Statistical Methods in Cancer Epidemiology using R

Janne Pitkäniemi

Faculty of Social Sciences, University of Tampere
Finnish Cancer Registry

Lecture 1

janne.pitkaniemi@cancer.fi>

Feb,10 2010

Contents

What is R

- Statistical software or ''package" — and a lot more
- R is a **language**and **environment**for statistical computing and graphics (`www.r-project.org/`)
- Developed by volunteers, coordinated by the **R Development Core Team**.
- Available for Windows, Linux, Mac, Unix, ...
- Is expanding rapidly: new version every 6 months.
- No licence fee(!) & source code open.

For further information and download:
{`http://www.r-project.org/`}

Properties of R

- Large repertory of basic and advanced methods.
- Versatile graphics of high quality.
- R Reads datasets from Stata, SAS, SPSS, Epi-Info – even Excel
- Deals simultaneously with different objects or data structures newline – not just a single data matrix.
- Results of analysis saved as **objects**, readily available for further processing.
- Parsimonious output listing!
- For advanced users! Easy to expand and tailor to specific needs using the **object-oriented**programming tools.

To learn more about R

- Hills, M., Plummer, M., Carstensen, B.
  A Short Introduction to R for Epidemiology, 2011.
  `http://bendixcarstensen.com/Epi/R-intro.pdf`

- Dalgaard, P. *Introductory Statistics with R, 2nd Ed.*
  Springer, New York, 2008.

- *Statistical Practice in Epidemiology Using R*. An international course, IARC, Tarto, 2020.
  `http://bendixcarstensen.com/SPE/`

- R blog

- Masses of books, articles, websites, etc ...

What does R offer for epidemiologists?

- Descriptive tools
  - Versatile tabulation
  - High-quality graphics
- Analytic methods
  - Basic epidemiologic statistics
  - Generalized linear models and their extensions
  - Survival analysis methods
  - Other ...

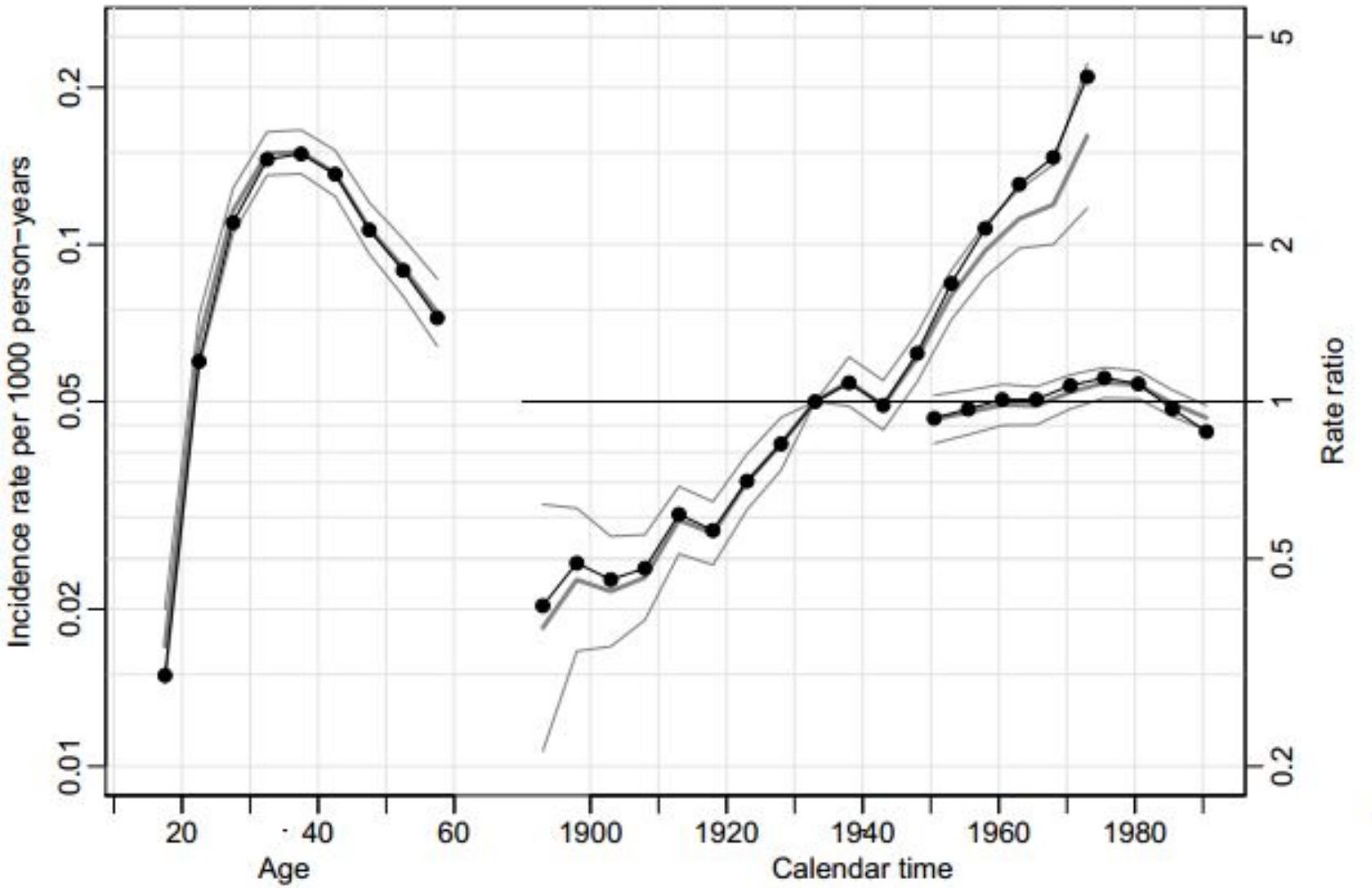
These are provided by SAS and Stata, too, so why R ...?

