Introduction to R

Oliver Kirchkamp

Rev. 29— 30th November 2013

Contents

1 Introduction

1.1 Statistical software

A comparison of statistical software

A personal opinion:

	R	SAS	Stata	SPSS
Free	+	−	−	−
Documentation	≈ /+	+	+	
Convenience		+	+	−
Correctness	+	+/-	+/−	−
Power	+	+	+/−	−
Quick to learn	−	−	+/−	+

- One advantage of a "free" (as in free beer) software is that it is portable. When you move to a new job you can still use your old work.
- Some of the commercial packages include a lot of documentation which is very well structured and follows a consistent pattern. The documentation that comes with R is sometimes less well structured. However, a few books can make up for this very soon.
- Correctness is always a problem, but it seems to be a smaller problem with R. The following example illustraes:

A Wilcoxon rank-sum test in R

```
wilcox.test(x ~ group)


Wilcoxon rank sum test

data:  x by group
W = 5, p-value = 0.04798
alternative hypothesis: true location shift is not equal to 0
```

A Wilcoxon rank-sum test in Stata

```
. ranksum x ,by(group)

Two-sample Wilcoxon rank-sum (Mann-Whitney) test

       group |      obs    rank sum    expected
-------------+---------------------------------
           1 |        5          20        32.5
           2 |        7          58        45.5
-------------+---------------------------------
    combined |       12          78          78

unadjusted variance       37.92
```

```
adjustment for ties          0.00
                        ----------
adjusted variance           37.92

Ho: x(group==1) = x(group==2)
             z =  -2.030
    Prob > |z| =   0.0424
```

The two p-values from R and Stata differ. R uses the *exact* method, Stata uses an approximation.

1.2 Ways to interact with R

There are lots of possible front ends:
Figure 1 shows RStudio.
Figure 2 shows the RCommander.
Figure 3 shows the command line.

1.3 The help system

From the command line we might have to say `help.start()` to start the interactive help system.

Getting help for R

```
help.start()
```

`RSiteSearch` searches R functions, package vignettes, and task views (regardless whether they are locally installed).

```
RSiteSearch("probit")
```

`help.search` searches the help system for a given topic:

```
help.search("probit")
```

This will give you a list of Vignettes, demos, and functions.
Use `vignette` to show a specific vignette.

