# Data Science and Statistics in Research: unlocking the power of your data

Session 1.3: Using R with your data

## Introduction

In this practical we will introduce the `R` statistical software and demonstrate its functionality.

## Getting Help with `R`

CRAN (the repository for `R` softwware and packages), requires all package authors to create, manuals with examples and help pages. The `help()` function and `?` operator in `R` provide access to help pages for all `R` functions and data sets in installed packages. Lets bring up the help pages for the sum function.

```
?sum
help(sum)
```

A help page will appear in the lower right pane in RStudio and will contain the following information (there maybe more):

- Description: purpose of the function
- Usage: an example of a typical implementation
- Arguments: a list of the arguments you can supply to the function and what each does
- Details: more detailed information about the function and its arguments
- Value: information about the likely output of a function (e.g. does the function return an integer, or a list, or a matrix, or something different)
- See Also: a list of useful related functions
- References: citations which can often be very useful
- Examples: example code

## Packages

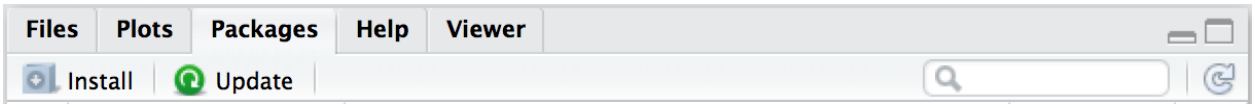
Packages are collections of `R` functions and data. The area where packages are stored is called the library. `R` comes with a standard set of packages, for example `base` (the `R` base package). Packages are often written externally as base `R` does not include the exact functionalilty required to do analyses.

To help others who wish to do similar analyses, developers often make these packages publically available on CRAN (www.r-cran.org), a repository for `R` software. These packages will need to be downloaded and installed.

The `install.packages()` function will download and install the packages that we need. For example, the `foreign` package allows the user to read in data from other statistical analysis software such as Stata or SAS, which we can install.

```
install.packages("foreign")
```

Alternatively, we can install packages clicking the 'Install' button in the 'Packages' pane and searching for the required package.

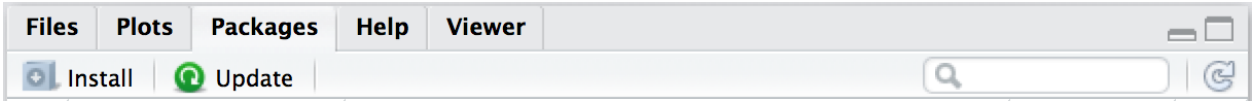



Once packages are downloaded and installed, we need to load them before using functions and/or data inside. We do this by using the `require()` function.

```
require("foreign")
```

`R` packages need to be loaded everytime a session in `R` is opened.

Packages are frequently updated. To update your packages, we use the `update.packages()`. Alternatively, we can update packages by clicking the 'Update' button in the Packages pane.




# The Basics of `R`

When working in `R`, you write/edit code in the editor (upper left window). To run your code, you hit the 'Run' button, on a line of code and the output is displayed in the console (lower left window). If your code runs properly, the results will display in black. If there are errors in your code, warnings and errors will display in red.

## Comments

`R` makes use of the `#` sign to add comments. Comments are useful so that you and others can understand what the `R` code does. It is good practice to always comment your code. Whenever `R` encounters the `#` operator, it will ignore everything printed after it in the current line so they don't influence your results.

```
# Calculating 3+4
3 + 4
[1] 7
```

As you can see the line beginning with `#` has produced no results, and the other line has. In its most basic form, `R` can be used as a simple calculator, using some of the following operators.

## Functions

R comes with many functions that are installed as part of the base package. All functions have names and take arguments in parentheses: `function(...)`. Here are some of the basic functions R knows

- `print()` – prints objects
- `sum()` – computes the sum of all elements entered
- `table()` – computes frequency tables
- `summary()` – computes a summary of inputted data
- `length()` – tells you the number of entries in a vector
- `round()` – rounds the input to the nearest decimal place.

```
# Printing 'Hello!'
print('Hello!')
[1] "Hello!"

# Sum 1,2 and 3
sum(1,2,3)
[1] 6

# Absolute value of 2 and -2
abs(2); abs(-2)
[1] 2
[1] 2
```

## Variable assignment

R is an object oriented program, and objects allow you to store data. There are two assignment operators in R: `<-` and `=`. Assigments allow you to repeatedly call and use data in your calculations.

```
# Assign the value 42 to x
x <- 42

# Print out the value of the object x
x
[1] 42

# Add 2 to x
x + 2
[1] 44

# Add 5 to x
x + 5
[1] 47
```

If a calculation is made, the results are printed but not stored, unless assigned to another object. So if we are interested in storing the results from our calculations to use in further ones later, they should be stored in other objects.

```
# Add 5 to x and reassigning to x
y <- x + 5
```

```
# Print out the value of the object x
y
[1] 47
```

When storing results of your calculations, beware that you do not overwrite any other object you wish to keep.