Many features of R are more appealing in the long run.

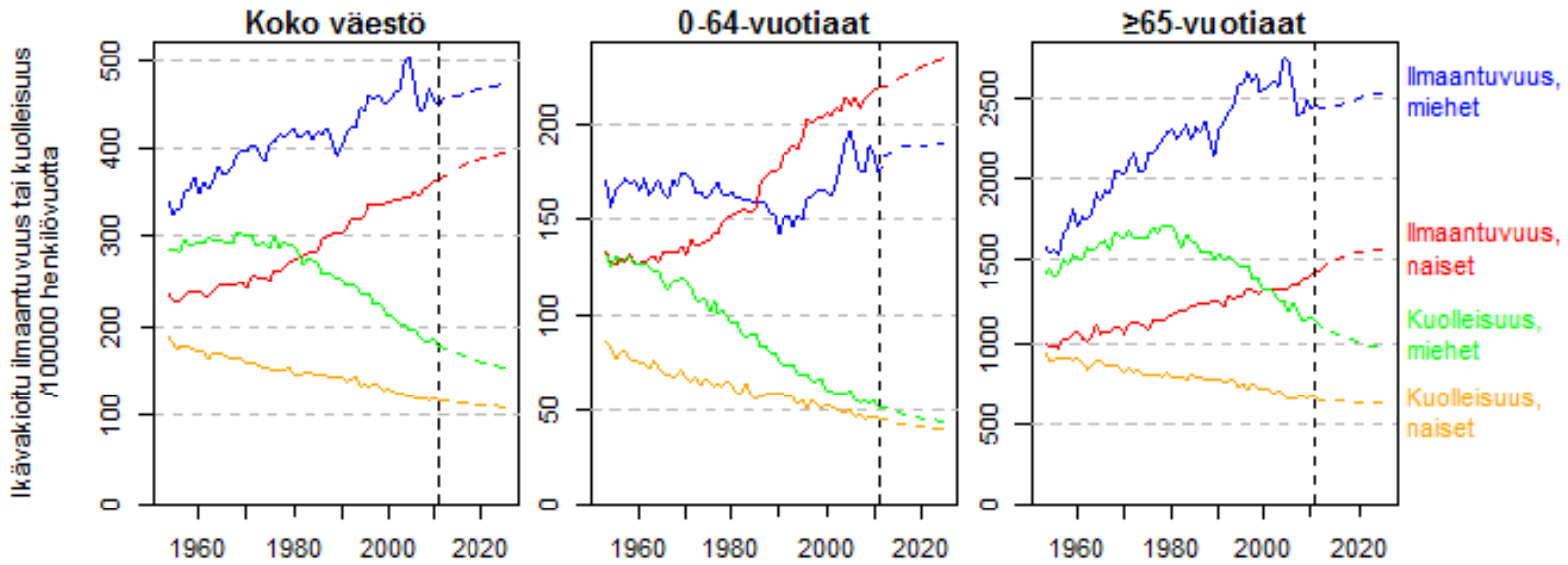
Graphics in R

- Versatile, flexible, high quality, . . .
- Easy to add items (points, lines, text, legends . . .) to an existing graph.
- Fine tuning of symbols, lines, axes, colours, etc. by *graphical parameters* ($>$ 67 of them!)
- Interactive tools using the mouse
  - Put new things on a graph
  - Identify points
- Modern lattice or *Trellis* graphics.
- Saving formats: Metafile, .pdf, .png, .bmp, .jpg, . . .

