Intro to R

Willem Vervoort and Dasapta Erwin Irawan

2018-05-20

Contents

1 Introduction

This is an introduction to R, originally written for the "Open data workshop in Bandung 5-9 Feb 2018", jointly organised by the University of Sydney and Institut Teknologi Bandung and funded by the Sydney SouthEast Asia Centre.

This work is based on earlier documents from the authors and it also builds on many of the introduction to R literature 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 simple R, basic statistics, data frame operations, reading in files and a plotting. It includes an introduction into the package `tidyverse`. 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 hat 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.

2.1 R and R Studio

2.1.1 Base R vs IDE

If R is the machine under the hood, then R Studio would be the dashboard, steering wheel, as well as the gas and brake paddles. People frequently mention R as `base R` and R Studio is an Integrated Development Environment (IDE).

Is there another IDE other than R Studio? The answer is Yes. You could check out R Commander, Revolution 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 the New Comer Code Ocean.

2.1.3 R is cross platform

R and R Studio are cross platform. So you could use R on these major 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 How to install R and R Studio

We recommend to install R first followed by R Studio. Install R from CRAN and R Studio from its official site.

2.1.5 R components

In R programming, as also in 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`.

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 function 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.

You could run this line to install a package from CRAN server. install.packages("packageName") # case sensitive library(packageName) # to load the package Other than CRAN, you may find packages that are still in development stage on GitHub, a repository of code and a tool for code management. You could install a package on GitHub using `install_github` command from `devtools` package.

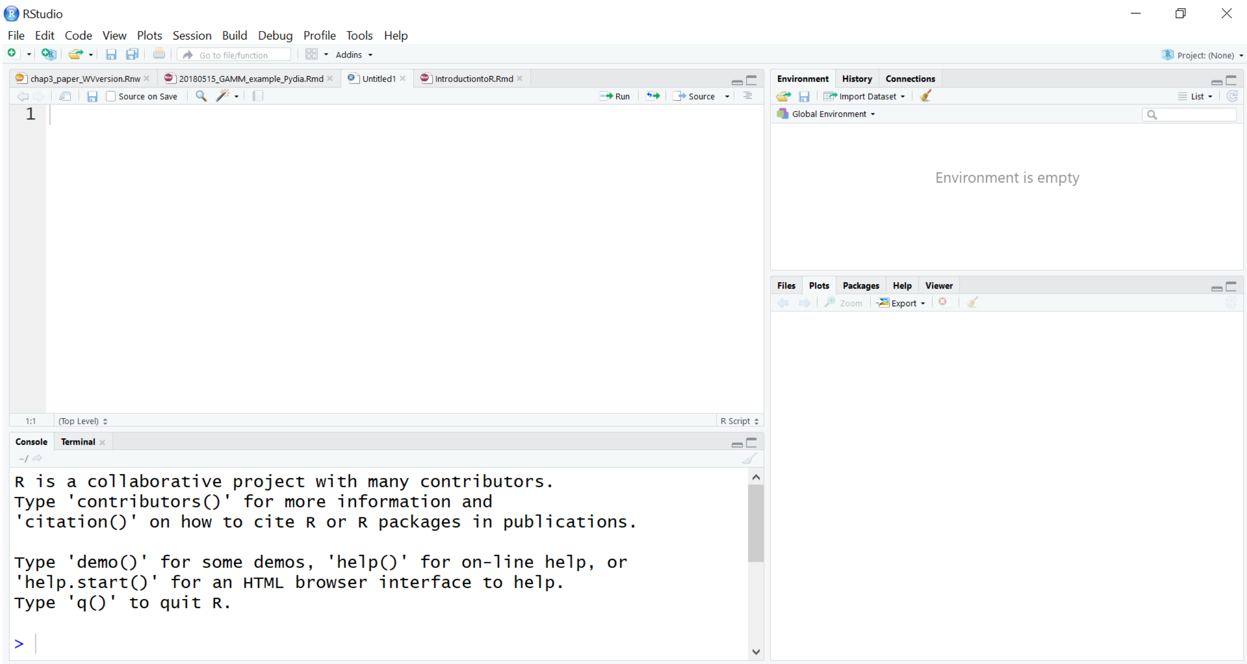


Figure 1: Four panels in R Studio

```
install.packages("devtools")
library(devtools)
install_github(repoAddress)
```

2.1.6 Navigation

In R Studio, you would 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 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.

2.1.7 Working folder structure

In R and in any other command line-based application, you would need to tell the app your current folder location and the location of the data. Usually we use the following folder structure:

- main project folder
- data: put your data here
- code: put your code here
- output: put your plots and tables here
- text: put your report here

But in practice, we usually work with code, 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. But when is the "final stage"? So do the file sorting several times.

2.1.7.1 Exercise

- Can you check your working folder/directory and what's inside it? (*hint* you can go to the menu item "session")

3 BASIC R

3.1 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
```

3.2 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*.

All objects will be loaded 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.

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

```
# 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
```

```
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?

3.3 A dataframe

A dataframe is a bit more complex, and here is a simple demonstration of its power.

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

```
##         City Rain_mm
## 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 can mix character columns (City) and numeric columns (Rain_mm). Here I used `c()` to generate vectors which I put in the columns. In addition, the colummns have names, which you can access using `colnames()`:

```r
colnames(Rainfall)
```

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

Once you have a dataframe, you can access parts of the dataframe or manipulate the dataframe.

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

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong 
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

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

```
##         City
## 1 Montevideo
## 2   New York
## 3  Amsterdam
## 4     Sydney
## 5     Moscow
## 6  Hong Kong
```

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

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong 
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

```r
# find a row
Rainfall[Rainfall["City"]=="Montevideo"]
```

```
## [1] "Montevideo" " 950"
```

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

```
##         City Rain_mm
## 1 Montevideo     950
## 2   New York    1174
```

```r
# find a subset
lots <- Rainfall[Rainfall["Rain_mm"] > 1000,]
lots
```

```
##        City Rain_mm
## 2  New York    1174
## 4    Sydney    1215
## 6 Hong Kong    2400
```

3.3.1 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?
- Which cities have rainfall below 1500 mm?

3.4 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 in code. 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".

