Intro to R ENVX3003

Willem Vervoort

2022-07-28

Contents

1 Introduction

This is an introduction to R, written for ENVX3003. This work is based on earlier documents from the author and co workers (in particular, Dasapta Erwin Irawan from ITBandung, Thomas Bishop and Floris van Ogtrop) and it also builds on many of the introduction to R literature on CRAN and elsewhere on the internet.

This course is not a complete introduction, and more in depth knowledge on R and the use of R can be gained from many courses on-line and by basic practice.

This course covers three main elements: reading in files, manipulating data and doing some plotting. It particularly uses the package `tidyverse` and Rstudio. More detail on how to use `tidyverse` is on this website.

We hope that this introduction offers sufficient depth to at least get you started with R and maybe later explore this in more depth yourself.

2 R as a modelling environment

The origins of R are in statistics, so this is what R does best. However, over time, it has proven to be a flexible language that can also be used quite effectively for programming and data science.

The key power of R is in using "scripts" and "notebooks" or "rmarkdown" files which are ways to record and document the code you are generating. In this course we will focus on using rmarkdown and notebooks.

2.1 R and R Studio

2.1.1 Base R vs IDE

If R is the machine under the hood, then R Studio (and in this case Ed Saturn notebooks) would be the dashboard, steering wheel, as well as the gas and brake paddles. People frequently refer to R as `base R` and things like R Studio is an Integrated Development Environment (IDE).

Are there other IDEs than R Studio? The answer is Yes. You could check out R Commander. Another interesting project is or Microsoft R Open, which offers a multithreaded version of R.

2.1.2 Running R online

Can we run R online? The answer is also Yes. R Studio offers a paid cloud service. You could try R fiddle for a limited range of code of package installation, CoCalc/Sage Math Cloud, Jupyter, and Code Ocean.

2.1.3 R is cross platform

R and R Studio are cross platform. So you could use R on all major operating systems (OS): Windows, Mac or Linux, so it's OK if you work with another person who doesn't use the same OS as you do. You just have to make sure that all parties have the same data and the same packages installed in the system, and the same code to run.

2.1.4 R components

In R, as in any other programming language, the two main components are the data and the codes. Using both, you could start an analysis and produce plots and tables as outputs. However in order to do some of the analyses, we will need `packages`. Packages are collections of functions that we can use in R.

The good thing about R is, there are *base functions*, that is commands that are included in the base R installation. This commands are progressing as you install newer versions of R. It's getting better and easier through time. But, because R is open source, users can develop their own scripts and functions or sets of functions. Sets of functions can be grouped as a *package*. So you would need to install the package first and load the package, before using the command or function inside that package. You would only need to install the package only once.

Generally we install a package from CRAN server using this basic code. Here `packageName` is the name of the package that you would like to install.

```
install.packages("packageName") # case sensitive and you need quotes
library(packageName) # to load the package, case sensitive
```

2.1.5 Navigation

If you use R Studio, you can see four panels (clock-wise): A *script* panel on top right, Environment, Files/folder/plots/packages, and console. You write your lines of code in the script panel and then, click the *run* button (or select code and press CTRL+ENTER, or CMD+ENTER) to run and observe the progress of your code in the console panel. Find out in the console, if your code is running well or has a problem (error messages), or just a warning. Then you could see all the *objects* and loaded data components that relate to your code in the Environment panel.

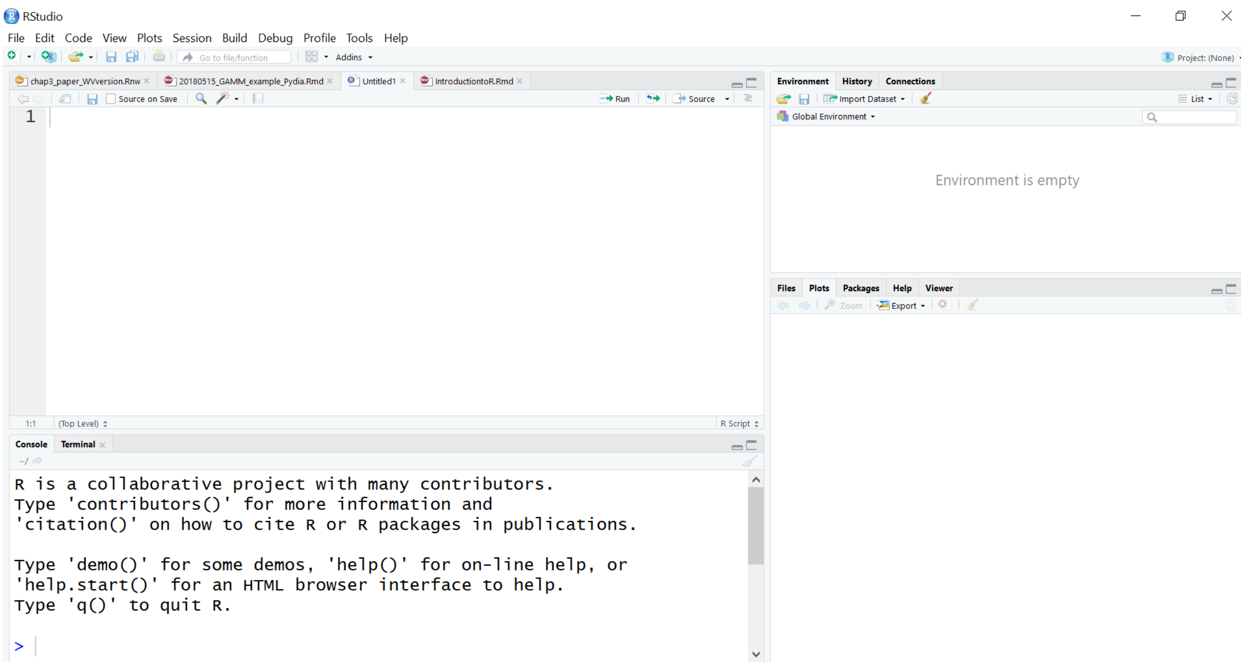


Figure 1: Four panels in R Studio

3 Working folder structure

In R and in any other command line-based application, you would need to tell the program what your current folder location is and what the location of any data is. This is one of the most important steps in the process and many of the errors generated relate to not correctly define the directory. Usually we use the following folder structure:

- main project folder
- data: put your data here
- code or scripts: put your code here
- output: put your plots and tables here

However often we work with scripts, data, outputs in one folder, but use it as a process or intermediate folder. We usually sort out the components at the final stage of our work.

3.1 The working directory

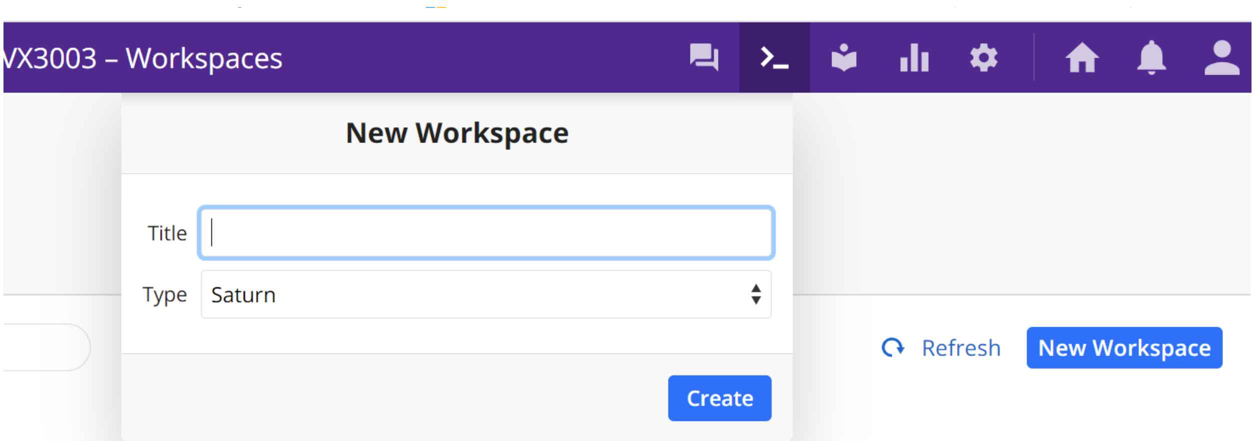
Generally R works from a "working directory". This is the directory on disk where it expects to find files or write files to. You can set this in Rstudio via the menu item "Session" –> "Set working directory", but you can also set this using code in your script. Setting the working directory is useful when you want to access data in files on your computer or the network.