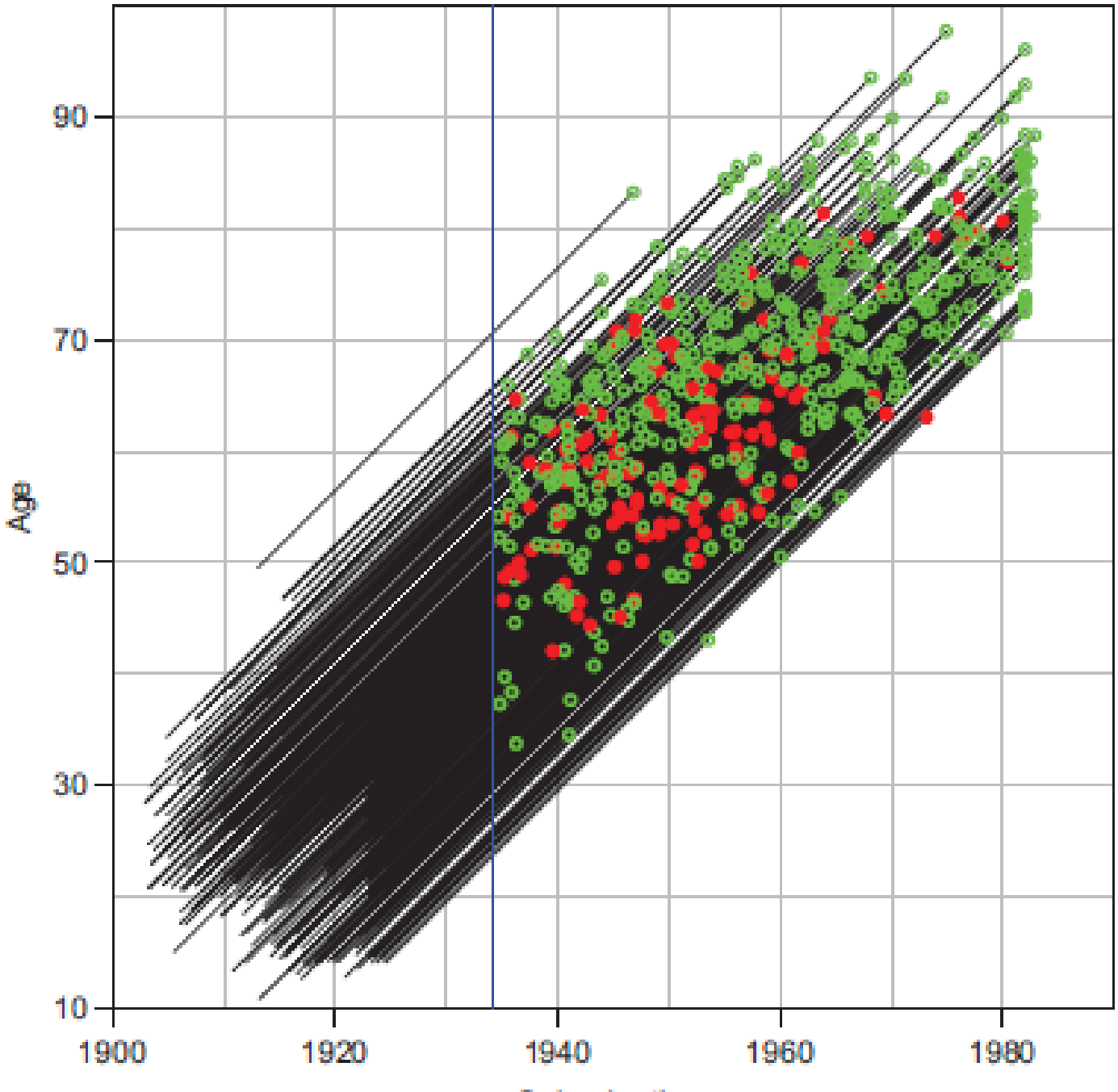
Age-period-cohort incidence in DK



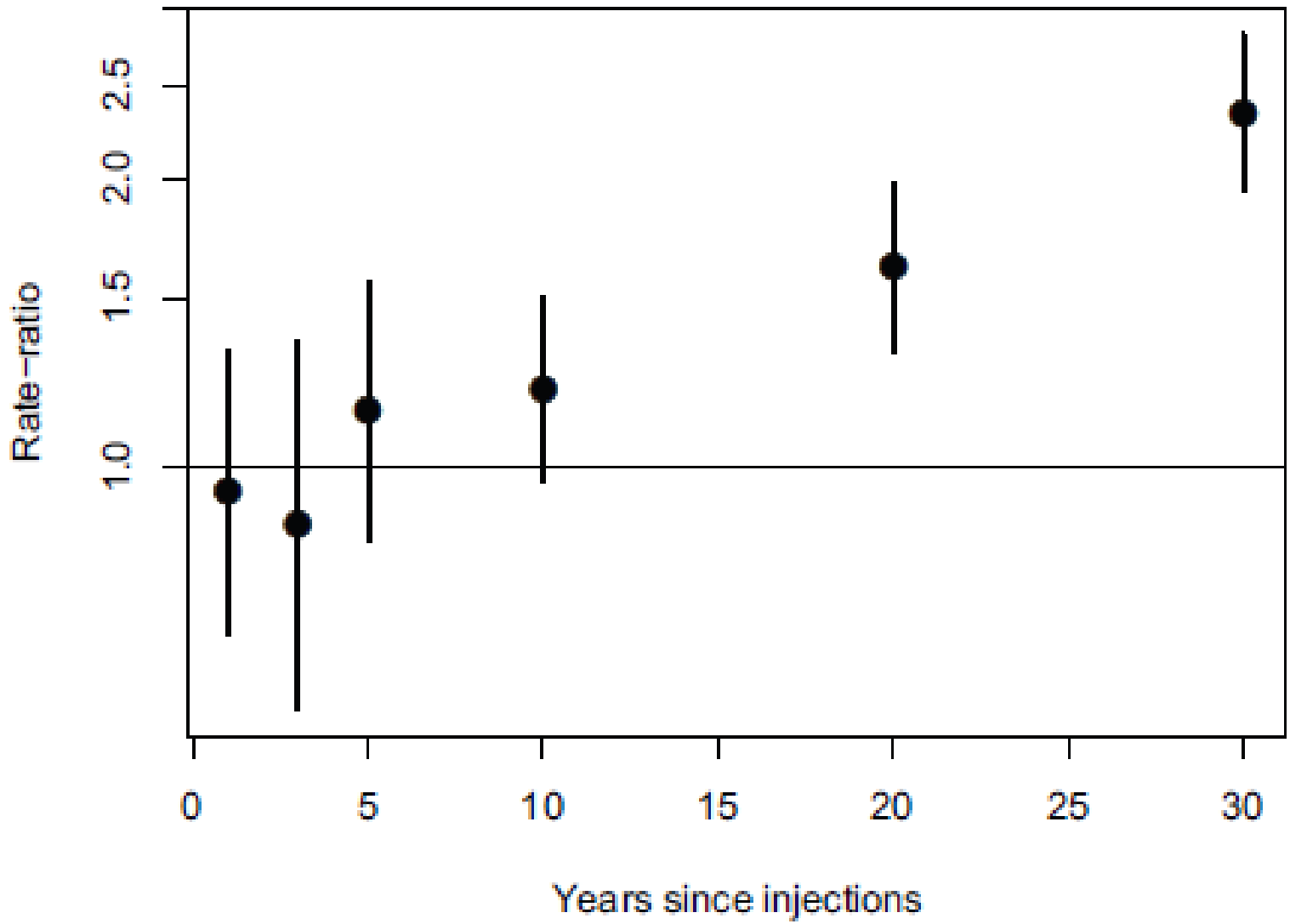
Cancer Predictions in Finland



Follow-up of Welsh nickel cohort in Lexis diagram



RRs & CIs by exposure in a cohort study



Package or library

- Collection of functions pertaining to some specialized application area, e.g. *survival, boot*
- Contributed by users of R.
- Available after loading:
  $>$ *library(survival)*
- Alternatively load from the menu bar:
  Packages - Load package. . . - Select one
- New versions easily updated from Internet.
  (https://www.rdocumentation.org/trends)

R script – R Studio – commands in a file

R script file is an ASCII file containing a sequence of R commands to be executed.

The **script editor** – use R-Studio

- In `R-Studio` open the script editor window: *New file - R script*, or when editing an existing *script file*: *File - Recent Files*,
- Save the `script file`: *Save* `e.g.` or *Save As* *.R
- Excecute a line *Ctrl-Enter*

R script (cont'd)}

- Paint the lines to be excecuted and *Ctrl -Enter* will execute lines.
- To run a whole script file, write in console window:
  `> source("c:/.../mycmds.R", echo=TRUE)`
- The script can also be written and edited by any external editor programs (like Notepad).

Data objects of different kinds

- `vector`: ordered set of similar elements
  e.g. real numbers or character sequences,
- `factor`: categorical variable with levels
