switchR

What is R and why use it?

R is a free software environment with its own programming language.

It is especailly suited for statistical analysis and graphical outputs.

R's popularity as a data science tool as well as it being open source has made its applications vast.

R can work with a large variety of data formats and is (with a few add-ons) compatible with data from other software solutions (Excel, SPSS, SAS, Stata).

R compared to other analysis software

Compared to analysis software like Stata and SPSS, the R language is much closer to a programming language.

R itself has a very limited GUI (graphical user interface). RStudio provides more options but the GUI does not assist in the data analysis tasks (i.e. there is no point-and-click options for using R as such).

A lot of R's versatility comes from the fact that R treats everything as different objects (more on this later). Compared to most other analysis software, this means that you are often working with several data sets simultaneously.

The RStudio environment

RStudio is an IDE for R (Integrated Development Environment). It makes for a nicer workspace providing a better overview and a GUI for the different project elements (scripts, plots, packages, environment and so on).

RStudio is free to use and can be downloaded here:
https://www.rstudio.com/products/rstudio/download/

The R language

R has it's own programming language. R works by you writing lines of code in that language (writing commands) and R interpreting that code (running commands).

R (and RStudio) has a limited user interface meaning almost all functionality (statistics, plots, simulations etc.) must be executed using code in the R language.

R as a calculator

So what does it mean that R interprets our code? It means that you tell R to do something by writing a command and R will do that (if R can understand you).

R, for example, understands mathematical expressions:

```
7 * 6
```

```
[1] 42
```

```
912 - 132
```

```
[1] 780
```

Using R scripts

Script files are text files containing code that R can interpret.

It is your "analysis recipe" showing what you have done as well as allowing you to re-run commands easily.

Always make a habit of writing your commands into a script, when you have the command figured out.

- `#` can be used for comments (skipped when run)
- `Ctrl` + `Enter`: Runs the current line or selection
- `Ctrl` + `Alt` + `R`: Runs the whole script

NOTE! There is no undo in R. When a code is executed, the change has been made. The only way to undo is to re-run previous code to get back to an earlier stage. This is what scripts are used for.

The R Language: Objects and Functions

R works by storing values and information in "objects". These objects can then be used in various commands like calculating a statistical model, saving a file, creating a graph and so on. To simplify a bit: An object is some kind of stored information and a function is something that can manipulate that stored information (which then creates a new object).

Most of R can be boiled down to these 3 basic steps:

1. Assign values to an object
2. Make sure R interprets the object correctly (its class)
3. Perfom some operation or manipulation on the object using a function

The R Language: Objects and Functions

Translated to data analysis, the steps would (in general terms) look as follows:

1. Load our dataset: `dataset <- read.csv("my_datafile.csv")`
2. Check the that the variables are the correct class: `class(dataset$age)`
3. Perform some kind of analysis: `mean(dataset$age)`

The gap between these steps of course vary greatly.

Objects

A lot of writing in R is about defining objects: A name to use to call up stored information.

Objects can be a lot of things:

- a word
- a number
- a series of numbers
- a dataset
- a URL
- a formula
- a result
- a filepath
- a series of datasets
- and so on...

When an object is defined, it is available in the current working space (or environment).

This makes it possible to store and work with a variety of informaiton simultaneously.

Defining objects - numbers

Objects are defined using the <- operator (`Alt` + `-`):

```
year <- 1964
print(year)
```

```
[1] 1964
```

When defined the object can be used like any other numeric value.

```
year + 10
```

```
[1] 1974
```

Notice that R differentiates between lower- and upper-case letters:

```
Year # Does not exist
```

```
Error in eval(expr, envir, enclos): object 'Year' not found
```

Defining objects - strings

Using ' ' or " " denotes that the input should be read as text. *This also applies to numbers!*

```
name <-  "keenan"
print(name)
```

```
[1] "keenan"
```

```
year_now  <- '2021'
print(year_now)
```

```
[1] "2021"
```

Defining objects - strings

Notice that numbers stored as text will be enclosed in quotes. Numbers stored as text cannot immediately be used as numbers:

```
year_now - 5
```

```
Error in year_now - 5: non-numeric argument to binary operator
```

This error happens because R differentiates between objects by assigning them to a specific *class*. The class denotes what is possible with the object.

Naming objects

Objects can be named almost anything but a good rule of thumb is to use names that are indicative of what the object contains.