```r
# set the working directory
setwd("Data")

# see some of the files
dir()[1:10]
```

```
##  [1] "Parana_CorientesSt.csv"       "semarang_chem.csv"           
##  [3] "UruguayRiver_ConcordiaSt.csv" NA                            
##  [5] NA                             NA                            
##  [7] NA                             NA
```

```
##  [9] NA                              NA
```

3.5 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 a few here.

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

```
##   Year Month Flow
## 1 1969     1 7888
## 2 1969     2 5951
## 3 1969     3 4296
## 4 1969     4 4173
## 5 1969     5 4539
## 6 1969     6 4857
```

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 of the most convenient package to use.

3.5.1 Exercise

- Can you read in the file: "Parana_CorrientesSt.csv"?

4 STATISTICAL ANALYSIS AND DATA MANIPULATION

Now it's time to look a bit further into more technical bits. How to manipulate data so we can perform some analyses on it to answer our research problem. There are, offcourse, base R commands to do the job, but find it easier for us to use `tidyverse` package. This package is actually a combo of several packages written by the same author.

4.1 Packages to use

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**.

Here we might want to use some of the features in the package tidyverse. The other package we will use later is the package zoo.

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

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)

4.1.1 Exercise

- Can you load (and maybe first install) the package zoo?

4.2 Statistical analysis

4.2.1 Summarising data

It is often important to summarise data, for example we might want to know the average monthly flow or the standard deviation of flow. R of course have several functions to deal with this.

4.2.2 Standard statistical functions

Here are some simple examples of standard statistical functions `mean`, `sd` and `cor` (and of course there are many more).

```r
# average monthly flow
mean(UR_flow$Flow)
```

```
## [1] 5456.553
```

```r
# st dev average flow
sd(UR_flow$Flow)
```

```
## [1] 3491.968
```

```r
# subset two years and correlate
flow1969 <- UR_flow[UR_flow$Year==1969,]
flow1970 <- UR_flow[UR_flow$Year==1970,]

cor(flow1969$Flow,flow1970$Flow)
```

```
## [1] -0.3164468
```

4.2.3 Using aggregate()

Another useful function is `aggregate()`, which allows you to apply a function over data frame and particular across different factors. Here is an example of summing the Uruguay river flow by year.

```r
# aggregate to annual flow
(annual_flow <- aggregate(UR_flow,list(Year=UR_flow$Year),sum))
```

```
##    Year  Year Month  Flow
## 1  1969 23628    78 53753
## 2  1970 23640    78 52130
## 3  1971 23652    78 66648
## 4  1972 23664    78 99562
```

```
## 5   1973 23676    78 103070
## 6   1974 23688    78  47130
## 7   1975 23700    78  68075
## 8   1976 23712    78  53500
## 9   1977 23724    78  73650
## 10  1978 23736    78  41700
## 11  1979 23748    78  61047
```

Note that the parentheses around the statement means that the result of the statement is printed.

4.2.3.1 Exercise

- Can you calculate the standard deviation of the monthly flow by year?

4.3 Data manipulation (using `tidyverse`)

Make sure you've done this.

install.packages("tidyverse")

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------- tidyverse 1.2.1 --

## v ggplot2 2.2.1     v purrr   0.2.4
## v tibble  1.3.4     v dplyr   0.7.4
## v tidyr   0.7.2     v stringr 1.2.0
## v readr   1.1.1     v forcats 0.2.0

## -- Conflicts ------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

4.4 Important commands

The following list is the important commands to remember:

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

Let's open this dataset. It's a water quality data in `csv` format. Note that we are now using the tidyverse version `read_csv` rather than `read.csv`.

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

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   ID = col_character(),
##   Area = col_character(),
##   Year = col_integer(),
##   UTM_east = col_integer(),
```

```
##   UTM_north = col_integer(),
##   UTM_zone = col_character(),
##   Depth = col_integer(),
##   TDS = col_integer(),
##   EC = col_integer(),
##   Aq = col_character(),
##   Fac = col_character()
## )
## See spec(...) for full column specifications.
```

```
chemdata
```

```
## # A tibble: 58 x 24
##        ID                   Area  Year      Lat     Long UTM_east
##     <chr>                  <chr> <int>    <dbl>    <dbl>    <int>
##  1 SB_185        PT. Ny.Meneer-1  1992 -6.95854 110.4562   439936
##  2 SB_273               PT. INAN  1992 -6.98295 110.4432   438500
##  3 SB_283       Obs. SD Kuningan  1992 -6.96437 110.4161   435500
##  4 SB_271      PT. Sango Keramik  1992 -6.98006 110.3133   424150
##  5 SB_270         Dolog Mangkang  1992 -6.97099 110.2934   421950
##  6 SB_278          Hotel Santika  1992 -6.99333 110.4292   436950
##  7 SB_325         PT Wahyu Utomo  1992 -6.99323 110.3477   427950
##  8 SB_190      PT. Gentong Gotri  1992 -6.98338 110.4287   436900
##  9 SB_256      Tambakharjo, Tugu  1992 -6.97832 110.3645   429800
## 10 SB_206 Tambak Udang, Mangkang  1992 -6.95020 110.3038   423100
## # ... with 48 more rows, and 18 more variables: UTM_north <int>,
## #   UTM_zone <chr>, Depth <int>, WL <dbl>, Elev <dbl>, TDS <int>,
## #   ph <dbl>, EC <int>, K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>,
## #   Cl <dbl>, HCO3 <dbl>, Bal <dbl>, Aq <chr>, Fac <chr>
```

4.4.1 select()

Suppose you want certain columns for your analysis. Use `select()`. In `tidyverse` package, we could use pipe operator `%>%` to give series of command. Instead of using many brackets.

