

# R: language and basic data management

Krista Fischer

Statistical Practice in Epidemiology, Lyon, 2018  
(initial slides by P. Dalgaard)

# Language

- ▶ R is a programming language – also on the command line
- ▶ (This means that there are *syntax rules*)

On the command line (or a line in a script) one could:

- ▶ Print an object by typing its name
- ▶ Evaluate an expression
- ▶ Call a function, giving the arguments in parentheses – possibly empty
- ▶ Notice `objects` **vs.** `objects()`

# R expressions

```
x <- rnorm(10, mean=20, sd=5)
m <- mean(x)
sum((x - m)^2)
```

- ▶ Object **names**
- ▶ Explicit **constants**
- ▶ Arithmetic **operators**
- ▶ **Function calls**
- ▶ **Assignment** of results to names

# Objects

- ▶ The simplest object type is *vector*
- ▶ Modes: numeric, character, factor, ...
- ▶ Operations are vectorized: you can add entire vectors with  
 $a + b$
- ▶ Recycling of objects: If the lengths don't match, the shorter vector is reused

## Example (numeric vectors)

```
> a <- c(2, 8, 3, 1, 0, 7)
> b <- c(3, 4, 1, 4, 5, 2)
> a+b
[1] 5 12 4 5 5 9
> mean(a)
[1] 3.5
> m <- mean(a)
> m
[1] 3.5
> a - m # notice recycling
[1] -1.5 4.5 -0.5 -2.5 -3.5 3.5

> z <- c(1, 2, 3)
> a - z #recycling!
[1] 1 6 0 0 -2 4
```

## Character vectors and factors

- ▶ The elements of character vectors are text strings that do not have any numeric value.
- ▶ **Factors** are used to describe groupings – these are just integer codes plus a set of names, as labels for the *levels*
- ▶ In model specifications, a factor variable is treated as a classification rather than as a quantitative variable

Example:

```
> x<-c(1, 3, 3, 2, 1, 3, 1)
> fx<-factor(x, labels=c("bad", "average", "good"))

> fx
[1] bad      good      good      average  bad      good

> levels(fx)
[1] "bad"      "average"  "good"
```

# Lists

- ▶ Lists are vectors where the elements can have different types – thus collections of any elements, gathered into one object
- ▶ Functions often return lists
- ▶ `lst <- list(A=rnorm(5), B="hello")`
- ▶ Special indexing:
  - ▶ `lst$A`
  - ▶ `lst[[1]]` first element (NB: double brackets)
- ▶ **Data frames** are special type of lists

# Matrices

- ▶ A **matrix** is a rectangular collection of data. All columns of a matrix should be of the same type.

```
> A<-matrix(c(1,4,2,6,7,8),nrow=3,ncol=2,
```

```
> A
```

	[, 1]	[, 2]
[1, ]	1	4
[2, ]	2	6
[3, ]	7	8

- ▶ One can also construct a matrix from its columns using `cbind`, whereas joining two matrices with equal no of columns (with the same column names) can be done using `rbind`.



# Data frames

- ▶ A **data frame** is similar to a matrix, but can have columns (variables) of different types
- ▶ A variable can be extracted using `dataframe$variable` (as data frames are lists)

```
> D<- data.frame(a=c(8,3,5),b=c("X","Z","Y"))
> D
  a b
1 8 X
2 3 Z
3 5 Y
> D$a
[1] 8 3 5
```

## Matrices or data frames?

- ▶ A (numeric or character) matrix can be converted to a data frame and vice versa (with `as.data.frame(A)` and `as.matrix(B)`).
- ▶ Most R functions for statistical analysis work with data frames, but in some cases it is useful to have a matrix (incl the occasions where you want to use some matrix algebra).
- ▶ If you need more dimensions than two, there is also `array`.