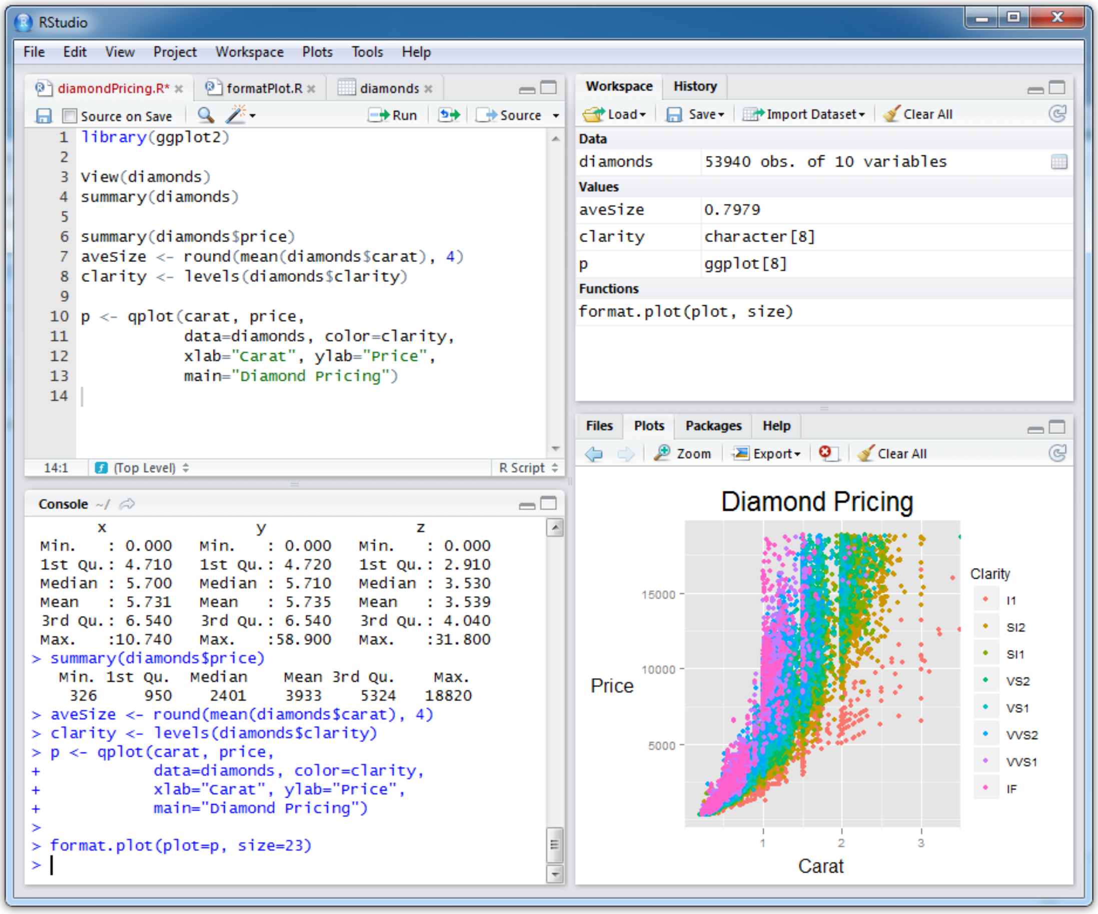


Figure 1: The RStudio Interface

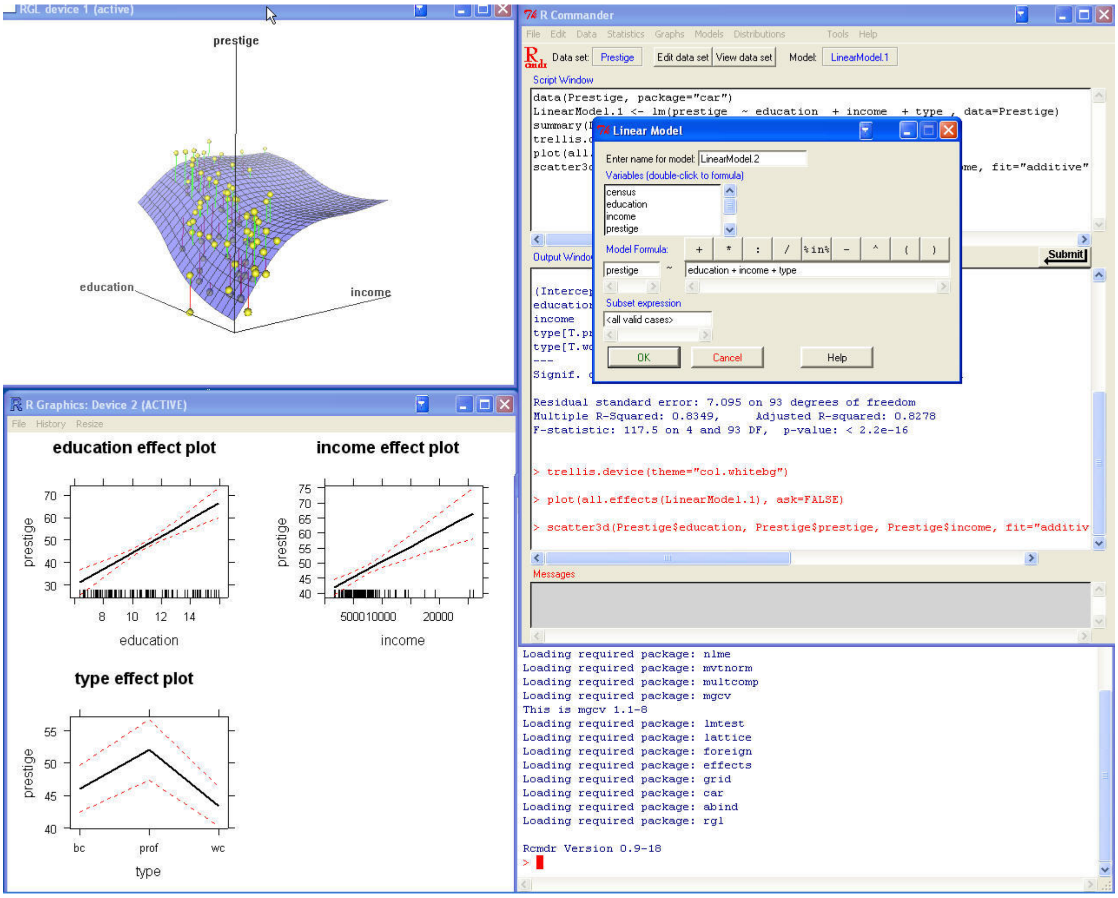


Figure 2: The RCommander Interface

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> |
```

Figure 3: The command-line interface

```r
vignette("probit", package = "Zelig")
```

Use `demo` to play a demo.