  e.g. `gender`, levels: `c(1,2)` or `c('male', 'female')`;
- `matrix, array`: 2- and k-dimensional tables,
- `data.frame`: "data matrix" (more of this soon!),
- `ts`: time series object,
- `list`: sequence of different types of objects.

Attributes of data objects

Functions that extract some key properties of objects:

- `length( )`: number of elements,
- `mode( )`: basic type of elements,
- `dim( )`: dimensions of arrays, matrices and data frames,
- `str( )`: overall structure,
- `class( )`: property that determines how certain **generic functions** (*e.g.* `summary(); plot()`) work when the object is given as argument.

Data frame – data matrix

- [Data frame] = a **list** of column vectors
- Rows correspond to observational units, and columns (same length) refer to variables.
- Column vectors can be `numeric`, `character` or `logical`
- Columns are **subobjects** of the data frame. Their names are not directly accessible. Two possibilities:
  - Use "surname$firstname", e.g. `mydata$var1` ,
  - Place the data frame in the search path at position 2: `attach(mydata)`; then use just "firstname": `var1`

Data frame import from external files

- Common ASCII files, for example:
  read.table("C:/owndir/rfiles/mydata.txt", ...);
  read.table("http://cc.oulu.fi/~tilel/esan.txt",...)
- Files with fixed-width format: `read.fwf();`
- Files created in SPSS, SAS, Stata *etc.*: functions
