, and instructions which you should execute in RStudio. **Text in bold** indicate actions that are to be taken within RStudio. `Text highlighted in grey` shows the code you should enter in RStudio.

Due date:

For Tutorial 2, it will be assumed that you have mastered the basic concepts discussed here.

THE `tidyverse` PACKAGES

1. The `tidyverse` is a collection of packages sharing a common philosophy of data and programming. These packages are specifically designed to work well together.

> **Tip:**
> You can read more about the tidyverse and the philosophy behind it here:
> https://tidyverse.tidyverse.org/articles/paper.html

We will use these packages to manipulate (wrangle) and visualise data. Most of what we will do using the `tidyverse` can also be done in base R. For instance, in Tutorial 1 you saw how to subset vectors using square brackets [ ]. This can also be done in the `tidyverse`, but in a much more intuitive way.

Most of the functions from the `tidyverse` we will be using in this module, come from the packages `dplyr` and `ggplot2`: `dplyr` for data manipulation and `ggplot2` for data visualisation. These packages don't have to be used together; many people use `ggplot2` for its good data visualisation capabilities, but use base R for data manipulation. However, since all the `tidyverse` packages are designed to work well together, we will use them throughout this module.

1.1 **Load the `tidyverse` packages in RStudio.**

Remember that you can do this from the Packages tab (bottom right hand window of RStudio), or by typing `library(tidyverse)` in the console.
(Of course if you are working on your own computer you have to install the `tidyverse` packages first, but I am assuming you did this in Practical 1.)

1.2 **Go to the Packages tab and check which packages have now been loaded.**

You should see tick marks next to the following packages:
`dplyr`, `forcats`, `ggplot2`, `purr`, `readr`, `stringr`, `tibble`, `tidyr`

DATA FRAMES vs. TIBBLES

2. You were introduced to *data frames* in Tutorial 1. Data frames are rectangular data structures, used to store tabular data, with rows and columns. In other words, data frames are similar to matrices, but they can consist of many different classes (types) of data, making them suitable to use with many different kinds of data sets.

2.1 R comes with many data sets pre-installed; some in base R, and other in packages.
Examine the `mtcars` dataset by typing `?mtcars` in the console.

2.2 You can print the contents of an entire data set in your R console by entering the name of the data set at the command prompt in the console.
Print the `mtcars` data set by typing `mtcars` in the console.

2.3 The `mtcars` datasets is fairly small so printing the entire dataset was not problematic.
Check the number of rows and columns in the `mtcars` dataset.
[Hint: recall from Tutorial 1 that you can do this using the `dim()` function.]

2.4 If you are working with a large data frame in R and you want to see what the data looks like, it is better to use the `head()` function to print only the first few rows of the data set.
Print the first few rows of `mtcars` by typing `head(mtcars)`.

2.5 When we are working with data in R, it is important to also examine the data types of the different variables, and to make sure that these are correct. (We discussed the different data types in Lecture 4, and in Tutorial 1 you encountered the different data types in R.)
The `str()` function is a useful way of getting a quick overview of a data frame.
View the data types of the different variables in `mtcars` by typing `str(mtcars)` in the console.

2.6 **Take note of the use of the terms *observations* and *variables* in the output.**

3. A *tibble* is just a special type of data frame. Printing and subsetting data are much easier when working with tibbles. Tibbles also avoid some of the behaviour in R that can be frustrating when working with data, for instance automatically turning characters into strings, and the automatic addition of row names.

3.1 The `mpg` dataset is part of the `ggplot2` package.
Examine the `mpg` dataset by typing `?mpg` in the console.

3.2 **Print the `mpg` data set by typing `mpg` in the console.**
Note that this gives a much neater presentation than in the case of the `mtcars` dataset, even though the `mpg` data set is much larger. This is because the `mpg` data set is a tibble and not a data frame.
It also automatically shows the size of the data set as well as the data types of each variable.

3.3 **Check the class of the `mtcars` and `mpg` data sets and confirm that the `mpg` data set is a tibble while the `mtcars` data set is an ordinary data frame.**
[Hint: recall from Tutorial 1 that you can use the `class()` function for this.]