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

```
## # A tibble: 58 x 2
##         Lat     Long
##       <dbl>    <dbl>
##  1 -6.95854 110.4562
##  2 -6.98295 110.4432
##  3 -6.96437 110.4161
##  4 -6.98006 110.3133
##  5 -6.97099 110.2934
##  6 -6.99333 110.4292
##  7 -6.99323 110.3477
##  8 -6.98338 110.4287
##  9 -6.97832 110.3645
## 10 -6.95020 110.3038
## # ... with 48 more rows
```

Note that this does not actually save your result into a new dataframe, in othere words, you cannot use the inro that is just printed unless you `assign` this to a new dataframe:

```
chemdata_LatLong <- chemdata %>%
  select(Lat, Long)
```

Or you want multiple columns `Lat`, `Long` until `Depth`. Again you can 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>    <int>     <int>    <chr> <int>
##  1 -6.95854 110.4562   439936   9230800      49M    96
##  2 -6.98295 110.4432   438500   9228100      49M    94
##  3 -6.96437 110.4161   435500   9230150      49M   150
##  4 -6.98006 110.3133   424150   9228400      49M    65
##  5 -6.97099 110.2934   421950   9229400      49M    NA
##  6 -6.99333 110.4292   436950   9226950      49M    86
##  7 -6.99323 110.3477   427950   9226950      49M    76
##  8 -6.98338 110.4287   436900   9228050      49M    NA
##  9 -6.97832 110.3645   429800   9228600      49M    NA
## 10 -6.95020 110.3038   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>    <int>     <int> <int>
##  1 -6.95854 110.4562   439936   9230800    96
##  2 -6.98295 110.4432   438500   9228100    94
##  3 -6.96437 110.4161   435500   9230150   150
##  4 -6.98006 110.3133   424150   9228400    65
##  5 -6.97099 110.2934   421950   9229400    NA
##  6 -6.99333 110.4292   436950   9226950    86
##  7 -6.99323 110.3477   427950   9226950    76
##  8 -6.98338 110.4287   436900   9228050    NA
##  9 -6.97832 110.3645   429800   9228600    NA
## 10 -6.95020 110.3038   423100   9231700    80
## # ... with 48 more rows
```

4.4.2 `filter()`

You want to select all data from Damar Formation. Use `filter()` function.

```
chemdata %>%
  filter(Aq == "Damar")
```

```
## # A tibble: 14 x 24
##        ID                         Area  Year      Lat     Long UTM_east
##     <chr>                        <chr> <int>    <dbl>    <dbl>    <int>
##  1 SB_271           PT. Sango Keramik  1992 -6.98006 110.3133   424150
##  2 SB_270              Dolog Mangkang  1992 -6.97099 110.2934   421950
```

```
##   3 SB_325             PT Wahyu Utomo  1992 -6.99323 110.3477   427950
##   4 SB_225            PDAM Manyaran  1992 -7.00096 110.3830   431850
##   5  SB_92                RS Kariadi  1992 -6.99104 110.4061   434400
##   6 SB_112            Hotel Siranda  1993 -6.99816 110.4184   435763
##   7 SB_215  S. Pantau PT. Kimia Farma  2003 -7.00257 110.3917   432811
##   8  SB_33       Es Prawito Jaya Baru  2003 -6.98830 110.3600   429310
##   9 SB_590 Bukit Perak, Jl. Raya Tugu  2003 -6.98598 110.3396   427050
##  10 SP_341            Obs. Indofood  2003 -6.99024 110.3343   426474
##  11    N_1               Sendangguwo  2006 -7.01219 110.4553   439836
##  12    N_3                   Tandang  2006 -7.01503 110.4445   438643
##  13    N_7                   Ngalian  2006 -7.00433 110.3393   427020
##  14 SB_217      Obs. Standart Battery  2007 -6.98360 110.3373   426797
## # ... with 18 more variables: UTM_north <int>, UTM_zone <chr>,
## #   Depth <int>, WL <dbl>, Elev <dbl>, TDS <int>, ph <dbl>, EC <int>,
## #   K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>, Cl <dbl>,
## #   HCO3 <dbl>, Bal <dbl>, Aq <chr>, Fac <chr>
```

4.4.3 arrange()

Sorting out data by Aq and Fac. Use `arrange()` function.

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

```
## # A tibble: 58 x 24
##        ID                       Area  Year      Lat     Long UTM_east
##     <chr>                      <chr> <int>    <dbl>    <dbl>    <int>
##   1 SB_271          PT. Sango Keramik  1992 -6.98006 110.3133   424150
##   2 SB_270            Dolog Mangkang  1992 -6.97099 110.2934   421950
##   3 SB_325             PT Wahyu Utomo  1992 -6.99323 110.3477   427950
##   4 SB_225            PDAM Manyaran  1992 -7.00096 110.3830   431850
##   5  SB_92                RS Kariadi  1992 -6.99104 110.4061   434400
##   6 SB_112            Hotel Siranda  1993 -6.99816 110.4184   435763
##   7 SB_215  S. Pantau PT. Kimia Farma  2003 -7.00257 110.3917   432811
##   8  SB_33       Es Prawito Jaya Baru  2003 -6.98830 110.3600   429310
##   9 SB_590 Bukit Perak, Jl. Raya Tugu  2003 -6.98598 110.3396   427050
##  10 SP_341            Obs. Indofood  2003 -6.99024 110.3343   426474
## # ... with 48 more rows, and 18 more variables: UTM_north <int>,
## #   UTM_zone <chr>, Depth <int>, WL <dbl>, Elev <dbl>, TDS <int>,
## #   ph <dbl>, EC <int>, K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>,
## #   Cl <dbl>, HCO3 <dbl>, Bal <dbl>, Aq <chr>, Fac <chr>
```

4.4.4 mutate()

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

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

```
## # A tibble: 58 x 25
##        ID                Area  Year      Lat     Long UTM_east
##     <chr>               <chr> <int>    <dbl>    <dbl>    <int>
##   1 SB_185        PT. Ny.Meneer-1  1992 -6.95854 110.4562   439936
##   2 SB_273               PT. INAN  1992 -6.98295 110.4432   438500
```

```
##   3 SB_283       Obs. SD Kuningan  1992 -6.96437 110.4161   435500
##   4 SB_271      PT. Sango Keramik  1992 -6.98006 110.3133   424150
##   5 SB_270         Dolog Mangkang  1992 -6.97099 110.2934   421950
##   6 SB_278          Hotel Santika  1992 -6.99333 110.4292   436950
##   7 SB_325         PT Wahyu Utomo  1992 -6.99323 110.3477   427950
##   8 SB_190      PT. Gentong Gotri  1992 -6.98338 110.4287   436900
##   9 SB_256      Tambakharjo, Tugu  1992 -6.97832 110.3645   429800
## 10 SB_206 Tambak Udang, Mangkang  1992 -6.95020 110.3038   423100
## # ... with 48 more rows, and 19 more variables: UTM_north <int>,
## #   UTM_zone <chr>, Depth <int>, WL <dbl>, Elev <dbl>, TDS <int>,
## #   ph <dbl>, EC <int>, K <dbl>, Ca <dbl>, Mg <dbl>, Na <dbl>, SO4 <dbl>,
## #   Cl <dbl>, HCO3 <dbl>, Bal <dbl>, Aq <chr>, Fac <chr>, ratio_Cana <dbl>
```

4.4.5 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 1040.517 15752.8   11.2    58
```

