Quantifying the Health Impacts of Air Pollution

Day 1: The Health Impacts of Air Pollution

In this session we will introduce the `R` statistical software and demonstrate its functionality through an example. The example performs an assessment of health impacts of outdoor air pollution.

An introduction to `R`

`R` is an integrated suite of software facilities for data manipulation, calculation and graphical display. It provides an environment within which almost all classical and modern statistical techniques have been implemented. A few of these are built into the base R environment, but many are supplied as packages. There are about 25 packages supplied with R and many thousands more are available through the CRAN family of Internet sites (cran.r-project.org)

`RStudio` is an integrated development environment (IDE) for `R`. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

Both `R` (cran.r-project.org) and `RStudio` (www.rstudio.com) are open source and can be downloaded free of charge. They are both available for Windows, Mac OSX and Linux.

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).

To help others who wish to do similar analyses, developers often make these packages publicly 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")
```

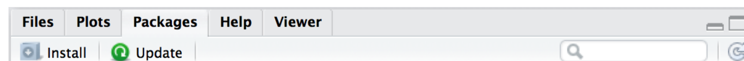
Alternatively, we can install packages clicking the 'Install' button in the 'Packages' pane and searching for the required package.

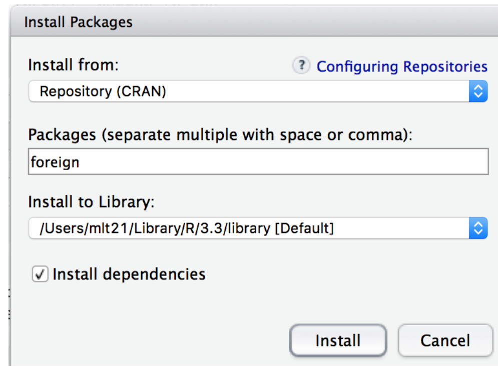
Once packages are downloaded and installed, we need to load them before using functions and/or data inside. We do this by using the `library()` function.

```
library(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 of RStudio.





Getting Help with R

CRAN (the repository for R software 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

The basic components of data analysis using R

When working in `RStudio`, 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
```

Storing numbers as variables

We can assign numbers (and many more things) to named variables so that we can use them again or use them as part of other calculations. To assign a number, for example, to a variable we `<-`.

```
# 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, be aware that assigning a result to a variable will overwrite what was already there.

Vectors

A vector is a simple tool to store data. They contain a collection of numbers or text. You can create a vector using the concatenates function `c()`. This function takes a series of numbers or text, separated by commas, and puts them together 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 `help(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 information 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 from 'a' and 'b'
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
```

Activities

- 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`.

Datasets in `R`

`R` also has an extensive repository of example datasets which we can use. 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 available within `R`.

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

Have a look at the `iris` dataset (Edgar Anderson's Iris Data, https://en.wikipedia.org/wiki/Iris_flower_data_set)

```
# Load the iris dataset in the workspace
data(iris)
```

Getting to know the structure of dataframes

Once a dataframe has been loaded into `R`, you should understand the data stored within. Initially, 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 iris dataset
nrow(iris)
[1] 150

# Viewing the first 5 rows of the iris dataset
ncol(iris)
[1] 5
```

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(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"