3.4 You can use the `View()` function to get a spreadsheet-style view of a data object in R.
Do this for the `mpg` data set by typing `View(mpg).`
Note that this opens in the RStudio viewer, in the top left window.

---

4. We discussed data types in Lecture 4.

4.1 **Print the `mpg` data set again in the console. You only have to type `mpg` in the console.**
At the top of the printout on the screen, take note of the fact that it indicates that `mpg` is a tibble with dimensions 234 `x` 11. Below the different variable names it then shows the data type of each variable, enclosed in < >.
For instance, the `manufacturer` variable is indicated as `<chr>`.

The abbreviations used in the column headers of a tibble to represent the different data types are as follows:

`Int`	Integers
`Dbl`	Doubles (real numbers)
`Chr`	Character vector / string
`Dttm`	Date-times
`Lgl`	Logical
`Fctr`	Factors
`Date`	Dates

DATA MANIPULATION WITH `dplyr`

5. The five `dplyr` functions you will use most often to manipulate date are:

`filter()`	Subset by row (i.e. pick observations based on their values)
`arrange()`	Reorder rows
`select()`	Subset by column (i.e. pick variables based on their names)
`mutate()`	Create new variables with functions of existing variables
`summarize()`	Collapse many values down to a single summary

Remember that when you use a function in R, you also have to specify the arguments. In all of the functions above, the first argument is a data frame, while subsequent arguments describe what to do with the data frame, using variable names. The result of these function calls is a new data frame, and you can also combine multiple steps in a single instruction.

We will explore the use of these functions for data manipulation purposes using the `mpg` data set.

---

6. With the `filter()` function, we can subset observations based on their values. The arguments provided to the function (other than the first argument which specifies the name of the data frame), are the expressions that are to be used to filter the data.
We will consider a few examples:

6.1 **Select all cars manufactured by Toyota:**

```
filter(mpg, manufacturer == "toyota")
```

[Hint: Remember that R is case sensitive!]

6.2 **Select all Toyota Corolla's:**

(in other words, all cars manufactured by Toyota and model equal to Corolla).

```
filter(mpg, manufacturer == "toyota", model == "corolla")
```

6.3 **Select all cars manufactured in 2008:**

```
filter(mpg, year == 2008)
```

Tip:
Take note of the fact that in specifying the condition on which to filter, you need to place the value to the right of the == in double quotation marks in the case of a character variable, but not when the variable is numeric.

6.4 **Select all cars obtaining more than 20 miles per gallon in the city:**

```
filter(mpg, cty > 20)
```

6.5 **Select all SUV's and minivan's:**

```
filter(mpg, class == "suv" | class == "minivan")
```

6.6 **Select all cars that were not manufactured in 2008:**

```
filter(mpg, year != 2008)
```

Note how the logical operators you encountered in Tutorial 1 were used in 6.5 and 6.6.

7. `arrange()` works in a similar fashion to `filter()`, except that it changes the order of the observations instead of selecting them. The first argument is the data frame, followed by the set of variable names (or expressions based thereon) to be used to order the observations (rows). If you provide more than one variable name, it will sort on the first variable first, and then the second will be used to break ties in the values of the first variable, and so on.

The default sort order is ascending; to sort in descending order you have to use the argument `desc()`.

7.1 **Sort by engine displacement, in ascending order:**

```
arrange(mpg, displ)
```

7.2 **Sort by engine displacement and city miles per gallon:**

```
arrange(mpg, displ, cty)
```

7.3 **Sort by engine displacement in descending order:**

```
arrange(mpg, desc(displ))
```

---

8. Data sets often contain large numbers of columns and many might not actually be of interest to the data scientist. You can use `select()` to choose only the columns (variables) that are of interest to you.

8.1 **Select only the model and manufacturer columns:**

```
select(mpg, manufacturer, model)
```

8.2 **Select all columns except model and manufacturer:**

```
select(mpg, -manufacturer, -model)
```

(Note the use of negative indexing, as encountered in Tutorial 1.)

8.3 **Select all columns from manufacturer to year:**

```
select(mpg, manufacturer:model)
```

(Recall from Tutorial 1 that the ':' operator means "to".)

---

9. Using the `mutate()` function we can create new variables. This is often an important part of the data science process! To do this you can use the standard arithmetic operators (such as +, -, *, / and ^), aggregate functions such as `mean()` or `sum()` and logical comparisons.