4.4.6 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  371.0714    70.0   11.2    14
## 2            Garang  445.3636   145.5   19.6    11
## 3 Quaternary marine 1522.9091 15752.8   25.0    33
```

4.4.6.1 Exercise

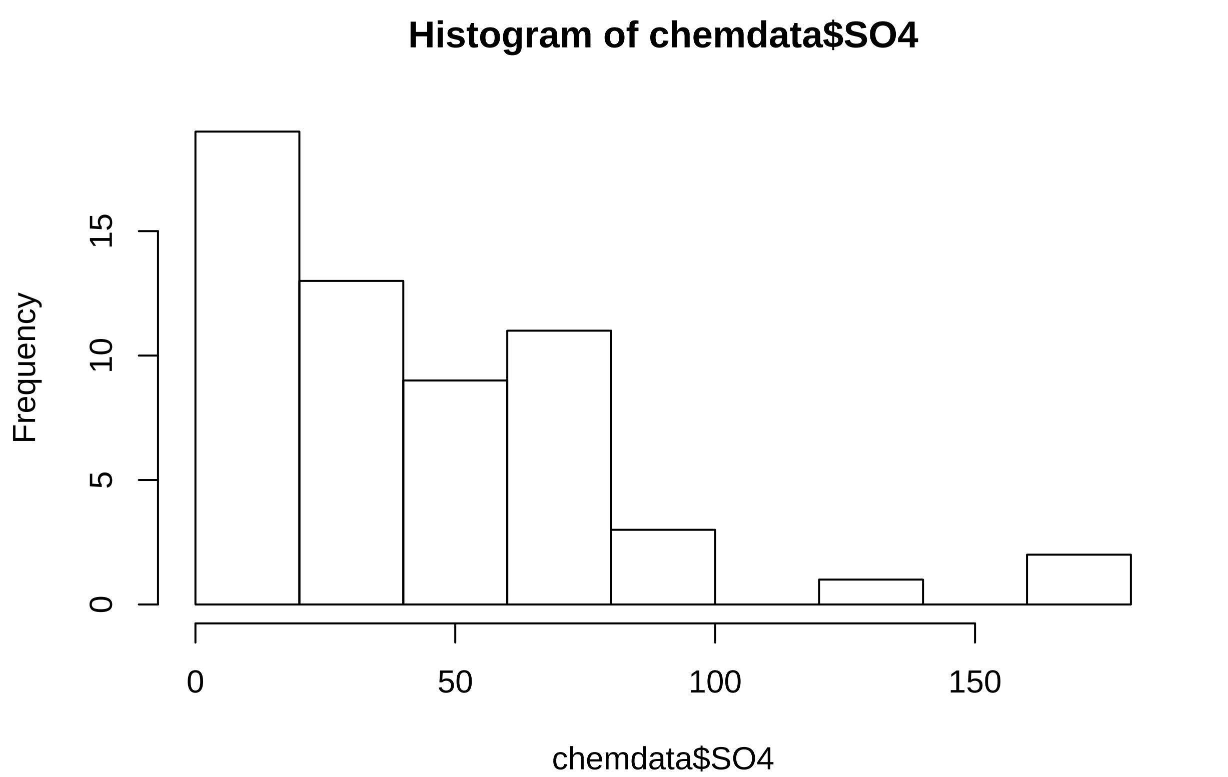
- Can you calculate the mean(Cl) and sd(Na) for the dataset grouped by Fac?