# Viewing the structure of the iris dataset
```

```
str(iris)
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

# View the first 6 rows of the iris dataset
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

# View the last 6 rows of the iris dataset
tail(iris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

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 `Sepal.Width` from `iris`, (ii) the first row from of `iris` and (iii) 3rd row for variable `Petal.Length` from `iris`.

```
# Extracting the variable a from mydata
iris[,'Sepal.Width']
  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9
 [18] 3.5 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2
 [35] 3.1 3.2 3.5 3.6 3.0 3.4 3.5 2.3 3.2 3.5 3.8 3.0 3.8 3.2 3.7 3.3 3.2
 [52] 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 2.0 3.0 2.2 2.9 2.9 3.1 3.0 2.7
 [69] 2.2 2.5 3.2 2.8 2.5 2.8 2.9 3.0 2.8 3.0 2.9 2.6 2.4 2.4 2.7 2.7 3.0
 [86] 3.4 3.1 2.3 3.0 2.5 2.6 3.0 2.6 2.3 2.7 3.0 2.9 2.9 2.5 2.8 3.3 2.7
[103] 3.0 2.9 3.0 3.0 2.5 2.9 2.5 3.6 3.2 2.7 3.0 2.5 2.8 3.2 3.0 3.8 2.6
[120] 2.2 3.2 2.8 2.8 2.7 3.3 3.2 2.8 3.0 2.8 3.0 2.8 3.8 2.8 2.8 2.6 3.0
[137] 3.4 3.1 3.0 3.1 3.1 3.1 2.7 3.2 3.3 3.0 2.5 3.0 3.4 3.0

# Extracting the row (or observation) from mydata
iris[1,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa

# Extracting the 3rd row for variable b from mydata
```

```r
iris[3,'Petal.Length']
[1] 1.3
```

Alternatively, you can extract variables from dataframes we using the `$` operator. We first specify the dataset then give the name of the variable that we want. Lets extract the variable `Sepal.Width` from our new dataframe `iris`.

```r
# Extracting the variable a from mydata
iris$Sepal.Width
  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9
 [18] 3.5 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2
 [35] 3.1 3.2 3.5 3.6 3.0 3.4 3.5 2.3 3.2 3.5 3.8 3.0 3.8 3.2 3.7 3.3 3.2
 [52] 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 2.0 3.0 2.2 2.9 2.9 3.1 3.0 2.7
 [69] 2.2 2.5 3.2 2.8 2.5 2.8 2.9 3.0 2.8 3.0 2.9 2.6 2.4 2.4 2.7 2.7 3.0
 [86] 3.4 3.1 2.3 3.0 2.5 2.6 3.0 2.6 2.3 2.7 3.0 2.9 2.9 2.5 2.8 3.3 2.7
[103] 3.0 2.9 3.0 3.0 2.5 2.9 2.5 3.6 3.2 2.7 3.0 2.5 2.8 3.2 3.0 3.8 2.6
[120] 2.2 3.2 2.8 2.8 2.7 3.3 3.2 2.8 3.0 2.8 3.0 2.8 3.8 2.8 2.8 2.6 3.0
[137] 3.4 3.1 3.0 3.1 3.1 3.1 2.7 3.2 3.3 3.0 2.5 3.0 3.4 3.0
```

Creating a new variable within a dataframe is straightforward. Let's create a variable `Sepal.Width.Plus.Sepal.Length` within `iris` which is the sum of the variables `Sepal.Width` and `Petal.Length`. For this we extract the variables `Sepal.Width` and `Sepal.Length` from `iris`, sum them and assign the results to `Sepal.Width.Plus.Sepal.Length` in `iris`.

```r
# Creating variable c by adding a and b together
iris$Sepal.Width.Plus.Sepal.Length <- iris$Sepal.Width + iris$Sepal.Length

# Printing the variable aplusb
iris$Sepal.Width.Plus.Sepal.Length
  [1]  8.6  7.9  7.9  7.7  8.6  9.3  8.0  8.4  7.3  8.0  9.1  8.2  7.8  7.3
 [15]  9.8 10.1  9.3  8.6  9.5  8.9  8.8  8.8  8.2  8.4  8.2  8.0  8.4  8.7
 [29]  8.6  7.9  7.9  8.8  9.3  9.7  8.0  8.2  9.0  8.5  7.4  8.5  8.5  6.8
 [43]  7.6  8.5  8.9  7.8  8.9  7.8  9.0  8.3 10.2  9.6 10.0  7.8  9.3  8.5
 [57]  9.6  7.3  9.5  7.9  7.0  8.9  8.2  9.0  8.5  9.8  8.6  8.5  8.4  8.1
 [71]  9.1  8.9  8.8  8.9  9.3  9.6  9.6  9.7  8.9  8.3  7.9  7.9  8.5  8.7
 [85]  8.4  9.4  9.8  8.6  8.6  8.0  8.1  9.1  8.4  7.3  8.3  8.7  8.6  9.1
 [99]  7.6  8.5  9.6  8.5 10.1  9.2  9.5 10.6  7.4 10.2  9.2 10.8  9.7  9.1
[113]  9.8  8.2  8.6  9.6  9.5 11.5 10.3  8.2 10.1  8.4 10.5  9.0 10.0 10.4
[127]  9.0  9.1  9.2 10.2 10.2 11.7  9.2  9.1  8.7 10.7  9.7  9.5  9.0 10.0
[141]  9.8 10.0  8.5 10.0 10.0  9.7  8.8  9.5  9.6  8.9
```