# The workspace

- ▶ The *global environment* contains R objects created on the command line.
- ▶ There is an additional *search path* of loaded packages and attached data frames.
- ▶ When you request an object by name, R looks first in the global environment, and if it doesn't find it there, it continues along the search path.
- ▶ The search path is maintained by `library()`, `attach()`, and `detach()`
- ▶ Notice that objects in the global environment may mask objects in packages and attached data frames

## How to access variables in the data frame?

Different ways to tell R to use variable X from data frame D:

- ▶ As mentioned, you can use the `dataframe$variable` notation

```
summary(D$X)
```

- ▶ Use the `with` function

```
with(D, summary(X))
```

- ▶ Use the `data` argument (works for some functions only)

```
lm(Y~X, data=D)
```

- ▶ Attach the dataframe – **DISCOURAGED!**  
(seems a convenient solution, but can actually make things more complicated, as it creates a temporary copy of the dataset)

```
attach(D)  
summary(X)  
detach()
```

## Data manipulation

To create a new variable in the data frame, you could use:

```
students$bmi <-  
  with(students, weight/(height/100)^2)
```

Or:

```
students <-  
  transform(students, bmi=weight/(height/100)^2)
```

(notice: you need an assignment, to save the transformed object)

whereas

```
with(students, bmi <- weight/(height/100)^2)
```

creates the variable `bmi` in the global environment (not in the data frame).

# Indexing – extracting elements from objects

**Square brackets [ ] are used for indexing!**

Examples:

- ▶ Elements of vectors: `a[5]` (5th element); `a[5:7]` (5th to 7th elements); `a[-6]` (all elements except the 6th)
- ▶ Logical index: `a[a<3]`, `a[b>2]`, `a[is.na(b)]`  
(elements of `a` corresponding to missing values of `b`)
- ▶ In a dataframe or matrix – two dimensions, two indexes:  
`students[5, 7]`, `students[1:10, c(1,2,5)]`,  
`students[students$age>40, ]`

## Examples of indexing

```
> x<- c(2,7,3,1,5,9,0)
> x[c(1,5,7)]
[1] 2 5 0
> x[x<3]
[1] 2 1 0
> A[2,1]      # a matrix has row and column indexes
[1] 2
> A[2,]       # the entire 2nd row
[1] 2 6

> NMRimp[1:2,1:4]      #quick look at a large data
  sample.id XXL.VLDL.P XXL.VLDL.L XXL.VLDL.PL
1    V18566   1.46e-04    0.0313    0.00331
2    V36115   9.00e-05    0.0195    0.00178

> fgsa[is.na(fgsa$height),"age"]
[1] 18 69 52 41 52 44 73 28 66 20 73 63 26
# equivalent: fgsa$age[is.na(fgsa$height)]
```

## Conditional assignment: `ifelse`

- **Syntax:** `ifelse(expression, A, B)`  
Expression (with values TRUE or FALSE) is a vector, A and B are constants or vectors of the same length.

Examples:

```
> x<-c(1,2,7,3,NA)
> ifelse(x<3,1,2)
[1] 1 1 2 2 NA
> ifelse(is.na(x),0,x) #replace missing values by 0
[1] 1 2 7 3 0
> y<-c(3,6,1,7,8); z<-c(0,1,0,2,1)
> ifelse(z==0,x,y)
[1] 1 6 7 7 8
> ifelse(is.na(x),0,ifelse(x>3,3,x))
[1] 1 2 3 3 0
```



# Naming

- ▶ Elements of vectors, rows and columns of matrices and data frames can have names

```
> x <- c(boys=1.2, girls=1.1)
> x
  boys girls
  1.2   1.1
> x["boys"]
boys
  1.2
> D[, "a"] # works for matrices and data frames
[1] 8 3 5
```

- ▶ You can extract and set names with `names(x)`; for matrices and data frames also `colnames(x)` and `rownames(x)`;

# Classes, generic functions

- ▶ R objects have *classes*
- ▶ Functions can behave differently depending on the class of an object
- ▶ E.g. `summary(x)` or `print(x)` does different things if `x` is numeric, a factor, or a linear model fit

```
> summary(x) # a numeric vector
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1      1      2      2      3      3

> summary(fx) # a factor
  bad average    good
    3      1      3
```

# Function calls

Round brackets ( ) are used for function calls!

Lots of things you do with R involve calling functions (you have seen that already!).

For instance

```
mean(x, na.rm=TRUE)
```

The important parts of this are

- ▶ The **name** of the function
- ▶ **Arguments**: input to the function
- ▶ Sometimes, we have **named arguments**

# Function arguments

Examples:

```
rnorm(10, mean=m, sd=s)
hist(x, main="My histogram")
mean(log(x + 1))
```

Items which may appear as arguments:

- ▶ **Names** of R objects
- ▶ Explicit **constants**
- ▶ **Return values** from another function call or expression
- ▶ Some arguments have their *default values*.
- ▶ Use `help(function)` or `args(function)` to see the arguments (and their order and default values) that can be given to any function.
- ▶ Quite often – first argument is not named, but the others are named

## Example

From R-help (`help(t.test)`):

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

- ▶ The first argument (x) does not have a default – you have to provide some data!
- ▶ The other arguments can be modified, if you need to.