## Vectors

Vectors are one-dimension sequences of numbers, text, or logical data. A vector is a simple tool to store data. You can create a vector using the function `c()`. This function takes a number of numbers, text or logicals, separated by commas, and concatenates them into a vector.

```
# Storing 5 values in a vector and assigning to 'a'
a <- c(1, 2, 3, 4, 5)

# Storing 5 values in a vector and assigning to 'b'
b <- c(-2, -4, 6, 7, 8)
```

More information on the `c()` function can be seen on the help pages by typing `?c` into `R`

Elements from a vector can be extracted using '[]' with the corresponding element(s) to extract.

```
# Extracting the first element of a
a[1]
[1] 1

# Extracting the first and third element of a
a[c(1,3)]
[1] 1 3
```

Vectors can be used in arithmetic expressions, and the operations are performed element-wise. This means that if we want to add two vectors together, `R` will take the first element of the vectors and add them, then take the second elements of the vectors and add them and so on.

```
# Lets add a and b
a + b
[1] -1 -2  9 11 13
```

Similarly, if we want to multiply by two and add five to all elements of a vector, `R` will do this element-wise.

```
# Lets add a and b
2 * a + 5
[1]  7  9 11 13 15
```

## Dataframes

Dataframes are very flexible objects for data analysis in `R`. Dataframes are two-dimensional, with an observation taking up a row, and a variable taking up a column, similar to a database or a spreadsheet. It can also be thought of as a collection of vectors concentrated into one object.

We can create dataframes using the `data.frame()` function. This function will take a list of vectors and concatenate them together into one object. You can also give specific variable names. More infomation and examples of the `data.frame()` function can be see by typing `data.frame` into `R`.

```
# Storing 5 values in a vector and assigning to 'a'
a <- c(1, 2, 3, 4, 5)

# Storing 5 values in a vector and assigning to 'b'
b <- c(-2, -4, 6, 7, 6)

# Creating a dataframe fr
mydata <- data.frame(a,b)

# Printing mydata
mydata
  a  b
1 1 -2
2 2 -4
3 3  6
4 4  7
5 5  6
```

`R` also has an extensive repository of example dataframes which can be used. To call datasets to the workspace, we use the `data()` function. If entered without arguments, it will bring up a list of all datasets that currently stored within `R`.

```
# List all the datasets stored in R
data()
```

**Activity**

- Create a vector called `c` which contains the elements 2, 3, 1, 6 and -1 and create a dataframe that contains variables `a`, `b` and `c`.

**Getting to know the structure of dataframes**

Once a dataframe has been loaded into `R`, you should understand the data stored within. Initally, we can understand our dataset by finding the number of observations and variables in dataframes by using the `nrow()` and `ncol()` functions respectively.

```
# Viewing the structure of the drugs dataset
nrow(mydata)
[1] 5

# Viewing the first 5 rows of the drugs dataset
ncol(mydata)
[1] 2
```

A quick way of viewing the dataset to see the data are using the `names()`, `str()` and `head()` functions. The `names()` function will display the variable names within a dataframe. The `str()` function will display the structure of the dataset and the `head()` function will display the first 6 rows in the dataframe.

```
# Display the variable names
names(mydata)
[1] "a" "b"

# Viewing the structure of the drugs dataset
str(mydata)
'data.frame':   5 obs. of  2 variables:
 $ a: num  1 2 3 4 5
 $ b: num  -2 -4 6 7 6

# Viewing the first 5 rows of the drugs dataset
head(mydata)
  a  b
1 1 -2
2 2 -4
3 3  6
4 4  7
5 5  6
```

**Extracting and creating variables**

Data within dataframes can be extracted using '[]'. As dataframes are two-dimensional objects, we need to specify two things, the row and/or the column, separated with a common for example [,]. Lets extract (i) the variable `a` from `mydata`, (ii) the first row from `a` from `mydata` and (iii) 3rd row for variable `b` from `mydata`.

```
# Extracting the variable a from mydata
mydata[,'a']
[1] 1 2 3 4 5

# Extracting the row (or observation) from mydata
mydata[1,]
  a  b
1 1 -2

# Extracting the 3rd row for variable b from mydata
mydata[3,'b']
[1] 6
```

Note that if we do not require a specific row or column entry, we leave it blank.