  `read.spss()`, `read.ssd()`, `read.dta()`, *etc.*
  in package `foreign`,
- Excel-files: either {`read.table("clipboard", ...)`}, or
  `(1) save the Excel-file in .csv or .txt format,`
  `(2) in R: read.csv2( ) or read.table( )`

- Relational DBMSs: several R packages available.

Data frame import from external files with R

Choose *Import Datasets*

- *from text (base)* for **text** files
- *from text (readr)* for **csv** files
- *from excel* for **excel** files
- *from SPSS* for **spss** files
- *from SAS* for **sas** files
- *from STATA* for **stata** files

Dealing with output

- The console contents, *i.e.* the flow of input commands and output results from the console window, can be
  \$ - \$ printed on paper: *File - Print...*
  \$ - \$ saved to an ASCII file: *File - Save to file...*

- Selected parts can be copied from the console and pasted to an external file.

- Function `sink("results.txt")` diverts all subsequent output to an external text file. Back to console: `sink()`.

- Choose *New File – R Markdown* output to MS-Word

- Graphs saved in desired format: *File - Save...*

R is a functional language

Most computations in R involve the {invocation} or {call} of functions. They are called by name with a set of arguments separated by commas, *e.g.* `fun(x, y, z)` ; **Function**

- $=$ sequence of rules on how to produce desired output:
- **value** of the function, from given input, *i.e.*
- **arguments** of the function.