Restrictions for naming objects

- Most special characters not allowed: `/`, `?`, `*`, `+` and so on (most characters mean something to R and will be read as an expression)
- Already existing names in R (will overwrite the function/object in the environment)

Good naming conventions

- Using '_': `my_object`, `room_number`

or:

- Capitalize each word except the first: `myObject`, `roomNumber`

Classes in R

R differentiates between objects via the "class" of the object. The class determines what operations are possible.

The function `class()` is used to check the class of an object:

```
name = "keenan"
year = 1964
```

```
class(name)
```

```
[1] "character"
```

```
class(year)
```

```
[1] "numeric"
```

Class coercion

In most cases, R can coerce values from one class to another. When doing this, values that are incompatible with the class are coded to missing (`NA`) so beware!

Values can be coerced to character values with `as.character()`

Values can be coerced to numeric values with `as.numeric()`

```
as.character(year)
```

```
[1] "1964"
```

```
as.numeric(name)
```

```
[1] NA
```

Booleans / logical values

"booleans" or "logical values" are values that are either `TRUE` or `FALSE`.

A number of operations in R always return a logical value:

- `>`
- `>=`
- `<`
- `<=`
- `==`
- `!=`

```
42 > 10
```

```
[1] TRUE
```

```
10 != 10
```

```
[1] FALSE
```

Functions

Functions are commands used to transform an object in some way and give an output.

The input to a function is an "argument". The number of arguments vary between function.

Functions have the basic syntax: `function(arg1, arg2, arg3)`.

Some arguments are required while others are optional.

```
name <- 'kilmister'
toupper(name) #Returns the object in upper-case
```

```
[1] "KILMISTER"
```

```
gsub("e", "a", name) #Replace all e's with a's
```

```
[1] "kilmistar"
```

Some functions also return a logical value:

```
startsWith("R", "potato")
```

R Libraries - Packages

R being open source means that a lot of developers are constantly adding new functions to R. These new functions are distributed as *R packages* that can be loaded into the R library.

All the commands you have been using so far have been part of the `base` package (ships with R).

Packages are installed using (name of package *with* quotes!):

```
install.packages('packagename')
```

The functions from the package is loaded into the environment using (name of package *without* quotes!):

```
library(packagename)
```

Information for installed packages can be found using (name of package *with* quotes!):

```
library(help = 'packagename')
```

R Libraries - Packages

```
ymd('2021-02-04') # function does not exists - part of package l
```

```
Error in ymd("2021-02-04"): could not find function "ymd"
```

```
library(lubridate) # read in package
ymd('2021-02-04') # use function
```

```
[1] "2021-02-04"
```

R Objects: Vectors

A "vector" is a basic data structure in R. A vector can be considered a seris of values of the same class.

Vectors are created using `c()`:

```
names  <- c('araya', 'keenan', 'townsend')
years <- c(1961, 1964, 1972)

print(names)
```

```
[1] "araya"    "keenan"   "townsend"
```

```
print(years)
```

```
[1] 1961 1964 1972
```

```
mean(years)
```

```
[1] 1965.667
```

R Objects: Vectors

Notice that vectors can only store values of the same type/class.

When trying to combine different types in a vector, R will coerce all values to a type compatible with all values (if possible)

```
names_years <- c('araya', 1961, 'keenan', 1964)
print(names_years) # Notice the numbers are now converted to tex
```

```
[1] "araya"  "1961"   "keenan" "1964"
```

Vectors can only contain values of the same class. The `class()` function therefore works on vectors too.

```
class(names_years)
```

```
[1] "character"
```

Types of vectors

There are six types of vectors: logical, integer, double, character, complex, and raw.

The types primarily used for data analysis are: logical, integer, double, character.

"integer" and "double" are both referred to as *numeric vectors* (whole number and decimal point, respectively).

The type of vector can be examined with either `typeof` or `class`:

```
print(class(names))
```

```
[1] "character"
```

```
print(class(years))
```

```
[1] "numeric"
```

```
print(typeof(years))
```

```
[1] "double"
```

R Objects: Data Frames

A "data frame" is the R-equivalent of a spreadsheet (a table of rows and columns). It is one of the most useful storage structures for data analysis in R.

R has some sample datasets that can be loaded in with the `data()` command. `mtcars` is one of such sample datasets:

```
data(mtcars)
```

Data frames and vectors

Data frames are essentially a collection of same length vectors.

R treats single columns (or variables) as "vectors".