```r
library(Zelig)
demo(Zelig::probit)
```

To learn about the syntax of a specific function, use `help` or `?`

```r
? glm
help("glm")
```

2 Datatypes

2.1 Overview

To get a first idea, let us load one of the many buitin datasets, and have a look at the first three rows of this dataset:

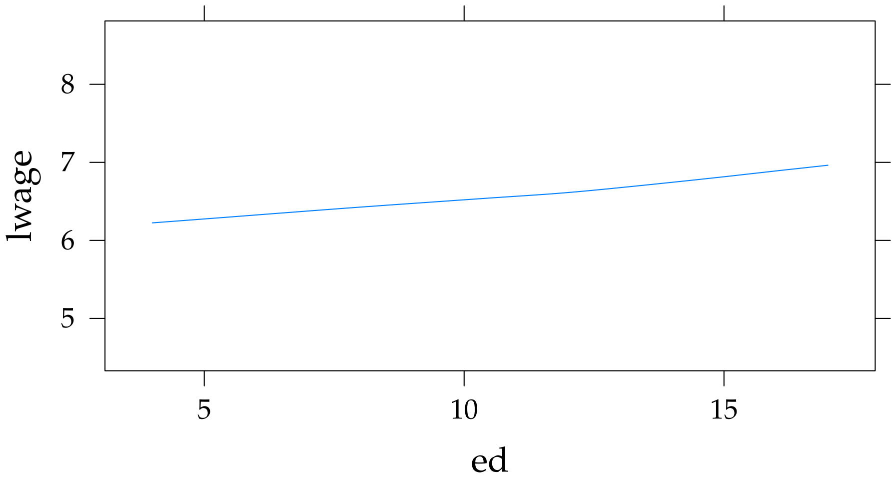
A data frame

```r
data(Wages, package = "Ecdat")
```

	exp	wks	bluecol	ind	south	smsa	married	sex	union	ed	black	lwage
1	3	32	no	0	yes	no	yes	male	no	9	no	5.56
2	4	43	no	0	yes	no	yes	male	no	9	no	5.72
3	5	40	no	0	yes	no	yes	male	no	9	no	6.00

```r
lm(lwage ~ ed, data = Wages)


Call:
lm(formula = lwage ~ ed, data = Wages)

Coefficients:
(Intercept)           ed
     5.8388       0.0652
```

```r
library(lattice)
xyplot(lwage ~ ed, data = Wages, type = "smooth")
```



Some primitive datatypes:

numbers	3, 42, 3.141592...
logicals	`TRUE`, `FALSE`
characters	`"Jena"`, `"Bonn"`,...
factors	`"Jena"`, `"Bonn"`,...
formulas	`y ~ x + z`
functions	`function(x) x^2`

Combining data

We can combine data of the *same type*:

- Vector (several members of the same type): $(3, 42, 3.1415926\ldots)$
- Matrix (rectangular): $\begin{pmatrix} 3 & 42 \\ 3.14 & 0 \end{pmatrix}$
- Array (multidimensional) $\left(\begin{pmatrix} 3 & 42 \\ 3.14 & 0 \end{pmatrix}, \begin{pmatrix} 12 & 16 \\ 21 & 11 \end{pmatrix}\right)$

We can also combine data of *different types*:

- List: (3, `"Jena"`, `FALSE`, ...)
- Data frame (rectangular, different types per column)

	exp	wks	bluecol	ind	south	smsa	married	sex	union	ed	black	lwage
1	3	32	no	0	yes	no	yes	male	no	9	no	5.56
2	4	43	no	0	yes	no	yes	male	no	9	no	5.72
3	5	40	no	0	yes	no	yes	male	no	9	no	6.00

2.2 Numbers

We see, R handles very different datatypes: numbers, characters, ... Let us start with something simple, with numbers. We can do usual arithmetic like this:

Numeric: simple arithmetic

```
2 + 3
```

```
[1] 5
```

```
2 - 3
```

```
[1] -1
```

```
2 * 3
```

```
[1] 6
```

```
2/3
```

```
[1] 0.6667
```

```
2^3
```

```
[1] 8
```

```
2^3 + 1/3
```

```
[1] 8.333
```

```
(2^3 + 1)/3
```

```
[1] 3
```

```
7 < 11
```

```
[1] TRUE
```

```
7 == 11
```

```
[1] FALSE
```

The assignment operator: `<-`

We can assign the result of our calculations to variables with `<-`. Usually, assignments produce no output.

```
x <- 2 + 2
```

We have to repeat the name of the variable to which we assigned our result to see something:

```
x

[1] 4
```

Names of variables

Variable names contain `a-z`, `A-Z`, `0-9` and `.` and `_` and start with a letter, e.g.

```
x
groupContribution5
group.contribution
group_contribution
```

In the example we have seen several operators. Here is a list:

Some operators

- Arithmethic operators: `+`, `-`, `*`, `/`, `^`,...
- Assignment: `<-`, `<<-`, `->`, `=`
- Logical: `==`, `<=`, `<`, `>`, `>=`
- and more ...(you can define you own operators)

2.3 Using functions

Functions and parameters

A function with one parameter:

```
log(100)

[1] 4.605
```

Calling a function with several **named** parameters:

```
log(x = 100, base = 10)

[1] 2
```

```r
log(base = 10, x = 100)
```

```
[1] 2
```

Functions and parameters
Calling a function with several **positional** parameters:

```r
log(100, 10)
```

```
[1] 2
```

Mixing **positional** and **named** parameters:

```r
log(base = 10, 100)
```

```
[1] 2
```

The arguments of a function
If we want to find out the arguments of a function:

```r
args(log)
```

```
function (x, base = exp(1))
NULL
```

We see that the paramter `base` has a *default* value (`exp(1)`).
If `base` is not specified (e.g. we just say `log(100)`), R assumes `base=exp(1)`

Getting help for a function We can find out what parameters a function takes and how they are called and in which position they come with the `help` function.