Example: Function `sqrt()` computes square roots:

argument vector defined

```
x <- c(0,1,2,3,4)
```

call with argument `x`

```
sqrt(x)
```

```
[1] 0.000000 1.000000 1.414214 1.732051 2.000000
```

Defining a new function (1)

Example. Function `CIapp` to calculate an approximate confidence interval from point estimate (`estim`) and std error (`SE`) by formula $\texttt{estim} \pm z_{\gamma/2} \times \texttt{SE}$.

Defining code (without prompts):

```
CIapp <- function(estim, SE, level = 0.95)
  {z <- qnorm(1- (1-level)/2 )  # setting the quantile
 lower <- estim - z*SE
 upper <- estim + z*SE
 CIapp <- c(lower, upper)
 CIapp }
```

- **Formal arguments**, here `estim, SE, level`

Calling the new function (1)

- **Actual arguments**, used in function call:

```
CIapp(3, 1, 0.9)
```

```
[1] 1.355146 4.644854
```

NB! **Positional matching**: order of actual arguments.
- **Keyword matching**: the order of arguments in the call is irrelevant if the names of formal arguments are given

```
CIapp(SE=1.0, level=0.90, estim=3)
```

```
[1] 1.355146 4.644854
```

- If a **default value** for an argument is given in the definition and is OK, it can be omitted in calling

```
CIapp(3, 1)
```

```
[1] 1.040036 4.959964
```

Function call & value object

- Simple call: Evaluates the value of the function with given arguments and prints value items (according to the print **method** specific to the **class** of the value object).

- Call of function and assignment of its value to an object.

To extract information & items from the value object, *e.g.*

- `str()`: overall structure,
- `names()`: names of the components,
- `print()`: selective printing of value items,
- `summary()`: selective print (not available for all functions).

Example, function range()

Returns the minimum and maximum values of a data vector.

```
y <- c(15.3, 10.8, 8.1, 19.5, 5.3) # data vector
range(y) # simple call with argument
```

```
[1]  5.3 19.5
```

```
ra <- range(y) # call with assignment of value
ra # print(ra)
```

```
[1]  5.3 19.5
```

```
str(ra) # structure of the value object
```

```
 num [1:2] 5.3 19.5
```

```
ra[1]  #extracting an item from the value object
```

```
[1] 5.3
```

Different kinds of functions

- Mathematical, *e.g.* {sqrt(x); log(x); exp(x)}.
  Arguments and values typically numeric vectors.

- Data handling, *e.g.*
  {dafr <- data.frame(x, y);
  adata <- read.table("a.dat", header=T, ...);

- Graphical, *e.g.*
  {plot(y ∼ x); Main argument(s): data object(s).
  Value: graph.
  Ancillary arguments: *e.g.* graphical parameters.

Value of the function

- numeric object (*e.g.* vector, matrix) for many mathematical and statistical functions,
- data object (e.g. vector, data frame) for data handling functions,
- graph for graphical functions,
- table for tabulating functions,
- **list** = a sequence of objects of different kinds, for many statistical functions.

Statistical functions

- *Main* argument(s): Typically data object(s).
  Often a *model formula* like $\texttt{y} \sim \texttt{x}$ with
  $\texttt{y}$ representing the *response* variable and
  expression $\texttt{x}$ = *explanatory* variable(s) or factor(s).

- *Ancillary* arguments or *parameters*: additional specifications.
  Some default values usually offered for these.

- *Value*: Usually a {list} object consisting of several
  components of different types.

Function values as list objects

- **List** = object consisting of an ordered collection of component objects, maybe of different types.
- Provides a convenient way to return the results of statistical computation.
- A list with named components formed from existing objects:
  `${ }  $ {Lista <- list(name=obj1,title=obj2,addr=obj3)}`
  `A single component identified:`
  `{Lista{$}name};`
- `Concatenation of several lists into one:`
  `{longlist <- c(list1, list2, ...)}.`

Ex: Function *t.test()*

Description of syntax in the `help()` page

```
a<-help("t.test",help_type = "tex")
print(a)
```

Default S3 method: t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95, ...)

t.test(formula, data, subset, na.action, ...)