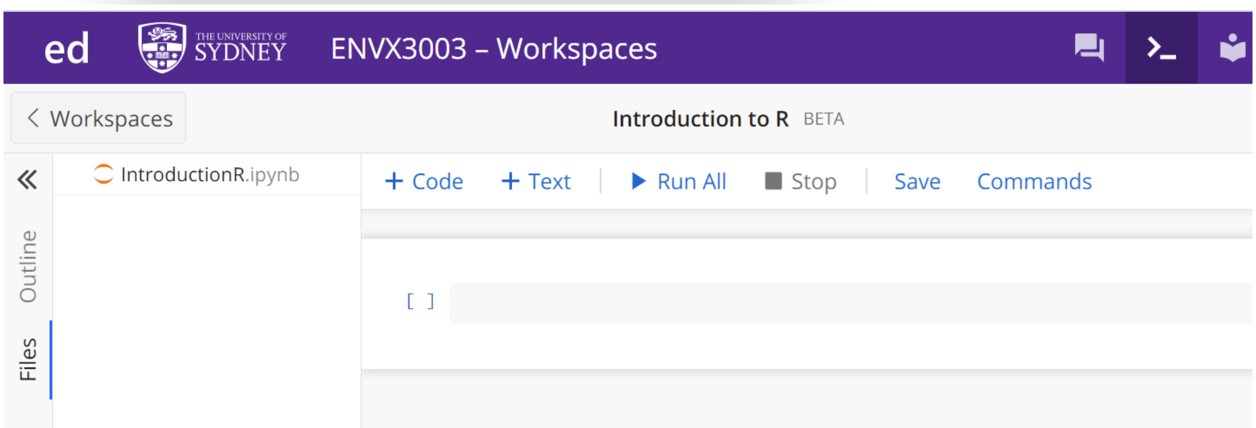
The basic function to use is `setwd("path/to/file")`. The thing to note is that in the path description you have to use "forward /" rather than the standard windows "backward".

However, if you are using an Rmarkdown file, you are generally discouraged to use "setwd", but here we can use another command which is part of the `knitr` package

3.2 Saturn notebooks in Edstem

Saturn notebooks are a special version of Jupyter notebooks that allow collaboration between students and staff. The notebooks can be created in the workspaces of your Ed environment





A Saturn notebook allows you to put in blocks of text or code. The workspace is directly linked to the notebook and files can be uploaded by right clicking in the files space on the left.

3.3 Rmarkdown notebooks in Rstudio

Rmarkdown notebooks in Rstudio are very similar to notebooks in Ed (in fact you can also open up a notebook in Rstudio). The nice thing about Rmarkdown is that you can also "publish" the notebook as an html file (webpage) or even as a word file or pdf file. In fact, this practical handout was written in R markdown. We will stick with html for the practical.

A new notebook can be opened from the file menu in R studio:

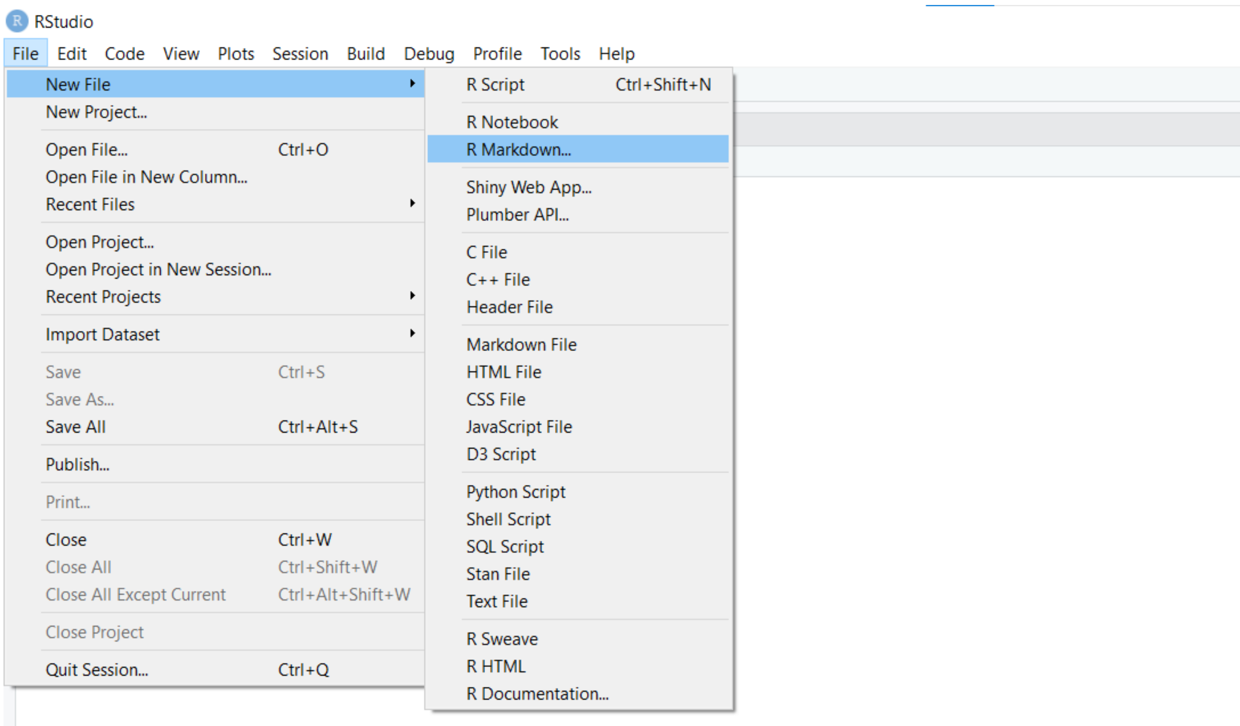


Figure 2: Create a new Rmarkdown notebook

and you have to give a title and your name (but you can change this later) which are inserted into the 'yaml header':

The Rmarkdown notebook comes with some example code below the 'yaml header' which you can delete if you want. Similar to the Ed Saturn notebooks you can write text and use code blocks. Use "CTRL-I" to insert a new code block in the notebook.