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.

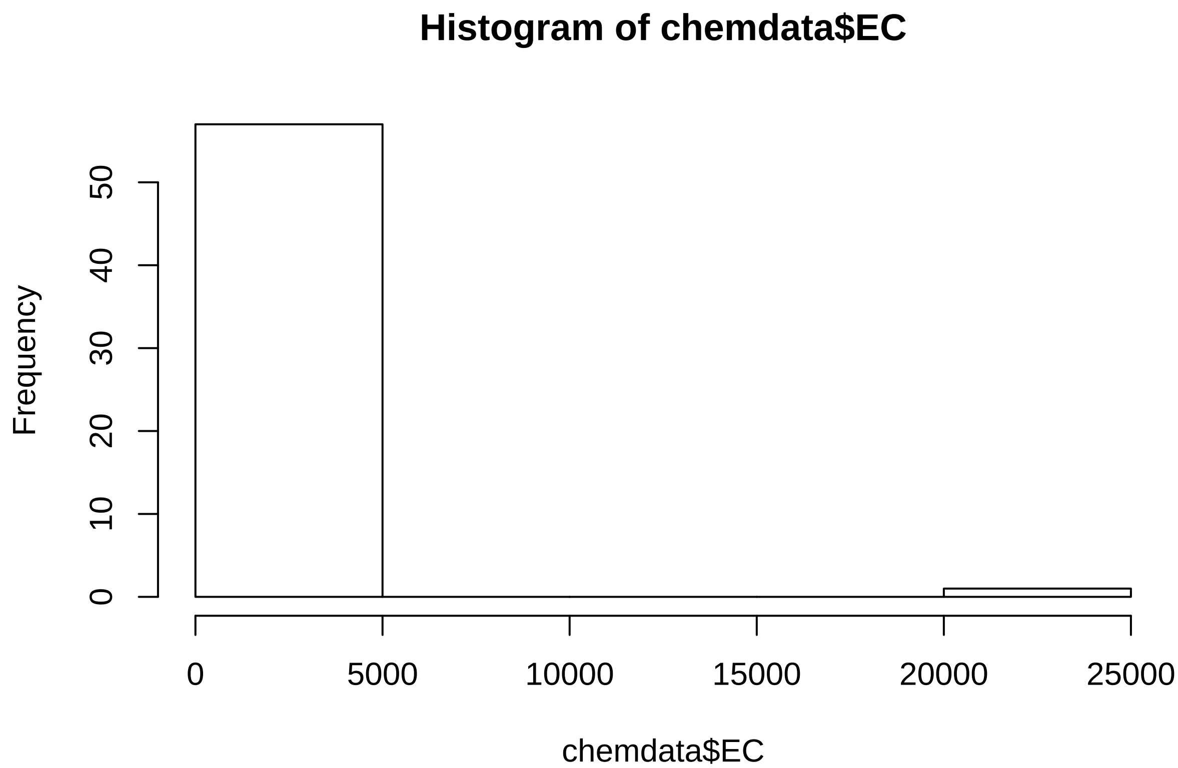
5 PLOTTING

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.

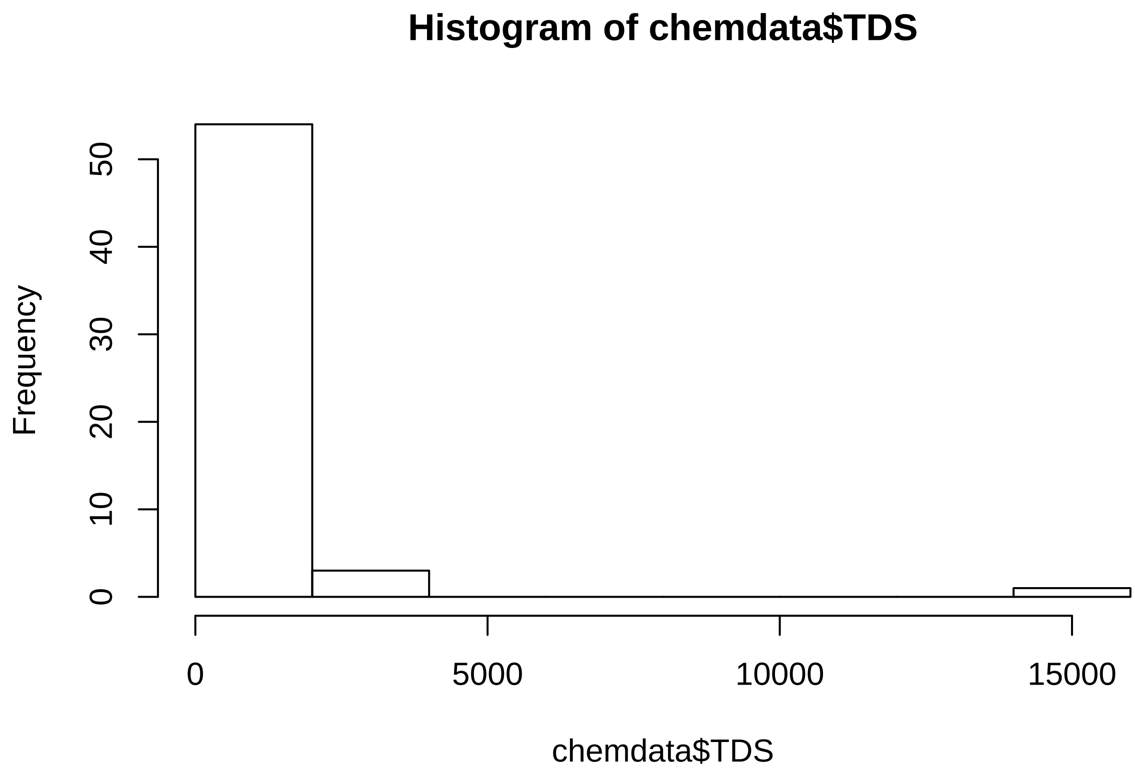
```
hist(chemdata$SO4)
```



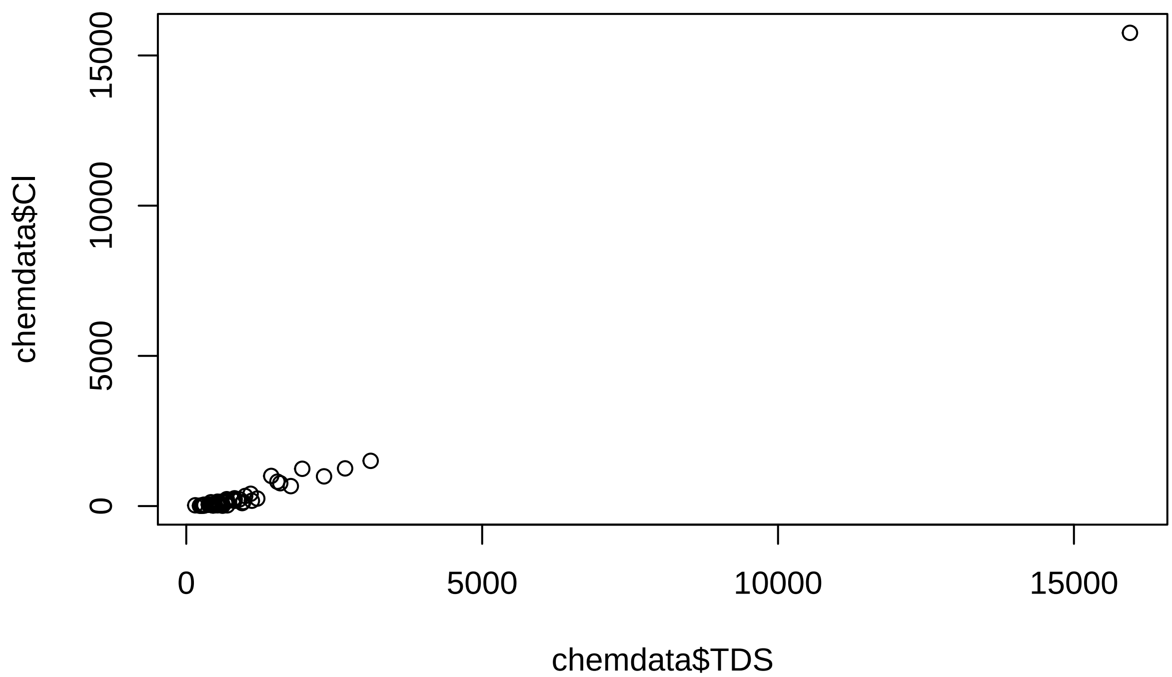