One refers to a single column in a data frame with $ (a vector).

```
head(mtcars$mpg) # First six values of yrbrn variable
```

```
[1] 21.0 21.0 22.8 21.4 18.7 18.1
```

Data frames and vectors

Each value in a vector is assigned an index refering to the position of the value in the vector (starts from 1).

A vector is indexed using `[]`:

```
mtcars$mpg[10] # Returns the 10th value (row 10) of the yrbrn va
```

```
[1] 19.2
```

```
mtcars$mpg[2:10] # Returns value 2-10 of the yrbn variable (both
```

```
[1] 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
```

Data frames and vectors

A range of useful functions exist for calculating descriptive measures for a vector; fx `mean()`, `min()`, `max()` and `length()`.

```
min(mtcars$mpg) # Returns smallest value
max(mtcars$mpg) # Returns largest value
mean(mtcars$mpg) # Returns mean value
length(mtcars$mpg) # Returns number of values in the vector (cor
```

`unique()` returns the unique values in a vector (useful for getting familiar with a variable):

```
unique(mtcars$gear)
```

```
[1] 4 3 5
```

Useful operations and functions on vectors

Below are some examples of different commands to interact with vectors.

Code	Description
`my_vec[-3]`	Everything but the 3rd element
`my_vec[c(1,4)]`	The 1st and 4th element
`my_vec[c(2:4)]`	The elements from index 2 to 4
`length(my_vec)`	The number of elements
`sort(my_vec)`	Sorts the elements in ascending order
`sum(my_vec)`	The sum of the vector elements (numeric)
`mean(my_vec)`	The mean of the vector elements (numeric)
`min(my_vec)`	The vector element with the lowest value (numeric)
`max(my_vec)`	The vector element with the highest value (numeric)

R Objects: Lists

Lists are - simply put - collections of other R objects. This means that lists can be a collection of all kinds of objects regardless of class, type or data structure.

Lists are created using `list()`.

Lists are used in a variety of ways. It is for example the default data structure for any hierarchical data (like JSON). Some functions also returns outputs as list, because it returns several kinds of output (like a model that returns various estimates and input parameters).

Lists can also be used for iteration by repeating the same commands across each entry in a list (like performing the same data handling operations on each data frame in a list).

R Objects: Lists

```
a_list <- list(42, "keenan", c(9, 3, 2))
```

Lists are indexed using [] for the list element and [[]] for the content of the list element:

```
a_list[3] # Returns element 3 - a list of length 1
```

```
[[1]]
[1] 9 3 2
```

```
a_list[[3]] # Returns the content of element 3 - a vector of val
```

```
[1] 9 3 2
```

```
print(c(class(a_list[3]),
      class(a_list[[3]]))
      )
```

```
[1] "list"    "numeric"
```

Using the help function

All R functions and commands are thoroughly documented so you do not have to remember what every function does or even how it should be written.

Every function and command in R has its own help file. The help file describes how to use the various functions and commands.

The help file for a specific function is accessed using the operator ?

Working directories

R reads and writes data by specifying *paths*. These can either be specified as *absolute* paths or *relative* paths.

Absolute paths specifies the entire path from root to file, i.e. `C:/my_data/data.csv`

Relative paths are specified relative to the current working directory.

- `.`: current directory (assumed either way)
- `..`: go up one level.

If data fx is in the working directory, one only has to specify the filename of the dataset (`data.csv`). If it is located inside a data folder in the working directory, it is specified as `./data/data.csv`.

Working directories

By default, R will set the working directory to the user documents folder. One can check the current working directory with the command `getwd()`.

The working directory can be changed with the command `setwd()` (or in RStudio via `Session -> Set Working Directory -> Choose Directory...`).

R Projects

R Projects are used to simplify different working directories across projects. When creating an R project, R will set the working directory to the directory where the R project file is stored.

Furthermore, R stores the workspace data (the "workspace image") to an `.RData` file with the project. This allows to store different workspaces across different projects.

Data handling with R - `tidyverse`

The tidyverse is collection of R packages designed for data science. All packages share an underlying design philosophy with regards to data structures ("tidy data"). They also share syntax grammar, fx by always having input data as the first argument of a function.

Install the complete tidyverse with:

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

Filtering and subsetting

R supports filtering and subsetting from "base" operations but there are packages with more intuitive functions (like the packages in tidyverse: https://www.tidyverse.org/).