3.3.0.1 Exercise

- Can you check your working folder/directory and what's inside it?

in R:

```
getwd() # this tells you your  current working directory
dir() # this gives you a list of the files in your current working directory
?setwd # this give you the help file on how to set your working directory in code
```

in Rmarkdown you can use `opts_knit$set(root.dir ="path/to/files")` to set your working directory.

```
knitr::opts_knit$set(root.dir = "c:/users/rver4657/Desktop") ## windows users
knitr::opts_knit$set(root.dir = "~/Desktop") ## Mac users
dir() # this gives you a list of the files in your current working directory
```

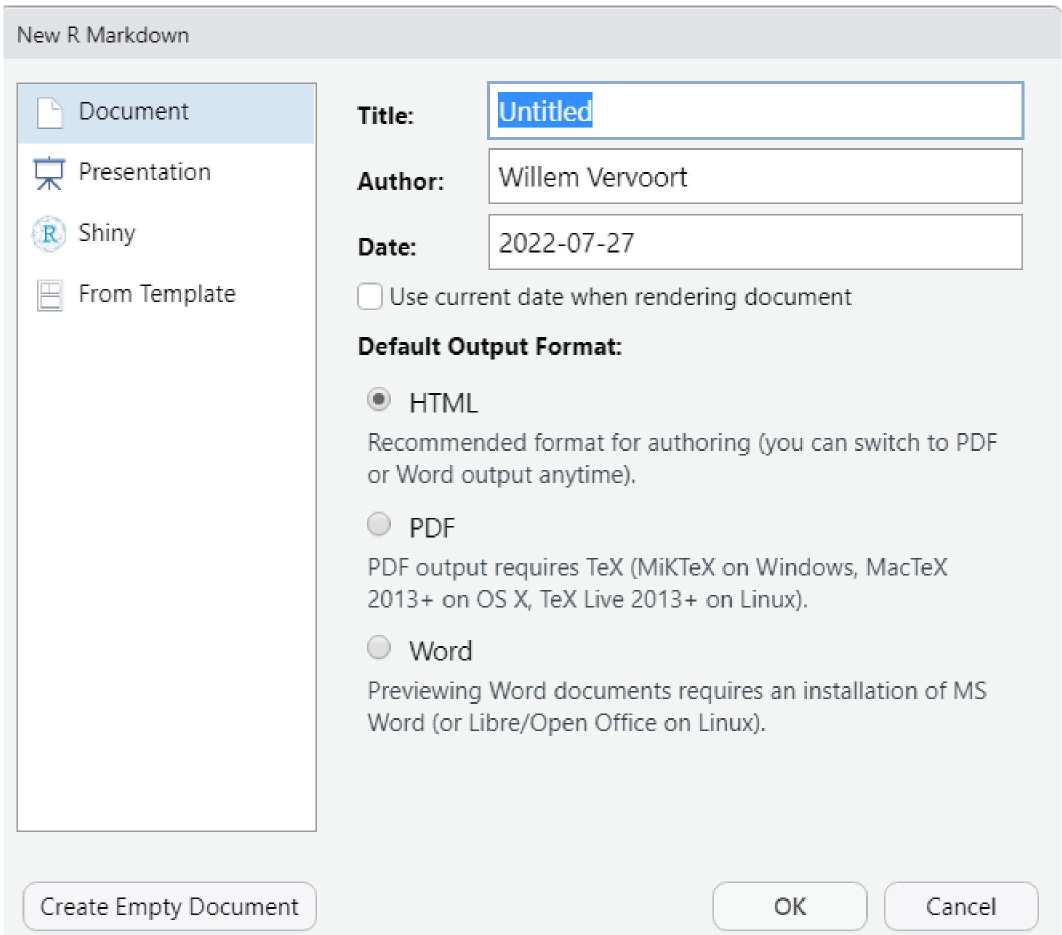


Figure 3: Name for new Rmarkdown notebook

4 BASIC R

4.1 R as a calculator

In its most basic form, R is a calculator

```
3*5
```

```
## [1] 15
```

```
50/100 + 0.1
```

```
## [1] 0.6
```

```
10 - 20
```

```
## [1] -10
```

4.2 Objects in R

The basic structure of R is based on objects, which are named. **R is case sensitive**, so keep this in mind. The main object we will use here is a *dataframe* or its modern variant the *tibble* in the package `tidyverse`.

All objects in R exist in the local R memory. So if you have a datafile, the first thing to do is to load it on your memory as an object that can be seen in the Environment panel. Thus, whatever you do with the object will not change your file, unless you save the object as a file.

As convention, R uses "<-" to assign a value (or another object) to an object. You may find "=" works just the same, but we don't recommend it, because you also use "=" with different meaning in the command and parameter setting.

```
# assign
x <- 5
y <- 2
```

You can call up what is stored in the object (inspect) again by just typing its name:

```
x
```

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

These objects will show up in the "Environment" window in Rstudio, or you can use `ls()` in the console to list the objects. The function `c()` can be used to stick things together into a vector. Redo the below commands in your own script.

```
# a vector
x <- c(1,2,5,7,8,15,3,12,11,19)
# another vector
y <- 1:10
# you have now two objects
ls()
```

```
## [1] "x" "y"
```

# you can add, multiply or subtract
z <- x + y
z
```

```
##  [1]  2  4  8 11 13 21 10 20 20 29
```

```r
zz <- x * y
zz
```

```
##  [1]   1   4  15  28  40  90  21  96  99 190
```

```r
zzz <- x - y
zzz
```

```
##  [1]  0  0  2  3  3  9 -4  4  2  9
```

```r
foo <- 0.5*x^2 - 3*x + 2
foo
```

```
##  [1]  -0.5  -2.0  -0.5   5.5  10.0  69.5  -2.5  38.0  29.5 125.5
```

- How many objects are now in your environment?

5 STATISTICAL ANALYSIS AND DATA MANIPULATION

Now it's time to look a bit further. In particular, we might be interested in how we might manipulate data so we can perform some analyses to answer our research or practical problem. There are, of course, base R commands to do the job, but because of the growth in use and the popularity, you might find it easier to use the `tidyverse` package. This package is actually a combo of several packages originally written by the same author, but now expanded by mutiple authors. There is also a nice link to this on-line book R for Data science, which is good for background information and to expand your knowledge.

5.1 Packages to use

As highlighted, much of the power in R comes from the fact that it is open source and this means many people write new code and share this code. The formal way to do this is via "packages", which, once checked and endorsed by the R community, appear in the CRAN repository as a **package**.

In this introduction we want to use some of the features in the package tidyverse. This package includes a series of other useful packages that you might need.

There are two components to using packages. The first is to make sure that the package is installed, for which we can use the functions `install.packages()`. Note that the name of the package is a *string* so needs to between quotes `""`.

Within the Edstem environment, we have made sure that those packages are installed, so you don't need to separately install those packages.

```r
install.packages("tidyverse")
```
```

If the package is installed in your personal library, you will need to load the package in R using `require()` or `library()`. There are subtle differences between these two functions, but they are currently not that important. Check the help files.

```
require(tidyverse)
```

5.1.1 Optional Exercise

- Can you load (and maybe first install) the package `lubridate`? This package is great for manipulating dates and times and works well with `tidyverse`.

5.2 A dataframe in tidyverse

A dataframe is a bit more complex. It is essentially a list, but presented as a table. Here is a simple demonstration of its power. We are using `tibble()` from tidyverse, which is a funny name, but is very similar to `data.frame()` in base R.

```
Rainfall <- tibble(City = c("Montevideo","New York",
                                "Amsterdam","Sydney",
                                "Moscow", "Hong Kong"),
                       Rain_mm = c(950, 1174, 838, 1215,
                                     707, 2400))
Rainfall
```

```
## # A tibble: 6 x 2
##   City       Rain_mm
##   <chr>        <dbl>
## 1 Montevideo     950
## 2 New York      1174
## 3 Amsterdam      838
## 4 Sydney        1215
## 5 Moscow         707
## 6 Hong Kong     2400
```

As you can see a data frame (in this case a `tibble`) can mix character columns (City) and numeric columns (Rain_mm). Here I used `c()` to generate vectors of numbers of characters which I put in the columns. In addition, the colummns have names, which you can access using `colnames()` or names:

```
colnames(Rainfall)
```

```
## [1] "City"    "Rain_mm"
```

```
names(Rainfall)
```

```
## [1] "City"    "Rain_mm"
```