```
hist(chemdata$EC)
```




```
hist(chemdata$TDS)
```

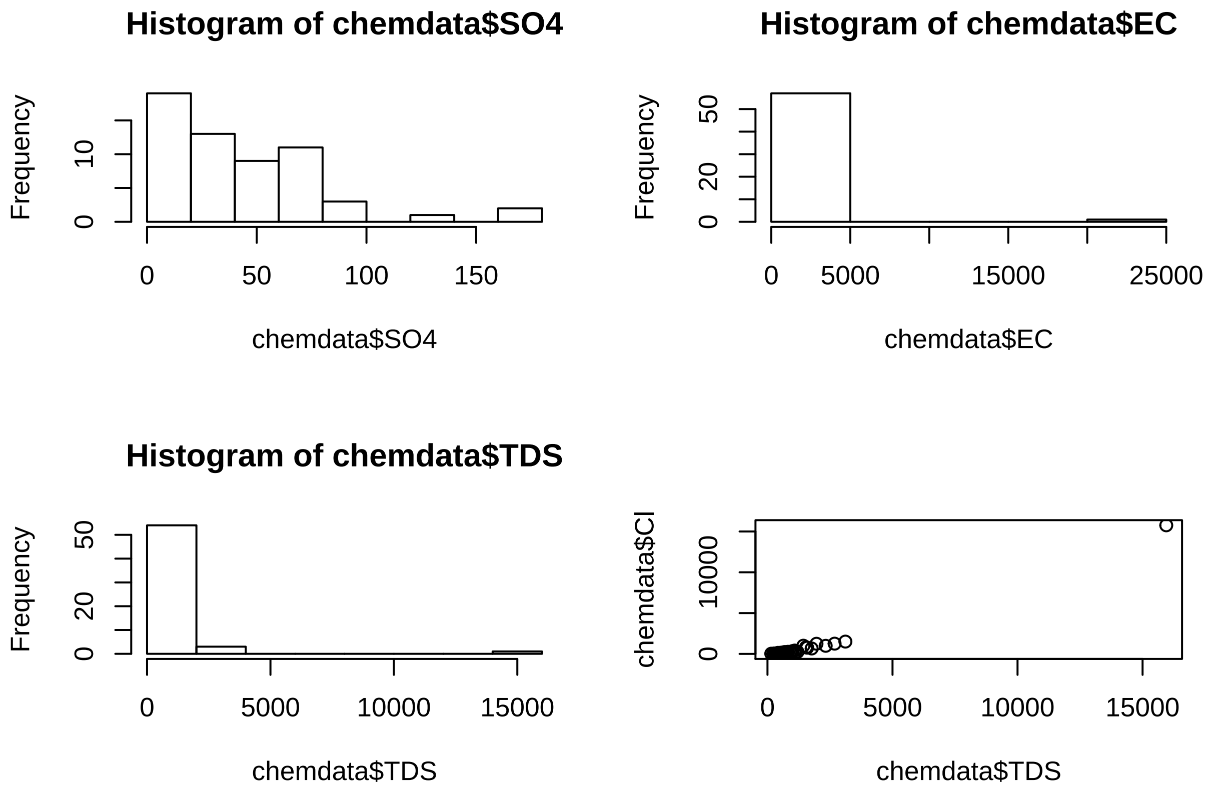



```
plot(chemdata$TDS, chemdata$Cl)
```



Maybe you want to look at them in one panel.

```
par(mfrow=c(2,2))
hist(chemdata$SO4)
hist(chemdata$EC)
hist(chemdata$TDS)
plot(chemdata$TDS, chemdata$Cl)
```