Activities

- Create a new variable called `Sepal.Width.Times.Sepal.Length` in `iris` which multiplies `Sepal.Width` and `Petal.Length` together.

Subsetting dataframes

We can use subsetting to easily access specific data from dataframes. 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 whether 1 == 2
1 == 2
[1] FALSE

# Checking whether1 == 1
1 == 1
[1] TRUE
```

Suppose we are interested in extracting the rows of the dataframe `iris` where the variable `Sepal.Width` is greater than 3. To subset data, we use the `subset()` function.

```
# Subsetting iris where `Sepal.Width` is less than 3
subset(iris, Sepal.Width > 4)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
16          5.7         4.4          1.5         0.4  setosa
33          5.2         4.1          1.5         0.1  setosa
34          5.5         4.2          1.4         0.2  setosa
   Sepal.Width.Plus.Sepal.Length
16                          10.1
33                           9.3
34                           9.7
# And assign it to a new dataframe
BigIris <- subset(iris, Sepal.Width > 3)
```

Activities

- Use subsetting to extract the rows of `iris` where the variable `Sepal.Width` is equal to 2.9.
- Use subsetting to extract the rows of `iris` where the variable `Sepal.Width` is less than 2.9 and `Sepal.Length` greater than 3.

Example: HIA associated with outdoor air pollution

We now demonstrate how using dataframes can allow us to perform HIA very efficiently. Following the example on air pollution you have already seen, we will be calculating the annual number of deaths attributable to $PM_{2.5}$.

We wish to estimate the annual number of deaths attributable to $PM_{2.5}$ air pollution. In order to do this, we need

- a relative risk (RR),
- the population at risk for the areas of interest,
- the overall mortality rate (OMR), and
- a baseline value for air pollution (for which there is no associated increase in risk).

In this example, we use a RR of 1.06 per $10\mu gm^{-3}$ increase in $PM_{2.5}$ exposure, the population at risk is 1 million and the OMR is 80 per 10000. We first enter this information into `R` by assigning these values to a series of variables.

```
# Relative Risk
RR <- 1.06
```

```
# Size of population
Population <- 1000000

# Unit for the Relative Risk
RR_unit <- 10

# Mortality rate
OMR = 80/10000

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

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)
```

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

For the second category, with $\text{PM}_{2.5} = 15$, 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, $\text{PM}_{2.5}$ is $10\mu\text{gm}^{-3}$ 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, $\text{PM}_{2.5} = 25$ which is again $10\mu\text{gm}^{-3}$ (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.

$$\text{RR}^{\left(\frac{\text{Category-Baseline}}{\text{Units}}\right)}$$

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 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.

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

# We can then calculate the excess numbers of deaths for each category
Impacts$Extra.Deaths <- Impacts$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.

```r
# Calculate the number of deaths gained
diff(Impacts$Extra.Deaths)
 [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))

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

Activities

- A different city has a population of 9.6 million and an overall mortality rate of 95 per 100,000 people. Carefully change the above code to recalcualte the health burden associated with $PM_{2.5}$.
- Suppose now we have an updated relative risk which is now 1.15 per $10\mu gm^{-3}$. Carefully change the above code to recalcualte the health burden associated with $PM_{2.5}$.

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 then everything will automatically reload when you reopen `R`.