Alternatively, you can extract variables from dataframes we using the `$` operator. We first specify the dataset then reference the variable required. Lets extract the variable `a` from our new dataframe `mydata`.

```
# Extracting the variable a from mydata
mydata$a
[1] 1 2 3 4 5
```

Creating a new variable within a dataframe is usually is a simple task in `R`. Let's create a variable `aplusb` within `mydata` which adds the variables `a` and `b`. For this we extract the variables `a` and `b` from `mydata`, add them together and assign the results to `aplusb` in `mydata`.

```
# Creating variable c by adding a and b together
mydata$aplusb <- mydata$a + mydata$b

# Printing the variable aplusb
mydata$aplusb
[1] -1 -2  9 11 11
```

**Activity**

- Create a new variable called `atimesb` in `mydata` which mulitplies the variables `a` and `b` together.

**Subsetting dataframes**

We can use subsetting to retrieve parts of a dataframe which are of interest for a specific purpose. Logical operators are crucial for subsetting data. When `R` evaluates statements containing logical operators it will return either `TRUE` or `FALSE`.

Here are some of the logical operators in `R`

- `<` - less than
- `<=` - less than or equal to
- `>` - greater than
- `>=` - greater than or equal to
- `==` - equal
- `!=` - not equal
- `&` = and
- `|` - or

```
# Checking 1 == 2
1 == 2
[1] FALSE

# Checking 1 == 1
1 == 1
[1] TRUE
```

Suppose we are interested in extracting the rows of the dataframe `mydata` where the variable `a` is less than 4. To subset data, we use the `subset()` function.

```
# Subsetting mydata where `a` is less than 4
subset(mydata, a < 4)
  a  b aplusb
1 1 -2     -1
2 2 -4     -2
3 3  6      9
```

**Activity**

- Use subsetting to extract the rows of `mydata` where the variable `b` is equal to 6.

## Calculating risks using dataframes

We now demonstrate how to use dataframes in a real analysis. We will be calculating the annual number of deaths attributable to $PM_{2.5}$ air pollution. You do not need to understand the technical detail of the example but use this to understand the functionality of dataframes

### Preliminaries

We wish to estimate the annual number of deaths attributable to $PM_{2.5}$ air pollution. In order to do this, we need (i) a relative risk (RR), (ii) the population at risk for the areas of interest, (iii) the overall mortality rate (OMR), and (iv) a baseline value for air pollution (for which there is no associated increase in risk).

In this example, we have a RR of 1.06 per $10\mu gm^{-3}$, the population at risk is 1 million and the OMR is 80 per 10000. We first enter this information into R by assigning the values to different variables.

```
# Relative Risk
RR <- 1.06

# Size of population
Population <- 1000000

# Unit for the Relative Risk
RR_unit <- 10

# Overall mortalilty count, used for calulating the overall mortality rate
OMR_count <- 80

# Denonmnator (population at risk), used for calulating the overall mortality rate.
OMR_pop <- 10000

# Mortality rate
OMR = OMR_count/OMR_pop
OMR
[1] 0.008

# Baseline value of PM2.5 for which there is no increased risk
baseline <- 5

# Population attributable fraction
#PAF = (Proportion of population exposed*(RR-1))/(Proportion of population exposed*(RR-1)+1).
#In this case the proportion of the population exposed is one.

PAF = (RR-1)/RR
PAF
[1] 0.05660377
```

In this example, we will calculate the attributable deaths for increments of 10, however the following code is general and will work for any increments.

```
# PM2.5 categories
PM2.5.cats <- c(5,15,25,35,45,55,65,75,85,95,105)

# Create a dataframe containing the PM2.5 categoriess
Impacts <- data.frame(PM2.5.cats)
```

### Calculating Risks

We now calculate the increases in risk for each category of $PM_{2.5}$. For each category, we find the increase in risk compared to the baseline.

For the second category, with $PM_{2.5} = 15$, the risk will be 1.06 (the original RR) as this is $10\mu gm^{-3}$ (one unit) greater than the baseline.

For the next category, $PM_{2.5}$ is $10\mu gm^{-3}$ higher than the previous category (one unit in terms of the RR) and so the risk in that category again be increased by a factor of 1.06 (on that of the previous category). In this case, the relative risk (with respect to baseline) is therefore `1.06 * 1.06 = 1.1236`.

For the next category, $PM_{2.5} = 25$ which is again $10\mu gm^{-3}$ (one unit in terms of the RR) higher, and so the relative risk is 1.06 multiplies by the previous value, i.e. `1.06 * 1.1236 =  1.191016`.