```r
help(log)
```

```
log                    package:base                    R Documentation

Logarithms and Exponentials

Description:

     'log' computes logarithms, by default natural logarithms, 'log10'
     computes common (i.e., base 10) logarithms, and 'log2' computes
     binary (i.e., base 2) logarithms.  The general form 'log(x, base)'
     computes logarithms with base 'base'.
```

```
     'log1p(x)' computes log(1+x) accurately also for |x| << 1 (and
     less accurately when x is approximately -1).

     'exp' computes the exponential function.

     'expm1(x)' computes exp(x) - 1 accurately also for |x| << 1.

Usage:

     log(x, base = exp(1))
     logb(x, base = exp(1))
     log10(x)
     log2(x)

     log1p(x)

     exp(x)
     expm1(x)

Arguments:

       x: a numeric or complex vector.

    base: a positive or complex number: the base with respect to which
          logarithms are computed.  Defaults to e='exp(1)'.

Details:
  ...

References:
  ...

See Also:
   ...

Examples:
   ...
```

2.4 Vectors

The simplest datatype in R is a vector.

Vectors

```r
c(21, 22, 23, 24, 25, 26, 27, 28, 29, 30)
```

```
 [1] 21 22 23 24 25 26 27 28 29 30
```

Since we need sequences like the above very often, there is a special operator that helps us to create sequences:

```r
21:30
```

```
 [1] 21 22 23 24 25 26 27 28 29 30
```

":" is nothing else but a shortcut for the function `seq`:

```r
seq(21, 30)
```

```
 [1] 21 22 23 24 25 26 27 28 29 30
```

Descending sequences are done like this:

```r
30:21
```

```
 [1] 30 29 28 27 26 25 24 23 22 21
```

The parameter `by` sets the step width of the sequence:

```r
seq(21, 30, by = 2)
```

```
[1] 21 23 25 27 29
```

```r
seq(21, 22, by = 0.2)
```

```
[1] 21.0 21.2 21.4 21.6 21.8 22.0
```

Alternatively `length` defines how many numbers we want within a given interval:

```r
seq(21, 22, length = 6)
```

```
[1] 21.0 21.2 21.4 21.6 21.8 22.0
```

2.5 The recycling rule

What happens if we combine two vectors? If both have the same length, the operation is performed for each matching element:

Recycling (when vectors have different lengths)

```
x <- c(1, 2, 3)
y <- c(10, 10, 10)
x + y

[1] 11 12 13
```

What if the two vectors do not have the same length? Then the elements of the shorter vector are *recycled*. This makes in particular sense if one vector has length one:

```
x <- c(1, 2, 3)
y <- 10
x + y

[1] 11 12 13
```

```
10^(0:3)

[1]    1   10  100 1000
```

```
log(c(0.1, 1, 10, 100), base = 10)

[1] -1  0  1  2
```

Here is an example where one vector has length 3 and the other length 2:

```
x <- c(1, 2, 3)
y <- c(10, 20)
x + y

Warning:  longer object length is not a multiple of shorter object length

[1] 11 22 13
```

```
(1:2)^(0:3)

[1] 1 2 1 8
```

2.6 Arithmetic with vectors

Arithmetic with vectors

```
x <- c(1, 2, 3)
y <- c(3, 4, 5)
x + y

[1] 4 6 8
```

element-wise product:

```
x * y
```

```
[1]  3  8 15
```

inner product:

```
x %*% y
```

```
     [,1]
[1,]   26
```

outer product:

```
x %o% y
```

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    8   10
[3,]    9   12   15
```

2.7 More on numbers

R distinguishes floating point numbers (`double`) and integer numbers (`integer`). The standard type for numbers is `double`.

We can find out the type of a variable with the help of the function `typeof`.

Floating point numbers

```
x <- c(1, 2, 3)
typeof(x)
```

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

`double` has about 16 significant decimal digits precision.
Largest exact integer number: 9,007,199,254,740,992
For numbers this large, R usually prints the first 7 significant digits:

```
2^53
```

```
[1] 9.007e+15
```

If we insist, we can see more digits:

```r
print(2^53, digits = 16)
```

```
[1] 9007199254740992
```

Floating point numbers

However, adding small quantities to such a large number leads to a rounding error:

```r
print(2^53 + 1, digits = 16)
```

```
[1] 9007199254740992
```

Only if we add a larger quantity (here 2), we get a visible increment.

```r
print(2^53 + 2, digits = 16)
```

```
[1] 9007199254740994
```