Compare and contrast

These two commands achieve the same result. Which is more intuitive?
Base:

```
subset <- ess18[ess18$gndr == 'Male', c('gndr', 'prtvtddk')]
```

Tidyverse:

```
subset <- ess18 %>%
    filter(gndr == 'Male') %>%
    select('gndr', 'prtvtddk')
```

Recoding and creating variables

Creating variables and (simple) recoding is usually done in the same way. The only difference being whether the recoding is assigned to a new variable or overwriting an existing (we are here only looking at recoding by arithmetic operations and not by replacing values).

In base R, we simply specify a variable that is not in the data and specify the contents:

```
ess18$inwth <- ess18$inwtm / 60 # Creating variable for length o

head(ess18$inwth)
```

```
[1] 1.0166667 1.1333333 1.4833333 0.8333333 1.2833333 0.8000000
```

```
ess18$inwth <- NULL # This line removes the variable
```

Recoding and creating variables using dplyr (tidyverse)

The function `mutate()` in `dplyr` is use for creating and recoding variables:

```
ess18 <- ess18 %>%
    mutate(inwth = inwtm / 60)

head(ess18$inwth)
```

```
[1] 1.0166667 1.1333333 1.4833333 0.8333333 1.2833333 0.8000000
```

Missing values

Data will often contain missing values. Missing values can denote a lot of things like a non-response, an invalid answer, an inaccessible information and so on.

Missing values are used to assign a value without assigning a value. They are denotes as `NA` in R.

The `summary()` function includes information about the number of missing values:

```
summary(ess18$inwtm)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  18.00   51.00   59.00   63.32   70.00  613.00       5
```

Missing values

Missing values are neither high or low in R. This means that it is not possible to perform computations on missing values:

```
min(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

```
max(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

```
mean(ess18$inwtm) # NA is neither high or low - returns NA
```

```
[1] NA
```

Usually one will have to deal with the missing values in some ways - either by replacing them or removing them.

Removing missing observations (listwise deletion)

`drop_na()` from `tidyr` is used for listwise deletion. If columns are specified, it would look for missing in those specific columns:

```
library(tidyr)
ess18_drop_all = drop_na(ess18)

print(c(dim(ess18),
        dim(ess18_drop_all))
)
```

```
[1] 1285   17  176   17
```

Removing missing observations (listwise deletion)

If columns are specified, it would look for missing in those specific columns:

```
ess18_drop_specific = drop_na(ess18, inwtm)

print(c(dim(ess18),
      (dim(ess18_drop_specific)))
)
```

```
[1] 1285   17 1280   17
```

```
ess18_drop_several = drop_na(ess18, inwtm, tygrtr)

print(c(dim(ess18),
        (dim(ess18_drop_several)))
)
```

```
[1] 1285   17 1130   17
```

Replacing missing values

`replace_na()` is used to replace missing values with a specified value. It can fx be used in combination with mutate:

```
ess18 %>%
    mutate(prtvtddk = replace_na(prtvtddk, 'MISSING')) %>%
    head()
```

```
# A tibble: 6 x 17
   idno netustm ppltrst vote  prtvtddk lvpntyr tygrtr gndr  yrbr
  <dbl>   <dbl>   <dbl> <chr> <chr>    <chr>   <chr>  <chr> <dbl
1  5816      90       7 Yes   SF Soci~ 1994    60     Male   197
2  7251     300       5 Yes   Dansk F~ 1993    40     Fema~  197
3  7887     360       8 Yes   Sociald~ 1983    55     Male   195
4  9607     540       9 Yes   Alterna~ 1982    64     Fema~  196
5 11688      NA       5 Yes   Sociald~ 1968    50     Fema~  195
6 12355     120       5 Yes   Sociald~ 1987    60     Male   196
# ... with 6 more variables: wkhct <dbl>, wkhtot <dbl>, grspnum
#   frlgrsp <dbl>, inwtm <dbl>, inwth <dbl>
```

Summarizing data

Summarizing data can be achieved by using base table commands. Alternatively, one could use `group_by()` and `summarise()` for creating summaries at group level.

Tables

Frequency and contingency tables can be done in a number of different ways in R dependent on the kind of tables you want to make.

R has a few built-in functions based around the `table()` function. The `table()` function is used for creating a table object that can then be manipulated.