We can calculate the relative risks for each category (relative to baseline) in `R`. For each category, we find the number of units from baseline and repeatedly multiple the RR by this number. This is equivalent to raising the RR to the power of (Category-Baseline)/Units, e.g. RR^((Category-Baseline)/Units)).

We add another column to the Impacts dataframe containing these values.

```
# Calculating Relative Risks
Impacts$RR <- RR^((Impacts$PM2.5.cats - baseline)/RR_unit)
```

Once we have the RR for each pollution level, we can calculate the rate for each category. This is found by applying the risks to the overall rate. Again, we add these numbers to the Impacts dataframe as an additional column.

```
# Calculating the rates in each category
Impacts$Rate <- Impacts$RR * OMR

# Add the PAFs for each category
Impacts$PAF <- Impacts$RR * (Impacts$RR-1)/Impacts$RR

# Add the nuimber of (expected) deaths  per year for each category
Impacts$Deaths.Per.Year <- Impacts$Rate * Population
```

For each category, we need to calculate the extra deaths (with reference to the overall rate). The number of deaths for the reference category is the first number in the `Deaths.Per.Year` column.

```
# The number of deaths
Impacts$Deaths.Per.Year[1]
[1] 8000

# We can then calculate the excess numbers of deaths for each category
Impacts$Extra.Deaths.Per.Year <- Impacts$Deaths.Per.Year - Impacts$Deaths.Per.Year[1]
```

For each category, we then want to calculate the number of deaths gained. These are the difference between the values in each category. We can find these using the `diff()` function. This will produce a set of differences for which the length is one less than the number of rows in our Impacts dataframe. We need to add a zero to this to ensure that they line up when we add them as another column.

```
# Calculate the number of deaths gained
diff(Impacts$Extra.Deaths.Per.Year)
 [1] 480.0000 508.8000 539.3280 571.6877 605.9889 642.3483 680.8892
```

```
 [8] 721.7425 765.0471 810.9499

# We can now add these gains to the main Impacts dataframe
Impacts$Gain <- c(0,diff(Impacts$Extra.Deaths.Per.Year))

# Show the results
Impacts
   PM2.5.cats       RR        Rate       PAF Deaths.Per.Year
1           5 1.000000 0.008000000 0.0000000        8000.000
2          15 1.060000 0.008480000 0.0600000        8480.000
3          25 1.123600 0.008988800 0.1236000        8988.800
4          35 1.191016 0.009528128 0.1910160        9528.128
5          45 1.262477 0.010099816 0.2624770       10099.816
6          55 1.338226 0.010705805 0.3382256       10705.805
7          65 1.418519 0.011348153 0.4185191       11348.153
8          75 1.503630 0.012029042 0.5036303       12029.042
9          85 1.593848 0.012750785 0.5938481       12750.785
10         95 1.689479 0.013515832 0.6894790       13515.832
11        105 1.790848 0.014326782 0.7908477       14326.782
   Extra.Deaths.Per.Year     Gain
1                  0.000   0.0000
2                480.000 480.0000
3                988.800 508.8000
4               1528.128 539.3280
5               2099.816 571.6877
6               2705.805 605.9889
7               3348.153 642.3483
8               4029.042 680.8892
9               4750.785 721.7425
10              5515.832 765.0471
11              6326.782 810.9499
```

## Reading in your own data

In your analyses, the data you want or need may not be stored in R. Frequently, you will have created or downloaded data in some other program (e.g. Excel, or Stata, etc). This means to analyse it in R we need to read it in.

A common way in R is to import data is from files in 'comma separated values' (.csv) format. A CSV is a simple file format used to store tabular data, such as a spreadsheet. Files in CSV format can be edited from programs, such as Microsoft Excel. To read a CSV file into R we use the `read.csv()` function.

```
# Calculate the number of deaths gained
mydata <- read.csv(file='<filename>.csv')
```

Let's read example data a CSV format into R. The dataset called `mtcars.csv` contains fuel consumption and 10 other attributes about design for 32 cars.

```
# Calculate the number of deaths gained
mydata <- read.csv(file='mtcars.csv')

# Lets view the first few rows
```

```
head(mydata)
             carname  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1          Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2      Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3         Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
4     Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
5  Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
6            Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

This dataset appear to have been read into R correctly. There are many more formats of data that can read into R. There is a package called **foreign** that includes functions to read datasets from other statistical software such as Stata, SPSS and SAS.

## Closing your R session

When closing down R, you will be asked whether you want to save your R workspace. Your R workspace contains all the data and plots that you have created. At the end of an R session, you can save the current workspace and will automatically reload when you reopen R.