Once you have a data frame or tibble, you can access parts of the dataframe or manipulate the dataframe. Such as finding a column

```r
# call a column
Rainfall$City
```

```
## [1] "Montevideo" "New York"   "Amsterdam"  "Sydney"     "Moscow"
## [6] "Hong Kong"
```

```r
# or
Rainfall["City"]
```

```
## # A tibble: 6 x 1
##   City
##   <chr>
## 1 Montevideo
## 2 New York
## 3 Amsterdam
## 4 Sydney
## 5 Moscow
## 6 Hong Kong
```

```r
# or
Rainfall[,1]
```

```
## # A tibble: 6 x 1
##   City
##   <chr>
## 1 Montevideo
## 2 New York
## 3 Amsterdam
## 4 Sydney
## 5 Moscow
## 6 Hong Kong
```

Sub-setting rows is slightly different, you can still use numbers

```r
# see the first two rows
Rainfall[1:2,]
```

```
## # A tibble: 2 x 2
##   City       Rain_mm
##   <chr>        <dbl>
## 1 Montevideo     950
## 2 New York      1174
```

5.2.1 filter()

But, because this is a tibble, we can use the power of tidyverse and the function `filter` to find rows

```r
Rainfall %>%
  filter(City=="Montevideo")
```

```
## # A tibble: 1 x 2
##   City       Rain_mm
##   <chr>        <dbl>
## 1 Montevideo     950
```

OK, what is that `%>%` thing?? It is a further little symbol (apart from assign `<-`) in R that you need to know. It means "then".

So to read the above code in words, it says:

- Take the Rainfall data, **then**, find the row where City equals "Montevideo"

Let's expand this idea and show another `filter`:

```
# find a subset
lots <- Rainfall %>%
  filter(Rain_mm > 1000)
lots
```

```
## # A tibble: 3 x 2
##   City      Rain_mm
##   <chr>       <dbl>
## 1 New York     1174
## 2 Sydney       1215
## 3 Hong Kong    2400
```

Here I did two things, I did the filter using a comparison (and got multiple rows) and I "assigned" the result to a new object `lots`.

5.2.2 select()

Similar to filter, we can extract columns using `select`. So repeating the above example:

```
# call a column
Rainfall %>%
  select(City)
```

```
## # A tibble: 6 x 1
##   City
##   <chr>
## 1 Montevideo
## 2 New York
## 3 Amsterdam
## 4 Sydney
## 5 Moscow
## 6 Hong Kong
```

5.2.3 Exercise

Using the above examples, can you do the following?

- Extract the column with the rainfall values?
- Extract the row with the annual rainfall at Amsterdam and assign to an object?
- Create an object with cities that have rainfall below 1500 mm?

5.3 Reading data from different sources

There are a multitude of functions to read data from the disk into the R memory, I will demonstrate only one here, but more are given in the tidyverse book

Because a lot of data is stored in comma delimited txt files (such as Excel exports), using `read_csv()` is a good standard option.

Here I am reading in some monthly data from the Concordia station in the Uruguay river in Argentina. This data was originally downloaded from the Global River Discharge Database

```
UR_flow <- read_csv("Data/UruguayRiver_ConcordiaSt.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Year = col_double(),
##   Month = col_double(),
##   Flow = col_double()
## )
```

```
# check the first few lines (6 by default)
UR_flow
```

```
## # A tibble: 132 x 3
##     Year Month  Flow
##    <dbl> <dbl> <dbl>
##  1  1969     1  7888
##  2  1969     2  5951
##  3  1969     3  4296
##  4  1969     4  4173
##  5  1969     5  4539
##  6  1969     6  4857
##  7  1969     7  4018
##  8  1969     8  3110
##  9  1969     9  2541
## 10  1969    10  2822
## # ... with 122 more rows
```

Previously you would have to save a specific program's data file, say in *xls* in to a pure text file such as *csv* or *txt*. However, there are now many packages that allow you to read a dataset directly from its binary format. There are many packages to do such task, readxl package is one of them. You could google your way to the most convenient package to use.

5.3.1 Exercise

- Can you read in the file: "Parana_CorrientesSt.csv" (supplied) from your data directory?

5.4 More on data manipulation (using `tidyverse`)

5.4.1 Important commands

Based on the above, and building on this, the following list are the most used commands in tidyverse:

- `select()` select columns
- `filter()` filter rows
- `arrange()` re-order or arrange rows (row sorting)
- `mutate()` create new columns
- `summarise()` summarise values
- `group_by()` allows for group operations in the "split-apply-combine" concept

5.4.1.1 mutate, arrange and select Here is a short demonstration for `mutate`, `select` and `arrange` using the Rainfall data:

`mutate` and `select` to add columns, select columns and rearrange columns

```
# I can add a column of countries
Rainfall_new <- Rainfall %>%
  mutate(country = c("UY", "US", "NL", "AU", "RU","CN")) %>%
  # and maybe a column of the average monthly rainfall
  mutate(M_rain = Rain_mm/12)

# You can use select to reorder the columns and put country to the front
Rainfall_new <- Rainfall_new %>%
  select(country, everything())

# And if you would like to drop the M_rain column you can use
Rainfall_new %>%
  select(-M_rain)
```

```
## # A tibble: 6 x 3
##   country City       Rain_mm
##   <chr>   <chr>        <dbl>
## 1 UY      Montevideo     950
## 2 US      New York      1174
## 3 NL      Amsterdam      838
## 4 AU      Sydney        1215
## 5 RU      Moscow         707
## 6 CN      Hong Kong     2400
```

`arrange` to sort rows, and add `desc()` to do this in decreasing order

```
Rainfall_new %>%
  arrange(desc(Rain_mm))
```

```
## # A tibble: 6 x 4
##   country City       Rain_mm M_rain
##   <chr>   <chr>        <dbl>  <dbl>
## 1 CN      Hong Kong     2400  200
## 2 AU      Sydney        1215  101.
```

```
## 3 US      New York      1174   97.8
## 4 UY      Montevideo     950   79.2
## 5 NL      Amsterdam      838   69.8
## 6 RU      Moscow         707   58.9
```

5.4.1.2 summarise and group_by Another useful function is `summarise()`, which allows you to apply a function over a data frame and particular across different factors. In tidyverse this is often combined with the function `group_by` to define how you would like to summarise.

Here is an example of summing the Uruguay river flow by year. Note that putting brackets around the statement makes it print out the result.

```
# aggregate to annual flow
(annual_flow <- UR_flow %>% #then
  group_by(Year=Year) %>% #then
  summarise(Sumflow = sum(Flow)))
```

```
## # A tibble: 11 x 2
##     Year Sumflow
##    <dbl>   <dbl>
##  1  1969   53753
##  2  1970   52130
##  3  1971   66648
##  4  1972   99562
##  5  1973  103070
##  6  1974   47130
##  7  1975   68075
##  8  1976   53500
##  9  1977   73650
## 10  1978   41700
## 11  1979   61047
```

If you just wanted the overall mean and standard deviation of monthly flow, you could also ask for (and note how we drop group_by):

```
# Mean and sd\ monthly flow
UR_flow %>% #then
  summarise(Meanflow = mean(Flow),
            SdFlow = sd(Flow))
```

```
## # A tibble: 1 x 2
##   Meanflow SdFlow
##      <dbl>  <dbl>
## 1    5457.  3492.
```

5.4.2 Exercise

- Can you calculate the standard deviation of the monthly flow by year using `summarise()`?

5.5 Different data set for practice

Let's open a water quality data set in `csv` format.