The following lines of code are equivalent:

```
t.test(a, b, alternative="less", paired=TRUE)  
t.test(a, b, paired=TRUE, alt="less")  
t.test(a, b, p=T, a="l")    #not a good style!
```

Order does not matter for named arguments!

Partial keyword matching is possible (“alternative” or “alt” or “a”)  
(partial matching is possible)

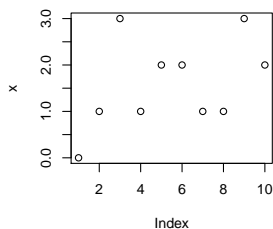
## Basic graphics

The `plot()` function is a generic function, producing different plots for different types of arguments. For instance, `plot(x)` produces:

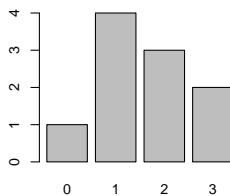
- ▶ a plot of observation index against the observations, when `x` is a numeric variable
- ▶ a bar plot of category frequencies, when `x` is a factor variable
- ▶ a time series plot (interconnected observations) when `x` is a time series
- ▶ a set of diagnostic plots, when `x` is a fitted regression model
- ▶ Similarly, the `plot(x, y)` produces a scatter plot, when `x` is a numeric variable and a bar plot of category frequencies, when `x` is a factor variable

## Some simple plots:

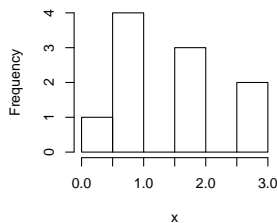
**plot(x)**



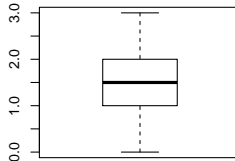
**plot(factor(x))**



**hist(x)**



**boxplot(x)**



## More on factors: the `cut` Function

- ▶ The `cut` function converts a numerical variable into groups (a factor variable) according to a set of break points
- ▶ The intervals are left-open, right-closed by default (`right=FALSE` changes that)
- ▶ ...and that the lowest endpoint is *not* included by default (set `include.lowest=TRUE` if it bothers you)



## Example

```
> age <- c(35,20,21,50,46,23,30)
> agegr<-cut(age, c(20,30,40,50))
> table(agegr)
agegr      # the 20-year old is not included!
(20,30] (30,40] (40,50]
      3         1         2
> agegr<-cut(age, c(20,30,40,50),right=FALSE)
> table(agegr)
agegr      # the 50-year old is not included!
[20,30) [30,40) [40,50)
      3         2         1
> agegr<-cut(age, c(20,30,40,50),right=FALSE,
+              include.lowest=TRUE)
> table(agegr)
agegr
[20,30) [30,40) [40,50]
      3         2         2
```

## Working with Dates

- ▶ Dates are usually read as character or factor variables
- ▶ Use the `as.Date` function to convert them to objects of class "Date"
- ▶ If data are not in the default format (YYYY-MM-DD) you need to supply a format specification

```
> as.Date("11/3-1959", format="%d/%m-%Y")  
[1] "1959-03-11"
```

- ▶ You can calculate differences between `Date` objects. The result is an object of class "difftime". To get the number of days between two dates, use

```
> as.numeric(as.Date("2017-6-1") -  
              as.Date("1959-3-11"), "days")  
[1] 17607
```

## Creating your own functions

A very simple example:

```
logit <- function(p)  log(p/(1-p))
```

The function `logit` requires one argument *p* and produces the logit of *p*. Try `logit(0.5)`, or `logit(0.25)`,...

More complex (but still simple):

```
simpsum <- function(x, dec=5) {  
  m <- mean(x, na.rm=TRUE)  
  s  <- sd(x, na.rm=TRUE)  
  round(c(mean=m, sd=s), dec) }
```

The function `simpsum` requires one argument *x*, but the second argument `dec` (no of decimal points in the output) has a default value 5. Try `simpsum(a)`, or `simpsum(a, dec=2)`.