```r
par(mfrow=c(1,1))
```

You could always tweak the plot to suits your needs. There are many resources about plotting in R, like:

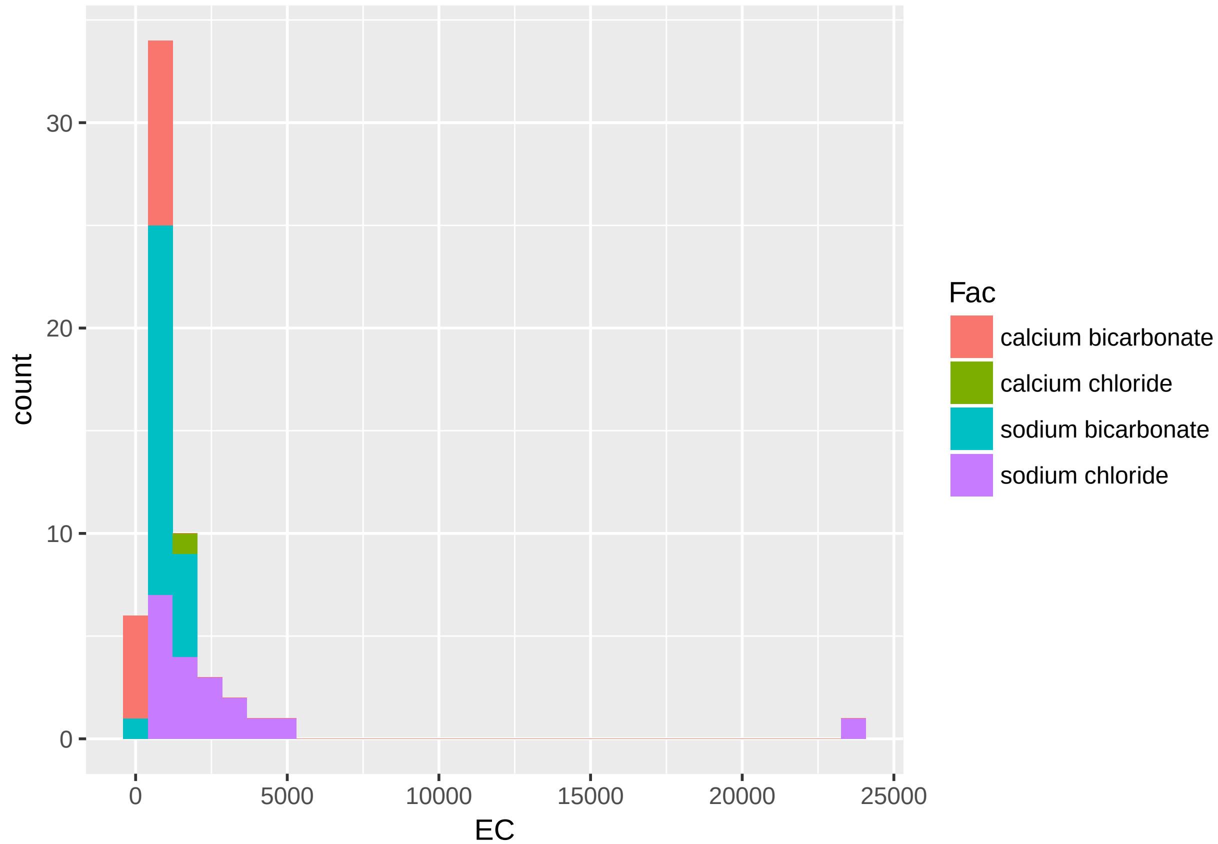
- Producing Simple Graphs with R,
- Quick R.
- and more.

Or you could you `ggplot2` plotting engine from `tidyverse`.

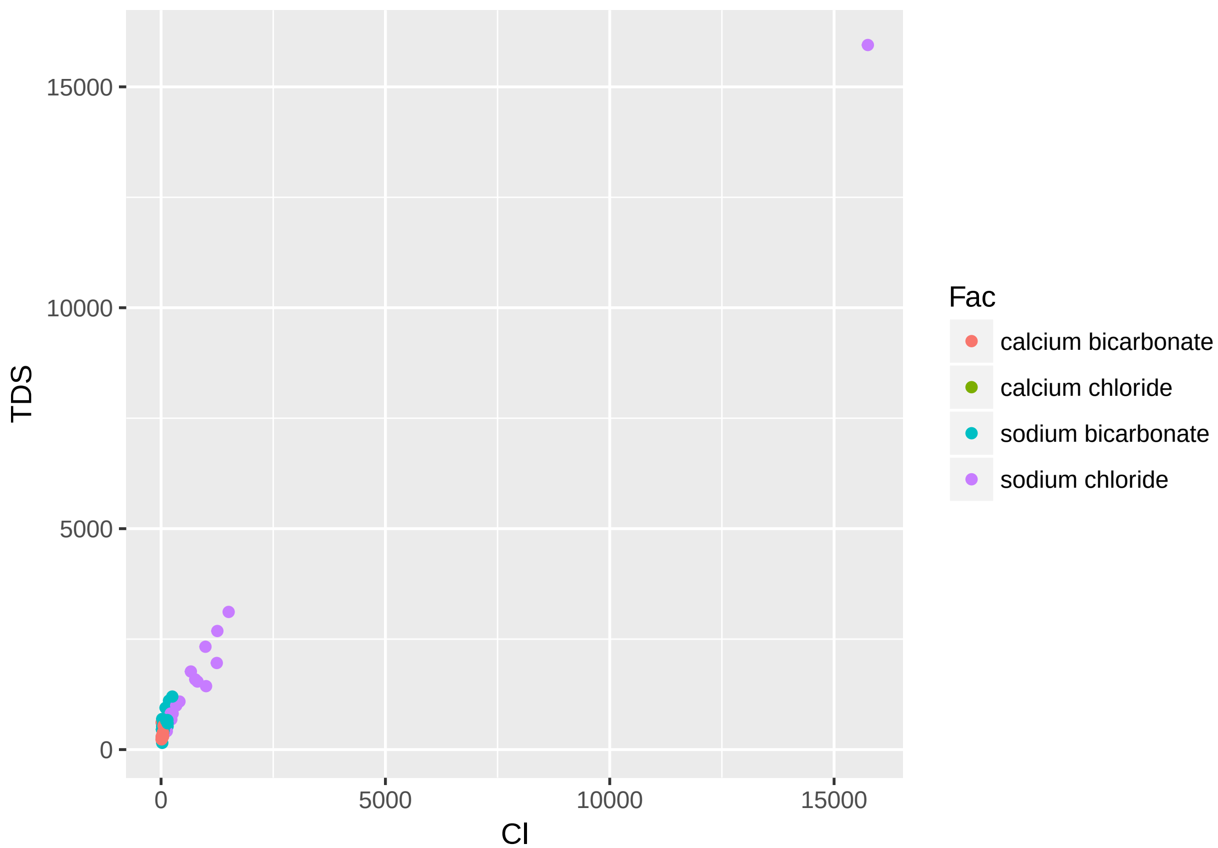
library(ggplot2)

```r
pl <- ggplot(chemdata, aes(EC, fill = Fac)) +
  geom_histogram()
pl
```

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




```
pl2 <- ggplot(chemdata, aes(Cl, TDS, colour = Fac)) +
  geom_point()
pl2
```



5.1 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 base R and ggplot2.
- can you tweak it by adding title to the plot and title to all axis.

6 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
```

```r
# 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.

6.0.1 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.

```r
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)
```

```
##     Year Month  Flow    V4
## 123 1979     3  1699 small
## 124 1979     4  1650 small
## 125 1979     5  5052 large
## 126 1979     6  2619 small
## 127 1979     7  3776 small
## 128 1979     8  6134 large
## 129 1979     9  3275 small
## 130 1979    10 15692 large
## 131 1979    11 12244 large
## 132 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:

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

```
##     Year Month  Flow    V4           V5
## 123 1979     3  1699 small        small
## 124 1979     4  1650 small        small
## 125 1979     5  5052 large intermediate
## 126 1979     6  2619 small intermediate
## 127 1979     7  3776 small intermediate
```

```
## 128 1979     8  6134 large intermediate
## 129 1979     9  3275 small intermediate
## 130 1979    10 15692 large        large
## 131 1979    11 12244 large        large
## 132 1979    12  7380 large intermediate
```

You can try out some of your own versions of this

6.1 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

```r
# 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.

```r
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)"
```

```r
# 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"
```

6.1.1 Exercise

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

6.1.2 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)

6.1.3 Exercise

Write a small program that uses comparison operators and a logical operator. # Plotting using ggplot and using zoo

R has many different plotting options, but recently ggplot2 appears to be the preferred option. One of the reasons for the support for ggplot2 is because it can also be used in Python.

As a start I will first modify the UR_flow data to make this into a "zoo" data frame, which has a timeseries based index and helps with plotting.

The function basically takes the data and links it to an index (such as the date). I first construct a vector of dates using the month and year from `UR_flow`.

```
#install.packages("zoo")  do this if you haven't done so.
require(zoo)
```

```
## Loading required package: zoo
```

```
## Warning: package 'zoo' was built under R version 3.4.4
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
UR_flow$Dates <- as.Date(paste(UR_flow$Year,UR_flow$Month,"01",sep="-"))
UR_flow_z <- zoo(UR_flow$Flow, order.by = UR_flow$Dates)
```

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

```
plot(UR_flow_z)
```

Using ggplot2 is a bit more involved, and does not work well directly with zoo. So we need to go back to UR_flow.