Here we will introduce some groundwater chemical data from Semarang in Indonesia, kindly supplied by Dasapta Erwin Irawan from ITB

```
chemdata <- read_csv("data/semarang_chem.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double(),
##   ID = col_character(),
##   Area = col_character(),
##   UTM_zone = col_character(),
##   Aq = col_character(),
##   Fac = col_character()
## )
## i Use `spec()` for the full column specifications.
```

```
chemdata
```

```
## # A tibble: 58 x 24
##    ID     Area    Year   Lat  Long UTM_east UTM_north UTM_zone Depth    WL  Elev
##    <chr>  <chr>  <dbl> <dbl> <dbl>    <dbl>     <dbl> <chr>    <dbl> <dbl> <dbl>
##  1 SB_185 PT. N~  1992 -6.96  110.   439936   9230800 49M         96 23.4      2
##  2 SB_273 PT. I~  1992 -6.98  110.   438500   9228100 49M         94 14.4     20
##  3 SB_283 Obs. ~  1992 -6.96  110.   435500   9230150 49M        150 15.2      2
##  4 SB_271 PT. S~  1992 -6.98  110.   424150   9228400 49M         65 31.5     25
##  5 SB_270 Dolog~  1992 -6.97  110.   421950   9229400 49M         NA 19.8     20
##  6 SB_278 Hotel~  1992 -6.99  110.   436950   9226950 49M         86  7.86     5
##  7 SB_325 PT Wa~  1992 -6.99  110.   427950   9226950 49M         76 44.6     72
##  8 SB_190 PT. G~  1992 -6.98  110.   436900   9228050 49M         NA 22.1      4
##  9 SB_256 Tamba~  1992 -6.98  110.   429800   9228600 49M         NA  4.64     2
## 10 SB_206 Tamba~  1992 -6.95  110.   423100   9231700 49M         80 11.6      1
## # ... with 48 more rows, and 13 more variables: TDS <dbl>, ph <dbl>, EC <dbl>,
## #   K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>, Cl <dbl>, HCO3 <dbl>,
## #   Bal <dbl>, Aq <chr>, Fac <chr>
```

5.5.1 `select()`

A bit more on select. If you want multiple columns, for example `Lat`, `Long` until `Depth`, you can simply use the `select()` function.

```
chemdata %>%
  select(Lat, Long:Depth)
```

```
## # A tibble: 58 x 6
##      Lat  Long UTM_east UTM_north UTM_zone Depth
##    <dbl> <dbl>    <dbl>     <dbl> <chr>    <dbl>
##  1 -6.96  110.   439936   9230800 49M         96
```

```
##  2 -6.98  110.   438500   9228100 49M         94
##  3 -6.96  110.   435500   9230150 49M        150
##  4 -6.98  110.   424150   9228400 49M         65
##  5 -6.97  110.   421950   9229400 49M         NA
##  6 -6.99  110.   436950   9226950 49M         86
##  7 -6.99  110.   427950   9226950 49M         76
##  8 -6.98  110.   436900   9228050 49M         NA
##  9 -6.98  110.   429800   9228600 49M         NA
## 10 -6.95  110.   423100   9231700 49M         80
## # ... with 48 more rows
```

Or you want multiple columns Lat, Long until Depth, but you don't want UTM_zone. Again, you can use the `select()` function.

```
chemdata %>%
  select(Lat, Long:Depth, -UTM_zone)
```

```
## # A tibble: 58 x 5
##      Lat  Long UTM_east UTM_north Depth
##    <dbl> <dbl>    <dbl>     <dbl> <dbl>
##  1 -6.96  110.   439936   9230800    96
##  2 -6.98  110.   438500   9228100    94
##  3 -6.96  110.   435500   9230150   150
##  4 -6.98  110.   424150   9228400    65
##  5 -6.97  110.   421950   9229400    NA
##  6 -6.99  110.   436950   9226950    86
##  7 -6.99  110.   427950   9226950    76
##  8 -6.98  110.   436900   9228050    NA
##  9 -6.98  110.   429800   9228600    NA
## 10 -6.95  110.   423100   9231700    80
## # ... with 48 more rows
```

5.5.2 arrange()

Sorting the data by the Aq (aquifer) and Fac (facies) columns. Use the `arrange()` function.

```
chemdata %>%
  arrange(Aq, Fac)
```

```
## # A tibble: 58 x 24
##    ID     Area    Year   Lat  Long UTM_east UTM_north UTM_zone Depth    WL  Elev
##    <chr>  <chr>  <dbl> <dbl> <dbl>    <dbl>     <dbl> <chr>    <dbl> <dbl> <dbl>
##  1 SB_271 PT. S~  1992 -6.98  110.   424150   9228400 49M         65 31.5     25
##  2 SB_270 Dolog~  1992 -6.97  110.   421950   9229400 49M         NA 19.8     20
##  3 SB_325 PT Wa~  1992 -6.99  110.   427950   9226950 49M         76 44.6     72
##  4 SB_225 PDAM ~  1992 -7.00  110.   431850   9226100 49M        100 56.6     65
##  5 SB_92  RS Ka~  1992 -6.99  110.   434400   9227200 49M         75  6.8      6
##  6 SB_112 Hotel~  1993 -7.00  110.   435763   9226415 49M        122 25.0     30
##  7 SB_215 S. Pa~  2003 -7.00  110.   432811   9225924 49M        152  5.97    18
##  8 SB_33  Es Pr~  2003 -6.99  110.   429310   9227496 49M         90 22.5     25
##  9 SB_590 Bukit~  2003 -6.99  110.   427050   9227750 49M         90 23.6     25
## 10 SP_341 Obs. ~  2003 -6.99  110.   426474   9227278 49M         NA 56.9     40
```

```
## # ... with 48 more rows, and 13 more variables: TDS <dbl>, ph <dbl>, EC <dbl>,
## #   K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>, Cl <dbl>, HCO3 <dbl>,
## #   Bal <dbl>, Aq <chr>, Fac <chr>
```

5.5.3 mutate()

Making new columns, for instance, calculating the ratio between Ca and Na. Use the `mutate() function`

```
chemdata %>%
  mutate(ratio_Cana = Ca / Na)
```

```
## # A tibble: 58 x 25
##    ID     Area    Year   Lat  Long UTM_east UTM_north UTM_zone Depth    WL  Elev
##    <chr>  <chr>  <dbl> <dbl> <dbl>    <dbl>     <dbl> <chr>    <dbl> <dbl> <dbl>
##  1 SB_185 PT. N~  1992 -6.96  110.   439936   9230800 49M         96 23.4      2
##  2 SB_273 PT. I~  1992 -6.98  110.   438500   9228100 49M         94 14.4     20
##  3 SB_283 Obs. ~  1992 -6.96  110.   435500   9230150 49M        150 15.2      2
##  4 SB_271 PT. S~  1992 -6.98  110.   424150   9228400 49M         65 31.5     25
##  5 SB_270 Dolog~  1992 -6.97  110.   421950   9229400 49M         NA 19.8     20
##  6 SB_278 Hotel~  1992 -6.99  110.   436950   9226950 49M         86  7.86     5
##  7 SB_325 PT Wa~  1992 -6.99  110.   427950   9226950 49M         76 44.6     72
##  8 SB_190 PT. G~  1992 -6.98  110.   436900   9228050 49M         NA 22.1      4
##  9 SB_256 Tamba~  1992 -6.98  110.   429800   9228600 49M         NA  4.64     2
## 10 SB_206 Tamba~  1992 -6.95  110.   423100   9231700 49M         80 11.6      1
## # ... with 48 more rows, and 14 more variables: TDS <dbl>, ph <dbl>, EC <dbl>,
## #   K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>, Cl <dbl>, HCO3 <dbl>,
## #   Bal <dbl>, Aq <chr>, Fac <chr>, ratio_Cana <dbl>
```

5.5.4 summarise()

Making a summary from your data. Use `summarise()` function.

```
chemdata %>%
  summarise(mean_TDS = mean(TDS),
            max_Cl = max(Cl),
            min_Cl = min(Cl),
            total = n())
```

```
## # A tibble: 1 x 4
##   mean_TDS max_Cl min_Cl total
##      <dbl>  <dbl>  <dbl> <int>
## 1    1041. 15753.   11.2    58
```

5.5.5 group_by()

Sorting out the data based on certain order. Use `group_by()` function.

```
chemdata%>%
  group_by(Aq) %>%
  summarise(mean_TDS = mean(TDS),
            max_Cl = max(Cl),
```

```
          min_Cl = min(Cl),
          total = n())
```

```
## # A tibble: 3 x 5
##   Aq                  mean_TDS max_Cl min_Cl total
##   <chr>                  <dbl>  <dbl>  <dbl> <int>
## 1 Damar                   371.     70   11.2    14
## 2 Garang                  445.    146.  19.6    11
## 3 Quaternary marine      1523. 15753.   25      33
```

5.5.5.1 Exercise

- Can you calculate the mean(Cl) and sd(Na) for the dataset grouped by Fac?