Specifying a single variable creates a one-dimensional frequency table:

```
table(ess18$gndr)
```

```
Female   Male
   630    655
```

Tables

Specifying two variables creates a crosstable of counts of every combination:

```
table(ess18$gndr, ess18$vote)
```

```
        No Not eligible to vote Yes
  Female  21                   37 571
  Male    40                   38 576
```

Margin tables

The function `margin.table()` calculates frequencies with a table object as input.

```
ess_table <- table(ess18$gndr, ess18$vote) # creating table obje

margin.table(ess_table, 1) # gndr frequencies (row frequencies)
```

```
Female   Male
   629    654
```

```
margin.table(ess_table, 2) # brncntr frequencies (column frequen
```

```
                  No Not eligible to vote                     Yes
                  61                    75                    1147
```

Prop tables

The function `prop.table()` calculates percentages with a table object as input.

```
prop.table(ess_table, 1) # gndr percentages (rows)
```

```
                 No Not eligible to vote        Yes
  Female 0.03338633            0.05882353 0.90779014
  Male   0.06116208            0.05810398 0.88073394
```

```
prop.table(ess_table, 2) # brncntr percentages (columns)
```

```
                No Not eligible to vote       Yes
  Female 0.3442623            0.4933333 0.4978204
  Male   0.6557377            0.5066667 0.5021796
```

The CrossTable() function (part of gmodels)

The package gmodels contains the function CrossTable().

CrossTable combines the various table functionalities in base R for an easier way to create crosstables. It also makes it easier to include various tests of independence.

The line below creates a crosstable for vote and gndr, displaying percentages column-wise and calculating the chi-squared.

```
library(gmodels)

CrossTable(data$vote, ess18$gndr,
           prop.r = FALSE, prop.c = TRUE,
           prop.t = FALSE, prop.chisq = FALSE,
           chisq = TRUE)
```

Grouped summaries

`group_by()` is part of the `dplyr` package. `group_by()` is used together with `summarise()` for creating summary statistics.

Below the mean time spent on the internet per day per gender is calculated and displayed:

```
ess18 %>%
    group_by(gndr) %>%
    summarise(mean_internettime = mean(netustm, na.rm = TRUE))
```

```
# A tibble: 2 x 2
  gndr   mean_internettime
  <chr>              <dbl>
1 Female              224.
2 Male                231.
```

Grouped summaries

Several summary statistics can be created for the same grouping:

```
ess18 %>%
    group_by(gndr) %>%
    mutate(age = 2018 - yrbrn) %>%
    summarise(mean_age = mean(age),
              mean_internettime = mean(netustm, na.rm = TRUE),
              count = n())
```

```
# A tibble: 2 x 4
  gndr   mean_age mean_internettime count
  <chr>     <dbl>              <dbl> <int>
1 Female     51.0               224.   630
2 Male       50.9               231.   655
```

Grouped summaries

Observations can be grouped based on several variables:

```r
ess18 %>%
    group_by(gndr, vote) %>%
    mutate(age = 2018 - yrbrn) %>%
    summarise(mean_age = mean(age),
              mean_internettime = mean(netustm, na.rm = TRUE),
              count = n())
```

```
# A tibble: 8 x 5
# Groups:   gndr [2]
  gndr   vote                  mean_age mean_internettime count
  <chr>  <chr>                    <dbl>             <dbl> <int>
1 Female No                        42.9              225.    21
2 Female Not eligible to vote      31.4              259.    37
3 Female Yes                       52.6              221.   571
4 Female <NA>                      53                300      1
5 Male   No                        43.6              262.    40
6 Male   Not eligible to vote      27.3              283.    38
7 Male   Yes                       53.0              225.   576
8 Male   <NA>                      33                488      1
```

Tabulating with `tidyverse` and `group_by()`

There are various ways of creating tables and cross-tables using functions from the tidyverse.

`count()` (part of `dplyr`) can be used for frequency tables:

```
library(dplyr)

ess18 %>%
    count(gndr)
```

```
# A tibble: 2 x 2
  gndr       n
  <chr>  <int>
1 Female   630
2 Male     655
```

Tabulating with tidyverse and group_by()

Crosstables can be achieved by combining group_by() summaries with pivot_wider():

```
library(tidyr)

ess18 %>%
  group_by(gndr, vote)%>%
  summarise(n=n())%>%
  pivot_wider(names_from = gndr, values_from = n)
```

```
# A tibble: 4 x 3
  vote                  Female  Male
  <chr>                  <int> <int>
1 No                        21    40
2 Not eligible to vote      37    38
3 Yes                      571   576
4 <NA>                       1     1
```