Floating point numbers

Even if we accept a rounding error, the is a largest and a smallest number R can represent:

Largest positive real: $1.797693 \cdot 10^{308}$

```r
2^1023.99999999
```

```
[1] 1.798e+308
```

```r
2^1024
```

```
[1] Inf
```

Smallest positive real: $4.940656 \cdot 10^{-324}$

```r
2^-1074
```

```
[1] 4.941e-324
```

```r
2^-1075
```

```
[1] 0
```

Integers

We can *force* a number to be an interger:

```
x <- as.integer(c(1, 2, 3))
typeof(x)
```

```
[1] "integer"
```

Largest `integer` number: 2,147,483,647

```
as.integer(2147483647 + 1)
```

```
Warning:  NAs introduced by coercion
```

```
[1] NA
```

We see that if an interger number becomes “too large” it becomes `NA`

Functions

- Arithmetic `log`, `exp`, `sin`, ...
- Aggregation:

  `mean`, `sum`, `sd`, `var`, ...

  `summary`
- Data:

  `c`, `seq`, `:`

  `order(c(2,3,1))=c(3,1,2)`

  `sort(c(2,3,1))=c(1,2,3)`

  `sample(c(1,2,3))=c(2,1,3)`

  `unique(c(1,1,2,2,3)=c(1,2,3)`

  `which(c(1,1,2,2,3)==2)=c(3,4)`

  `length(c(1,2,3))=3`

  `typeof`, `class`, `as.numeric`, `is.numeric`, `as.integer`, ...
- Randomness

  `runif`, `rnorm`, `rt`, `rf`, `rchi`, `rpoisson`, `rbinom` ,...

3 Not exactly numbers

3.1 Characters

Textual information is represented as `characters`:

Characters

```
x <- c("07745", "Jena", "Kahlaische Straße", "10")
x

[1] "07745"             "Jena"              "Kahlaische Straße"
[4] "10"
```

Again, we can find out the type of a variable with *typeof*:

```
typeof(x)

[1] "character"
```

We can also force this variable to `numeric`:

```
as.numeric(x)

Warning:  NAs introduced by coercion

[1] 7745   NA   NA   10
```

Pasting strings together

And we can *collapse* the different elements of this character vector to a single character string:

```
paste(x, collapse = " ")

[1] "07745 Jena Kahlaische Straße 10"
```

Functions

- `as.numeric`, `as.character`, `is.character`, ...
- `paste`

3.2 Factors

Factors are a way to store characters more efficiently. They associate a number with a meaning.

Factors

Let us start with a character:

```r
x <- c("07745", "Jena", "Kahlaische Straße", "10")
x
```

```
[1] "07745"              "Jena"               "Kahlaische Straße"
[4] "10"
```

When we convert `x` to a `factor`, it looks almost the same:

```r
y <- as.factor(x)
y
```

```
[1] 07745             Jena              Kahlaische Straße 10
Levels: 07745 10 Jena Kahlaische Straße
```

Something new appears. `x` has now `levels`, i.e. R has stored the different values our character variable could take.

```r
levels(y)
```

```
[1] "07745"              "10"                 "Jena"
[4] "Kahlaische Straße"
```

Factors

```r
y
```

```
[1] 07745             Jena              Kahlaische Straße 10
Levels: 07745 10 Jena Kahlaische Straße
```

To store the original character values, it is now sufficient to only remember the number of the `level`.

```r
as.numeric(y)
```

```
[1] 1 3 4 2
```

```r
levels(y)[as.numeric(y)]
```

```
[1] "07745"              "Jena"               "Kahlaische Straße"
[4] "10"
```

Factors are useful, when a variable takes only a limited number of values (e.g. `male`, `female` or `Jena`, `Berlin`, `Bonn`)

If something is not a factor, but we still want to find unique values, we can use `unique`:

```r
unique(c(1, 2, 3, 1, 2, 3, 1, 2))
```

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

Functions

- `as.factor`
- `levels`
- `reorder`
- `relevel`

3.3 Logicals

We use logicals when a variable can be only `TRUE` or `FALSE`.

Logicals

```r
c(TRUE, FALSE, TRUE, TRUE)
```

```
[1]  TRUE FALSE  TRUE  TRUE
```

The operator `!` inverts `TRUE` and `FALSE`.

```r
!c(TRUE, FALSE, TRUE, TRUE)
```

```
[1] FALSE  TRUE FALSE FALSE
```

Logicals can be coerced as `numeric`. `TRUE` becomes `1`, `FALSE` becomes `0`.

```r
as.numeric(c(TRUE, FALSE, TRUE, TRUE))
```

```
[1] 1 0 1 1
```

Logicals are automagically coerced to numeric when we calculate with them. Thus, `sum` counts the number of `TRUE` values, `mean` calculates the relative fraction of `TRUE`.

```r
sum(c(TRUE, FALSE, TRUE, TRUE))
```

```
[1] 3
```

```r
mean(c(TRUE, FALSE, TRUE, TRUE))
```

```
[1] 0.75
```