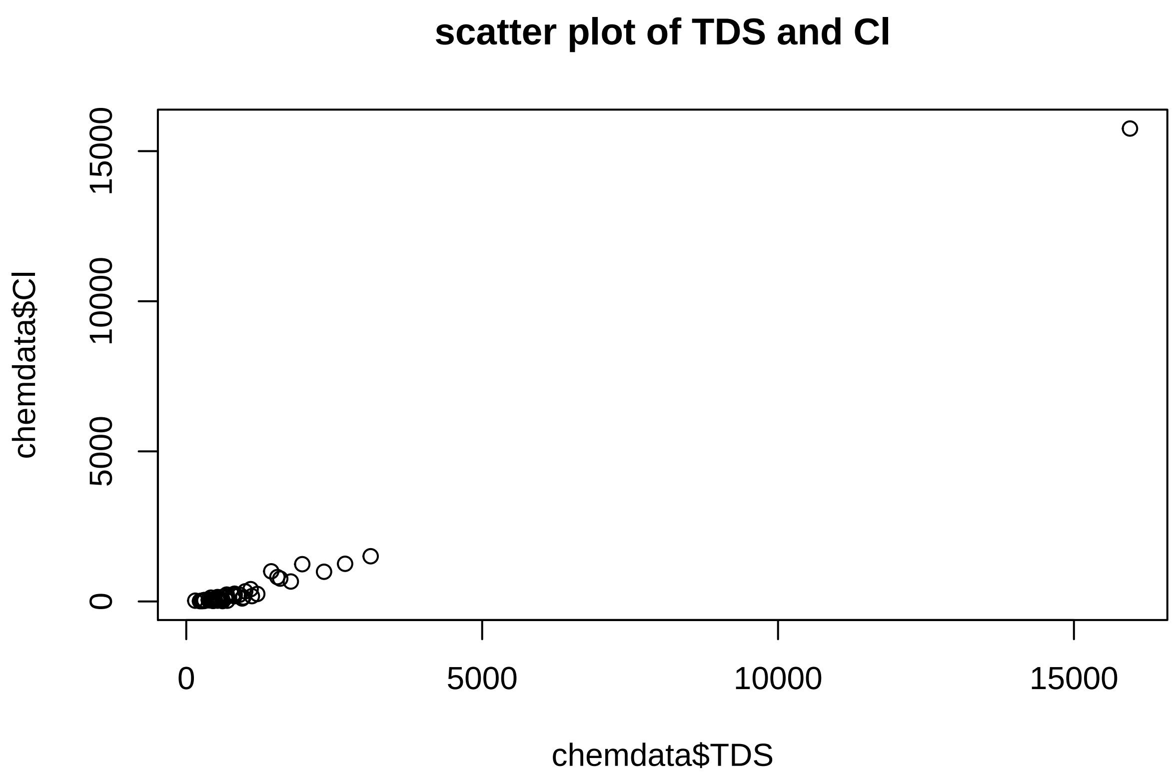
As we have indicated earlier, be sure to check out R for Data Science for more info about `tidyverse` and its use in data science.

6 PLOTTING

R is good at plotting. There are many ways to create a plot. So you just have to choose which one is the easiest for you. One way is using base R plotting engine. Like these plots.

```
hist(chemdata$SO4, main = "histogram of SO4")
```

```
plot(chemdata$TDS, chemdata$Cl, main="scatter plot of TDS and Cl")
```

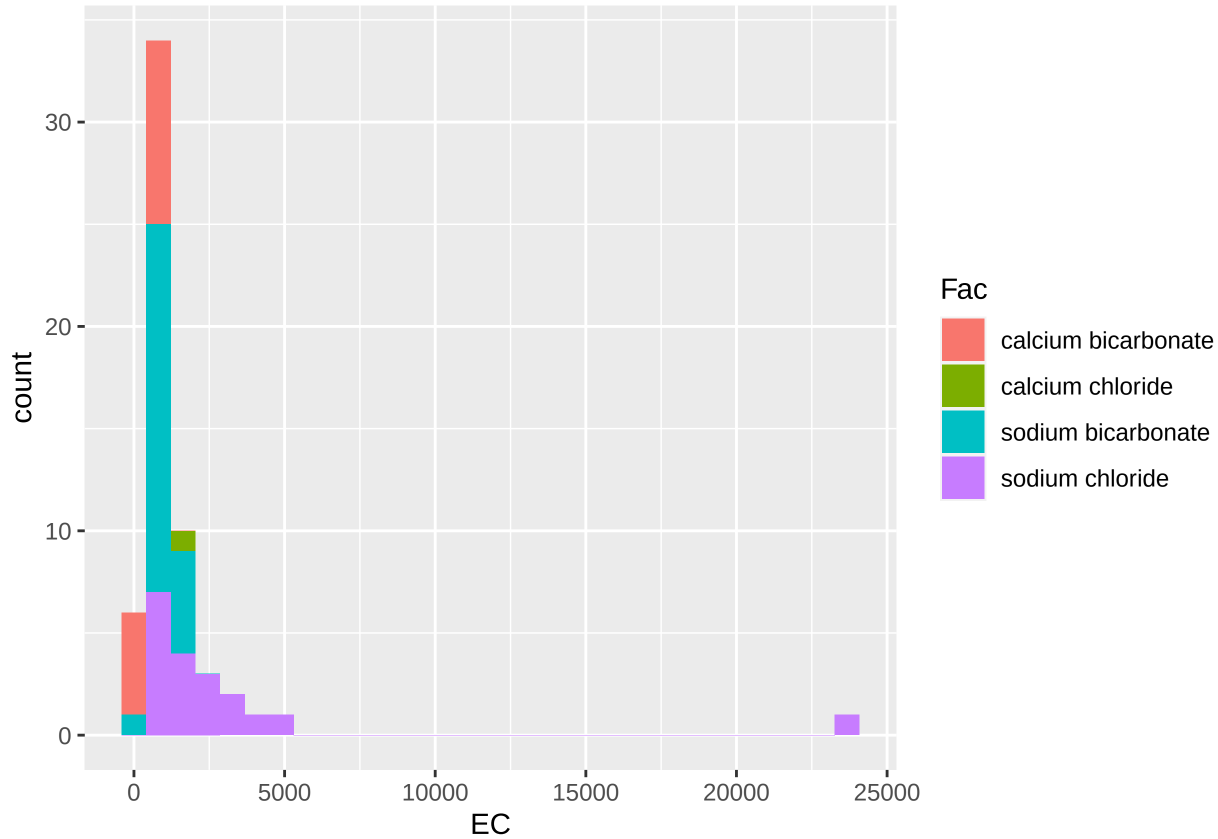


You could always tweak the plot to suits your needs. There are many resources about plotting in R on the web.