```
p <- ggplot(UR_flow, aes(x=Dates, y = Flow)) + geom_line()
p
```

7 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:

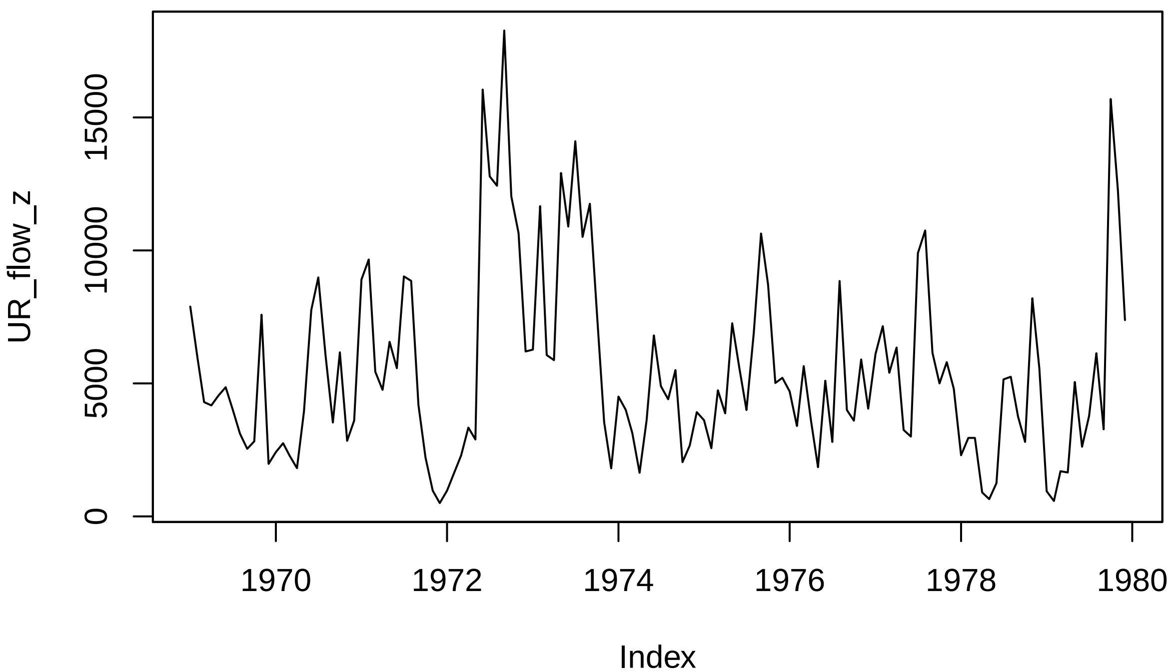


Figure 2: Demonstration of the simple plotting package in R

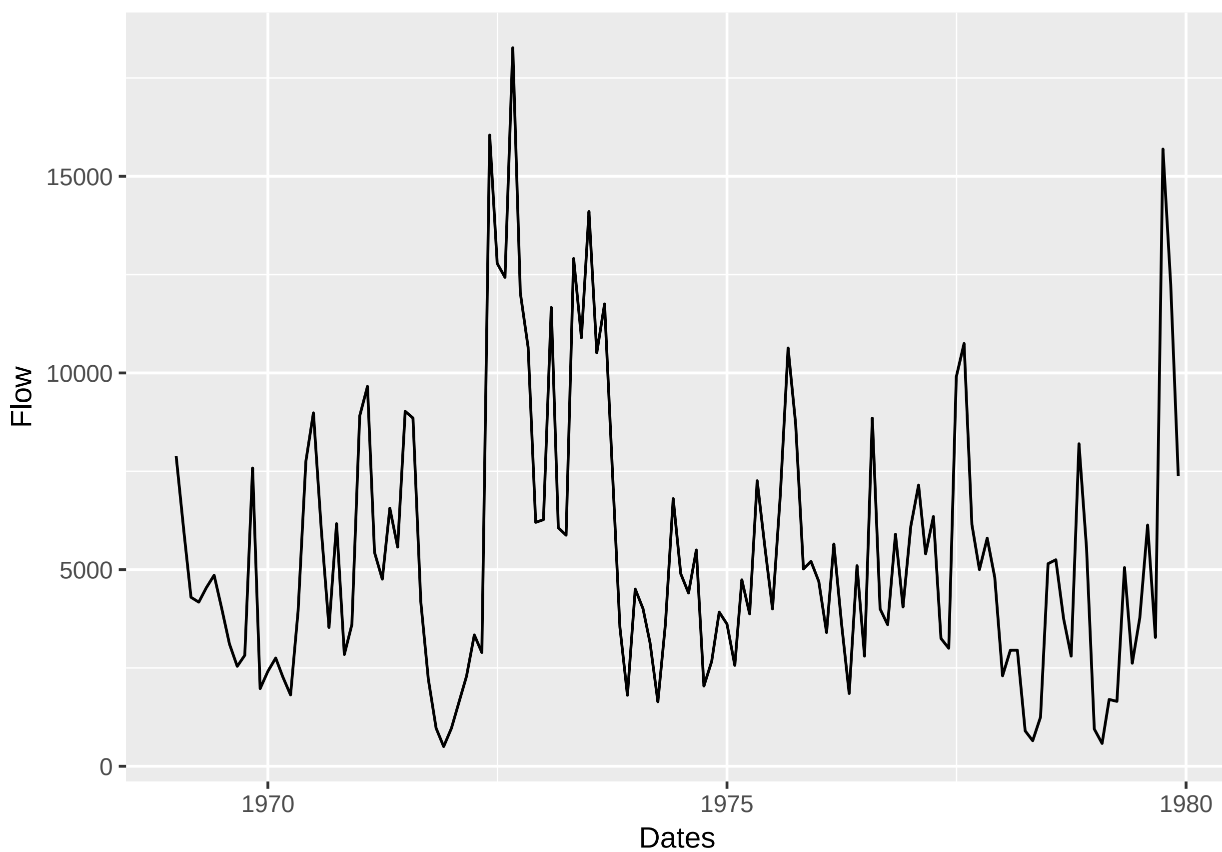


Figure 3: Demonstration of the ggplot2 package in R

```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.

7.0.1 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