Functions

- `!`, `&`, `|`, `&&`, `||`, `xor`, `...`
- `ifelse( c(TRUE,FALSE,TRUE), c(1,2,3), c(10,20,30)) = c(1,20,3)`
- `sum`, `mean`,`...`

3.4 Dates

Sometimes it might be sufficient to store dates as characters. However, when we want to calculate with dates, we should store values `as.Date`. R tries to guess the format from the character string, but it is always safe to specify the format:

Dates

```r
as.Date("24/Jul/2012", format = "%d/%b/%Y")
```

```
[1] "2012-07-24"
```

```r
x <- as.Date("Aug/24/2012", format = "%b/%d/%Y")
x
```

```
[1] "2012-08-24"
```

We can use `format` to print a string in a different format:

```r
format(x, format = "%a, %d.%m.%Y")
```

```
[1] "Fri, 24.08.2012"
```

```r
format(x + 1, format = "%A, %d.%m.%Y")
```

```
[1] "Saturday, 25.08.2012"
```

Dates

We can now calculate the difference between two dates:

%a	abbreviated weekday (e.g., Sun)
%A	full weekday (e.g., Sunday)
%b	abbreviated month (e.g., Jan)
%B	full month name (e.g., January)
%C	century (e.g., 20)
%d	day of month (e.g., 01)
%e	day of month, space padded; same as %_d
%g	last two digits of year of ISO week number (see %G)
%G	year of ISO week number (see %V); normally useful only with %V
%h	same as %b
%j	day of year (001..366)
%m	month (01..12)
%u	day of week (1..7); 1 is Monday
%U	week number of year, with Sunday as first day of week (00..53)
%V	ISO week number, with Monday as first day of week (01..53)
%w	day of week (0..6); 0 is Sunday
%W	week number of year, with Monday as first day of week (00..53)
%x	locale's date representation (e.g., 12/31/99)
%y	last two digits of year (00..99)
%Y	year

Table 1: Date formats

```
y <- as.Date("Dec/24/2012", format = "%b/%d/%Y")
y - x

Time difference of 122 days
```

```
as.numeric(y - x)

[1] 122
```

R uses the date formats shown in table 1.

Date formats

Date and time

If we want to store date and time together, we can use two datatypes:

- `POSIXct`: (signed) number of seconds since the beginning of 1970 (in the UTC timezone) as a numeric vector.

  This is easy to integrate into a data frame.

- `POSIXlt`: is a named list of vectors representing `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday`, `isdst`.

  This is difficult to integrate into a data frame.

Date and time

```
x <- as.POSIXct("Dec/24/2012 20:00:00", format = "%b/%d/%Y %H:%M:%S")
x

[1] "2012-12-24 20:00:00 CET"
```

```
format(x, format = "%a, %d.%m.%Y")

[1] "Mon, 24.12.2012"
```

```
format(x, format = "%A, %e %B '%y, %l:%M %p")

[1] "Monday, 24 December '12,  8:00 PM"
```

Date and time

As above, we can calculate with date-time objects:

```
y <- as.POSIXct("Aug/21/2011 07:30:00", format = "%b/%d/%Y %H:%M:%S")
x - y

Time difference of 491.6 days

as.numeric(x - y)

[1] 491.6
```

Calculations are done on the level of seconds:

```
x + 1000

[1] "2012-12-24 20:16:40 CET"
```

Time formats

R uses the date formats shown in table 2.

`%H`	hour (00..23)
`%I`	hour (01..12)
`%k`	hour, space padded (0..23); same as %_H
`%l`	hour, space padded (1..12); same as %_I
`%M`	minute (00..59)
`%p`	either AM or PM
`%P`	like %p, but lower case
`%r`	locale's 12-hour clock time (e.g., 11:11:04 PM)
`%R`	24-hour hour and minute; same as %H:%M
`%S`	second (00..60)
`%T`	time; same as %H:%M:%S
`%z`	+hhmm numeric time zone (e.g., -0400)
`%:z`	+hh:mm numeric time zone (e.g., -04:00)
`%::z`	+hh:mm:ss numeric time zone (e.g., -04:00:00)
`%:::`	z numeric time zone with : to necessary precision (e.g., -04, +05:30)
`%Z`	alphabetic time zone abbreviation (e.g., EDT)

Table 2: Time formats

Functions

- `as.Date`
- `format`
- `as.POSIXct`
- `as.POSIXlt`

3.5 Special values

Next to regular "numbers", `R` also knows "special values":

Special values

`NA`	missing	`is.na(...)`
`NULL`	not defined	`is.null(...)`
`NaN`	not a number (numeric)	`is.nan(...)`
`Inf`	infinity (numeric)	`is.finite(...)`
		`is.infinite(...)`

`NaN` and `Inf`:

```
c(1, 1, 0)/c(1, 0, 0)

[1]   1 Inf NaN
```

Special values

`NA` and `NULL`:

```
length(c(1, 2, 3, NA, 5))

[1] 5

length(c(1, 2, 3, NULL, 5))

[1] 4
```

Use `NA` to denote *missings*, `NULL` to *delete*.