9.1 **Create a new variable in the `mpg` data set which gives the average miles per gallon (for city and highway consumption). You should call this variable `avgcons`.**

```
mutate(mpg, avgcons = (cty + hwy)/2)
```

Tip:

You might have noticed that the new variables you created were added at the end of your data set. If you don't want to keep the original variables, only the new ones, you can use `transmute()` instead of `mutate()`. You can try this out on your own.

---

10. The `summarize()` function is seldom used on its own. Instead, it is used in conjunction with the `group_by()` function.

10.1 You could use `summarize()` to **calculate the average city miles per gallon** in the following way:

```
summarize(mpg, mean(cty))
```

However, you could just as well have done it in the following way:

```
mean(mpg$cty)
```

10.2 The real power of `summarize()` lies in combining it with the `group_by()` function.

For instance, we could **calculate the average city miles per gallon for cars with different number of cylinders**:

```
by_cyl <- group_by(mpg, cyl)
summarize(by_cyl, mean(cty))
```

Now for instance you can see that cars with more cylinders have a lower average miles per gallon.

PIPE OPERATOR

11. In (10.2) above, we created an intermediate data frame, `by_cyl`. For complicated analyses, it can become cumbersome to include all these intermediate steps. The *pipe operator*, %>%, allows us to combine multiple operations. It also makes code easier to read. When reading code, you can think of the pipe as the word "then".

11.1 To replicate (10.2) using the pipe (i.e. **calculate the average city miles per gallon for cars with different number of cylinders**), you only have to type a single line of code and there are no intermediate objects that are created:

```
mpg %>% group_by(cyl) %>% summarize(mean(cty))
```

Note that you start with the data set you are working with, and then follow it with the operations you want to perform, separating each operation with a pipe.
So the above line of code means:
Start with the `mpg` data, then group by `cyl`, then summarize by calculating the mean of `cty`.
(See how easy to read the code using the pipe operator is!)

11.2 We can also include operations on more than one variable in the `summarize()` function.
Using pipes, calculate the average city and highway fuel consumption by number of cylinders:

```
mpg %>% group_by(cyl) %>% summarize(mean(cty), mean(hwy))
```

11.3 Repeat the last instruction, but **also include a count of the number of cars per number of cylinders**.

```
mpg %>% group_by(cyl) %>% summarize(count = n(), mean(cty), mean(hwy))
```

[Remember that instead of retyping everything you can cycle through previous commands by simply pressing the up arrow on your keyboard.]

11.4 Code can be more readable if you **spread it over multiple lines and use indentation**.

```
mpg %>%
    group_by(cyl) %>%
    summarize(
        count = n(),
        mean(cty),
        mean(hwy))
```

Note that the pipe will only work with `tidyverse` packages, and not base R.

You should now understand how to use the `filter()`, `arrange()`, `select()`, `mutate()`, `summarize()` and `group_by()` functions from the `dplyr` package, as well as the pipe operator %>%, in order to manipulate data in R.

In Tutorial 2, you will practice these skills.

ADDITIONAL CONTENT

In order to consolidate the knowledge acquired in this practical, it is strongly recommended that you complete another `swirl` course. There is a course available on `swirl` titled `Getting and Cleaning Data`. While the `R programming` course you did last week came pre-installed with `swirl`, this course needs to be manually loaded. Follow the instructions below to install this course in `swirl`. Once you've done that, you can launch `swirl` in the usual way (remember to load the library first), and the newly installed course should show up in the `swirl` course repository, alongside the `R Programming` course you've done before. Choose this new course (i.e. `Getting and Cleaning Data`). You

will see that there are 4 lessons in this course. You only have to do the first two (namely, `1: Manipulating Data with dplyr` and `2: Grouping and Chaining with dplyr`). These lessons cover exactly the same material as was covered in this practical, so it is a great opportunity to practice these new skills.

Installing `Getting and Cleaning Data` in `swirl`

Type the following to load `swirl`, install the new course and then launch `swirl`:

```
library(swirl)
install_course("Getting and Cleaning Data")
swirl()
```

If you struggle to install the course, please ask for help during the tutorial session, or post your question(s) on the discussion forum on SUNLearn.