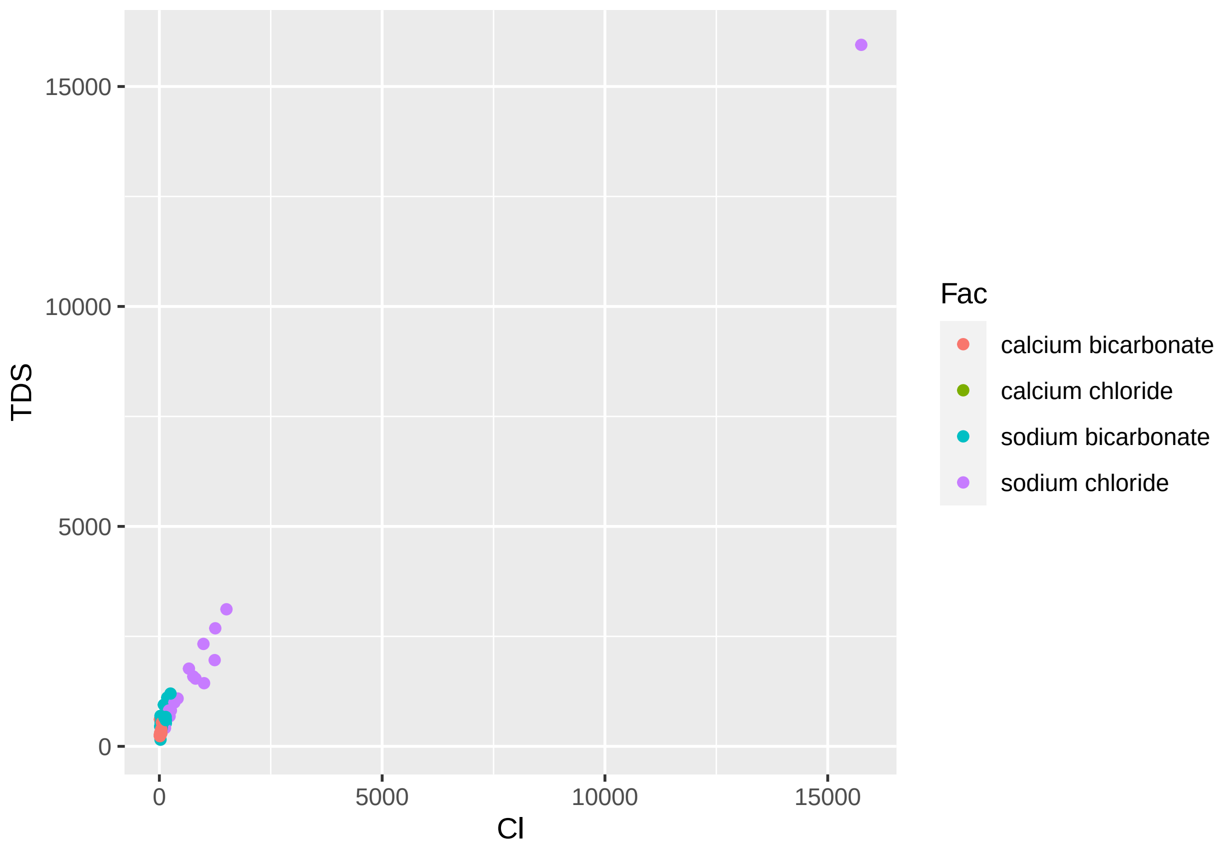
Nicer plots are made with `ggplot2` which is the plotting engine from `tidyverse`. The best book for this is the **R Graphics Cookbook**

```
chemdata  %>%
  ggplot(aes(EC, fill = Fac)) + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
chemdata %>%
  ggplot(aes(Cl, TDS, colour = Fac)) + geom_point()
```



In the second plot, note how colour is used to identify different facies.

6.1 Exercise

- try to make a plot between Ca vs Na using base R and ggplot2.
- try to make histogram for one parameter that you have in your dataset. Use ggplot2.
- can you tweak it by adding title to the plot and title to all axis. Hint: use `ggtitle()` and `labs()`.

7 Programming: if else and for loops

Loops are essential in programming. There are a range of different types of loops, but here I will only demonstrate "if else" and "for". I have always been told that all other loops are just derivatives or short cuts.
An "if else" loop allows you to program a switch in the code.
Basically it is used to evaluate an expression if the statement is TRUE and to evaluate another expression if the statement is FALSE:

if (comparison) do this, else do that

You might call the above line "pseudo code". It is sometimes handy to first write something in pseudo code, basically you want to write in broad language what you want to happen.
You could also use only the "if" part, which means nothing happens if the statement is FALSE.

As an example I want to do: *if (a data frame has more than 10 rows) write data frame is LONG, else write data frame is SHORT.*

```
#We will first generate the data frame:
x <- UR_flow
# now wrote loop
if (nrow(x) > 10) {
print("the dataframe is LONG")
} else {
print("the dataframe is SHORT")
}
```

```
## [1] "the dataframe is LONG"
```

There are a few things to note here. I use the statement `nrow()` to check how many rows the data frame has. I use the statement `print()` to write something to the screen.

7.0.1 Exercise

- Try generating another data frame x and rerun the program, or change the program to get it to say "the dataframe is short".

R also has an `ifelse()` command. This is a vectorized version of the if command - which means that it can be used on vectors of data - the command is applied to each element (or value) of the vector in turn. The if command only evaluates single values.
Using the ifelse command will return a vector of values, the same length as the longest argument in the expression.
Wherever possible, it is preferable to use the ifelse command rather than using the if command in combination with a loop - writing the program is more efficient and R evaluates vectorised functions more efficiently than it does loops. Here is an example which changes the program above.

```
x <- UR_flow
# add a column which identifies whether the flow < 5000
x[,4] <- ifelse(x[,3] > 5000, "large", "small")
# this creates a third column
tail(x,10)
```

```
## # A tibble: 10 x 4
##     Year Month  Flow V1
##    <dbl> <dbl> <dbl> <chr>
##  1  1979     3  1699 small
##  2  1979     4  1650 small
##  3  1979     5  5052 large
##  4  1979     6  2619 small
##  5  1979     7  3776 small
##  6  1979     8  6134 large
##  7  1979     9  3275 small
##  8  1979    10 15692 large
##  9  1979    11 12244 large
## 10  1979    12  7380 large
```

I check in the second column of the data.frame whether the flows are greater than 5000 or not. I then write in the third column whether they are large or small numbers. A more complex (nested) ifelse version would be:

```
x[,5] <- ifelse(x[,3] > 2500,ifelse(x[,3] > 10000, "large", "intermediate"), "small")
tail(x,10)
```

```
## # A tibble: 10 x 5
##     Year Month  Flow V1    V1
##    <dbl> <dbl> <dbl> <chr> <chr>
##  1  1979     3  1699 small small
##  2  1979     4  1650 small small
##  3  1979     5  5052 large intermediate
##  4  1979     6  2619 small intermediate
##  5  1979     7  3776 small intermediate
##  6  1979     8  6134 large intermediate
##  7  1979     9  3275 small intermediate
##  8  1979    10 15692 large large
##  9  1979    11 12244 large large
## 10  1979    12  7380 large intermediate
```

You can try out some of your own versions of this

7.1 The "for" loop, getting the program to do something repeatedly

Loops are used to repeat a set of commands. Normally, there will be a variable which changes value in each successive loop through the commands. Reference to this changing value results in differences in output from successive iterations.

The for loop is used when the number of required iterations is known before the loop begins. It is used in the following way: *for (name in expression1) {expression2}*

- name is the name of the loop variable. Its value changes during each iteration, starting with the first value and ending with the last value in expression1.
- expression1 is a vector expression (often a sequence, such as 1:10).
- expression2 is a command or group of commands that are repeatedly evaluated. It usually contains references to name, which result in changes to the value of the expression as the value of name changes.