- Main argument(s): data vector(s) `x` (and `y`) or formula
- Ancillary arguments, like `var.equal, conf.level`: Default values given.
- **NB.** Dots ' `...`'

Example. Red cell folate levels

The data describe red cell folate levels (variable `folate`, μg/l) in two groups
of cardiac bypass surgery patients given two different nitrous oxide ventilation (50% NO + 50% O_2) treatments (variable `group`):

- group 1 ($n_1 = 8$) continuously for 24 h (label `"24 h"`),
- group 2 ($n_2 = 9$) only during the operation (`"oper"`).

Observed folate levels in the two groups:

```
folate[group=="24 h"]
```

```
[1] 243 251 275 292 347 354 380 392
```

```
folate[group=="oper"]
```

```
[1] 206 210 226 249 255 273 285 295 309
```

Ex: Call of `t.test()` by

formula argument

```
t.test(folate ~ group, var.equal=TRUE, conf.level=0.9)
```

```
    Two Sample t-test

data:  folate by group
t = 2.5653, df = 15, p-value = 0.02153
alternative hypothesis: true difference in means is not equ
90 percent confidence interval:
  19.09502 101.51610
sample estimates:
mean in group 24 h mean in group oper
          316.7500             256.4444
```

Ex: Value returned by `t.test()` is a *list*}

Function value assigned to an object and examined:

```
tfol <- t.test(folate ~ group, var.equal=TRUE, conf.level=0.9)
str(tfol)  # The structure of the object
```

```
List of 10
 $ statistic  : Named num 2.57
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 15
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.0215
 $ conf.int   : num [1:2] 19.1 101.5
  ..- attr(*, "conf.level")= num 0.9
 $ estimate   : Named num [1:2] 317 256
  ..- attr(*, "names")= chr [1:2] "mean in group 24 h" "mean in group oper"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ stderr     : num 23.5
 $ alternative: chr "two.sided"
 $ method     : chr " Two Sample t-test"
 $ data.name  : chr "folate by group"
 - attr(*, "class")= chr "htest"
```

```
names(tfol)
```

```
 [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
 [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

Ex: Value of t.test()} utilized}

- Extracting items for further processing:

```
tfol$estimate  # contents of the 'estimate' component
```

```
mean in group 24 h mean in group oper
          316.7500             256.4444
```

- Utilizing the component value in further calculations:

```
mean.diff <- tfol$estimate[1] -
                tfol$estimate[2]
```

- Item names in the parent object "inherited". Can be renamed:

```
names(mean.diff) <- c("Mean difference")
mean.diff
```

```
Mean difference
       60.30556
```

Defining a new function (2)}

We now create a new function `T.estimCI()`. It will return only the mean difference between the groups (which is not reported by `t.test()`!) and its confidence interval.

The function is defined as follows:

```
T.estimCI <- function(x, ... )
 { tt <- t.test(x, ...)
   mean.diff <- tt$estimate[1] - tt$estimate[2]
   names(mean.diff) <- c("Mean difference")
   T.estimCI <- list(Meandiff = mean.diff,
                     Conflimits = tt$conf.int)
   T.estimCI }
```

Calling the new function (2)

When `t.estimCI()` is called, a list with 2 named components is returned and printed:

```
T.estimCI(folate ~ group, var.equal=T, conf.level=0.9)
```

```
$Meandiff
Mean difference
       60.30556

$Conflimits
[1]  19.09502 101.51610
attr(,"conf.level")
[1] 0.9
```

Dealing with functions

- Defining code can (mostly) be viewed by typing the function name without parentheses and arguments.
- Functions can be saved into a separate script or source file, *e.g.* `myfuns.R`, which may contain several functions.
- Source file accessible in an R run after
  `> source("C:/.../myfuns.R")`
- Alternatively from menu bar: *File – Source R code . . . *
- Loading from Internet:
  `> source("http://.../myfuns.R")`

Tabulation functions

- `table(c1, c2)`: simple contingency tables
- `tapply(var,fac,fun)` tabulates values of function `fun()` (for example `mean()`) applied to values of variable `var` in categories of factor `fac`,
- ```
  stat.table( index = list(rvar, cvar),
    contents = list(count(), percent(rvar) ),
    ...  )
  ```
  in package `Epi` for more informative tabulation.
- package plyr and ddply-funtion ...
- package **data.table** for BIG data ...
- missing variables...
- other ...
  ```