4 Working with a lot of data

4.1 Indexing and subsetting

Indexing

```
x <- c("07745", "Jena", "Kahlaische Straße", "10")
x

[1] "07745"             "Jena"              "Kahlaische Straße"
[4] "10"
```

We can now access a single element of `x`:

```
x[3]

[1] "Kahlaische Straße"
```

This also works with numbers:

```
x <- 21:30
x

 [1] 21 22 23 24 25 26 27 28 29 30

x[5]

[1] 25
```

Indexing

We can also address a range of elements:

```
x[5:7]

[1] 25 26 27
```

Such a range of elements need not be contiguous:

```
x[c(1, 5, 9)]

[1] 21 25 29
```

Indexing with logicals
We can index with either a vector of indices (which can be smaller or larger than the original vector), or with a vector of logicals of the same size of the original vector.

```
x <- c(3, 7, -5)
x[c(TRUE, FALSE, TRUE)]

[1]  3 -5

x[c(1, 3)]

[1]  3 -5
```

Indexing with logicals is interesting when the logical is actually a condition:

```
x > 0

[1]  TRUE  TRUE FALSE

x[x > 0]

[1] 3 7
```

4.2 Matrices of Numbers

The function `cbind` and `rbind` combine several vectors (of the same length) to form a matrix:

Matrices of numbers
Here is a 3×2 matrix. Note how the recycling rule works in the construction of this matrix.

```r
cbind(c(1, 2), c(40, 20, 30))

Warning: number of rows of result is not a multiple of vector length (arg
1)

     [,1] [,2]
[1,]    1   40
[2,]    2   20
[3,]    1   30
```

Here is a 2×3 matrix. Note again how the recycling rule works in the construction of this matrix.

```r
rbind(c(1, 2), c(10, 20, 30))

Warning: number of columns of result is not a multiple of vector length (arg
1)

     [,1] [,2] [,3]
[1,]    1    2    1
[2,]   10   20   30
```

What happens if we add x and y which have not the same dimension ?

Matrices of numbers

```r
x <- cbind(c(1, 2), c(40, 20, 30))

Warning: number of rows of result is not a multiple of vector length (arg
1)

y <- rbind(c(1, 2), c(10, 20, 30))

Warning: number of columns of result is not a multiple of vector length (arg
1)

x + y

Error: non-conformable arrays
```

We see that the recycling rule allows us to work with vectors of different length, but not with matrices of a different dimension.

Elements of a matrix

Here is the entire matrix again:

```
x
```

```
     [,1] [,2]
[1,]    1   40
[2,]    2   20
[3,]    1   30
```

Now we extract only the first row:

```
x[1, ]
```

```
[1]  1 40
```

Here we extract only the second column:

```
x[, 2]
```

```
[1] 40 20 30
```

Here are the first two rows:

```
x[c(1, 3), ]
```

```
     [,1] [,2]
[1,]    1   40
[2,]    1   30
```

One way to access elements (or rows or columns) is with indices. Alternatively we can use logicals. Only those elements where the index is `TRUE` are used. The following gives us the first and the third row:

```
x[c(TRUE, FALSE, TRUE), ]
```

```
     [,1] [,2]
[1,]    1   40
[2,]    1   30
```

This is a first step to extract a subset of a matrix on a certain condition. Here is the condition:

```
x[, 2] > 20
```

```
[1]  TRUE FALSE  TRUE
```

A subset

```
x[x[, 2] > 20, ]

     [,1] [,2]
[1,]    1   40
[2,]    1   30
```

Of course, the same can be done for data sets:

Elements of a data sets

```
Wages

     exp wks bluecol ind south smsa married    sex union ed black lwage
1      3  32      no   0   yes   no     yes   male    no  9    no 5.561
2      4  43      no   0   yes   no     yes   male    no  9    no 5.720
3      5  40      no   0   yes   no     yes   male    no  9    no 5.996
 [ reached getOption("max.print") -- omitted 4162 rows ]
```

```
Wages[1, ]

  exp wks bluecol ind south smsa married  sex union ed black lwage
1   3  32      no   0   yes   no     yes male    no  9    no 5.561
```

Elements of a data sets

```
Wages[, 1]

 [1]  3  4  5  6  7  8  9 30 31 32 33 34 35 36  6  7  8  9 10 11 12 31 32 33 34
[26] 35 36 37 10 11 12 13 14 15 16 26 27 28 29 30
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

If the rows or columns of a matrix or a data set have names, we can also use the names (instead of the index):

Columns of a data set by name