Here is the classic example of a loop

```
# Hello world
for (i in 1:5) {
print(paste(i, "hello world"))
}
```

```
## [1] "1 hello world"
## [1] "2 hello world"
## [1] "3 hello world"
## [1] "4 hello world"
## [1] "5 hello world"
```

Note the use of `paste()` to combine character vectors.
Here is another simple loop that tells you the first 5 values of the flow data.

```
for (i in 1:5) {
print(paste(UR_flow$Flow[i], "is the flow (ML/day)"))
}
```

```
## [1] "7888 is the flow (ML/day)"
## [1] "5951 is the flow (ML/day)"
## [1] "4296 is the flow (ML/day)"
## [1] "4173 is the flow (ML/day)"
## [1] "4539 is the flow (ML/day)"
```

```
# or more complex:
for (i in 1:5) {
print(paste("in Year", UR_flow$Year[i],"and month",
UR_flow$Month[i],
"the flow is", UR_flow$Flow[i], "(ML/day)"))
}
```

```
## [1] "in Year 1969 and month 1 the flow is 7888 (ML/day)"
## [1] "in Year 1969 and month 2 the flow is 5951 (ML/day)"
## [1] "in Year 1969 and month 3 the flow is 4296 (ML/day)"
## [1] "in Year 1969 and month 4 the flow is 4173 (ML/day)"
## [1] "in Year 1969 and month 5 the flow is 4539 (ML/day)"
```

You can also nest loops, that is, embed one loop into another. Here is an example that prints both the year and the flow using the column names in the dataframe.

```
for (i in 1:5) {
for (j in c(1,3)) {
print(paste(UR_flow[i,j], colnames(UR_flow)[j]))
}
}
```

```
## [1] "1969 Year"
## [1] "7888 Flow"
## [1] "1969 Year"
```

```
## [1] "5951 Flow"
## [1] "1969 Year"
## [1] "4296 Flow"
## [1] "1969 Year"
## [1] "4173 Flow"
## [1] "1969 Year"
## [1] "4539 Flow"
```

7.1.1 Exercise

- Write another program that includes a loop and a logical test

7.1.2 Comparison and Logical Operators

Comparison operators return a true or false value:

- == Equal to
- > Greater than
- >= Greater than or equal to
- < Less than
- <= Less than or equal to

Comparison operators can be combined with logical operators to describe more complex conditions.

Logical operators:

- ! Not
- or (used for vectors, with the ifelse command)
- || or (used for single values)
- & and (used for vectors, with the ifelse command)
- && and (used for single values)

7.1.3 Exercise

Write a small program that uses comparison operators and a logical operator.

8 Date and times

This is a small demonstration of the package `lubridate`, which works well with `tidyverse` and allows you to convert times and dates with less effort. There is more description about this in this chapter of the tidyverse book.

```
#install.packages("lubridated")  do this if you haven't done so.
require(lubridate)
UR_flow_d <- UR_flow %>%
  mutate(Dates = make_datetime(Year, Month))
```

Once we have this zoo data frame, it can be plotted quite easily with the basic plotting package.

```
UR_flow_d %>%
  ggplot(aes(Dates,Flow)) + geom_line(colour="blue")
```

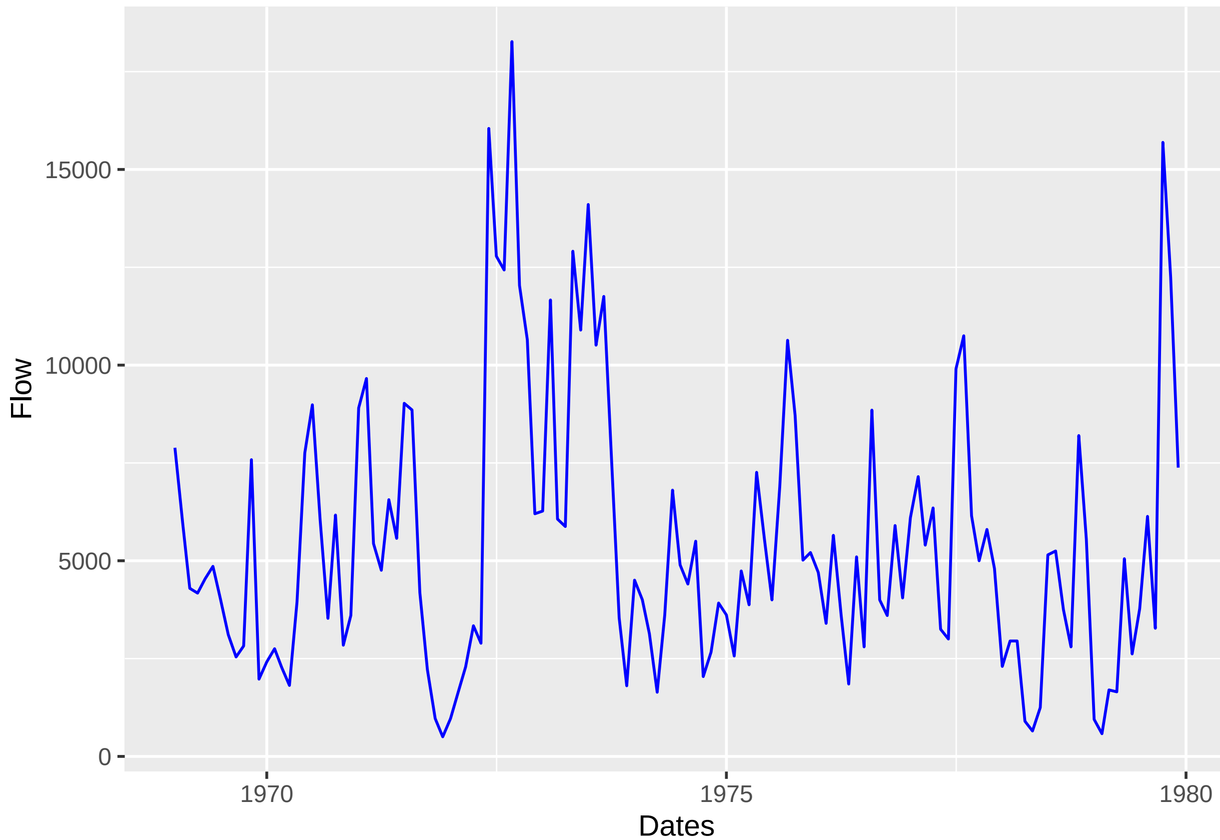


Figure 4: Demonstration of dates in plotting R

9 Writing functions in R

Until now you have used several functions in R that are part of packages or part of "core" R. However, another powerful element in R is the ability to write your own functions. There are two major advantages with writing functions:

1. They are easy to test, as they are contained. This is especially true if keep functions short.
2. They are short cuts and repeatable and therefore limit the possibility of typos.

Let's go back to the "hello world" example that we used in a loop earlier. We can write the same example in a function.
The first thing to do is to decide which inputs we want the function to use to create the output. In this case I suggest we might want to change how many times the function produces output (which was 5 in the earlier example) and the actual output text, which was "hello world" in the original function.

The basic structure of a function is:

```
NameOfFunction <- function(input1, input2,...) {
doSomething <- ....
return(doSomething)
}
```

Here is the hello world example:

```r
HW <- function(n, outtext) {
  for (i in 1:n) {
    print(outtext)
  }
  # return("nothing")
}

# test
HW(5, "Hello World")
```

```
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
```

```r
# switch input by naming
HW(outtext ="I can switch the inputs", n = 3)
```

```
## [1] "I can switch the inputs"
## [1] "I can switch the inputs"
## [1] "I can switch the inputs"
```

Note that in this case the function produces output as part of its execution rather than returning an actual value (which is why I commented out the `return` statement). In the first example, you can see that you don't have to name the inputs if you keep the inputs in the same order as the defined function. R assumes that you mean `n = 5` and `outtext = "Hello world"`. In the second example I show that you can switch the inpurs if you name them and that the function allows you to choose different inputs.

9.0.1 Exercise

- Can you write a function that calculates y = a*x + b for different values of x, a and b?
- Make the function return the output using return()

END OF DOCUMENT