```
Wages

     exp wks bluecol ind south smsa married    sex union ed black lwage
1      3  32      no   0   yes   no     yes   male    no  9    no 5.561
2      4  43      no   0   yes   no     yes   male    no  9    no 5.720
3      5  40      no   0   yes   no     yes   male    no  9    no 5.996
 [ reached getOption("max.print") -- omitted 4162 rows ]
```

```r
Wages[, "exp"]
```

```
 [1]  3  4  5  6  7  8  9 30 31 32 33 34 35 36  6  7  8  9 10 11 12 31 32 33 34
[26] 35 36 37 10 11 12 13 14 15 16 26 27 28 29 30
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

The columns of a data set can also be addressed with $:

```r
Wages$exp
```

```
 [1]  3  4  5  6  7  8  9 30 31 32 33 34 35 36  6  7  8  9 10 11 12 31 32 33 34
[26] 35 36 37 10 11 12 13 14 15 16 26 27 28 29 30
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

In a similar way we can now specify *columns* and *rows* at the same time:

Elements of a data set

```r
Wages[1, "exp"]
```

```
[1] 3
```

```r
Wages$exp[1]
```

```
[1] 3
```

```r
Wages[c(1:3), "exp"]
```

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

```r
Wages[c(1:3, 15, 30:32), c("exp", "black", "lwage")]
```

```
   exp black lwage
1    3    no 5.561
2    4    no 5.720
3    5    no 5.996
15   6    no 5.652
30  11    no 6.620
31  12    no 6.633
32  13    no 6.983
```

Subsetting a data set

Let us have another look at `Wages`:

```
Wages[1:3, ]
```

```
  exp wks bluecol ind south smsa married  sex union ed black lwage
1   3  32      no   0   yes   no     yes male    no  9    no 5.561
2   4  43      no   0   yes   no     yes male    no  9    no 5.720
3   5  40      no   0   yes   no     yes male    no  9    no 5.996
```

Here is a condition that `exp` (the work experience) must be 1:

```
Wages$exp == 1
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

Here is the subset of `Wages` where this condition is `TRUE`:

```
Wages[Wages$exp == 1, ]
```

```
     exp wks bluecol ind south smsa married    sex union ed black lwage
624    1  48      no   0    no  yes      no female    no 12    no 6.215
1548   1  45     yes   0    no  yes      no   male    no 11    no 5.704
1919   1  30      no   0    no  yes     yes   male    no 17    no 6.215
 [ reached getOption("max.print") -- omitted 5 rows ]
```

Let us be more specific and only look at `exp==1` and `married="yes"`:

Subsets

```
Wages[Wages$exp == 1 & Wages$married == "yes", ]
```

```
     exp wks bluecol ind south smsa married  sex union ed black lwage
1919   1  30      no   0    no  yes     yes male    no 17    no 6.215
2878   1  50      no   0    no  yes     yes male    no 16    no 6.372
4110   1  51      no   0   yes   no     yes male    no 16    no 5.438
```

This can be written in a simpler way with `subset`:

```
subset(Wages, exp == 1 & married == "yes")
```

```
     exp wks bluecol ind south smsa married  sex union ed black lwage
1919   1  30      no   0    no  yes     yes male    no 17    no 6.215
2878   1  50      no   0    no  yes     yes male    no 16    no 6.372
4110   1  51      no   0   yes   no     yes male    no 16    no 5.438
```

We can subset rows and columns at the same time:

Subsets of rows and columns

```
Wages[Wages$exp == 1 & Wages$married == "yes", ][, c("sex", "black", "lwage")]
```

```
      sex black lwage
1919 male    no 6.215
2878 male    no 6.372
4110 male    no 5.438
```

```
subset(Wages, exp == 1 & married == "yes")[, c("sex", "black", "lwage")]
```

```
      sex black lwage
1919 male    no 6.215
2878 male    no 6.372
4110 male    no 5.438
```

4.3 Sorting

In R we usually sort with indices. `order` gives us a vector of indices:

Sorting with indices

```
order(Wages$lwage)
```

```
 [1] 3279 1163 1164 2551 3280  526 1277 3281 4145 1278 1450 3277  512 1058 1732
[16] 2521 3004 4146 4147 4148 1275  547  162 1723 3278 4110 1059 1060 1733  548
[31] 2522 3599 3600 1276  163  513  533  890 1165 1451
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

We can use this vector of indices to order a vector:

```
Wages$lwage[order(Wages$lwage)]
```

```
 [1] 4.605 5.011 5.011 5.017 5.081 5.165 5.193 5.273 5.298 5.347 5.394 5.394
[13] 5.416 5.416 5.416 5.416 5.416 5.416 5.416 5.416 5.421 5.429 5.438 5.438
[25] 5.438 5.438 5.460 5.460 5.464 5.472 5.481 5.501 5.501 5.517 5.521 5.521
[37] 5.521 5.521 5.521 5.521
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

We can use the same vector of indices to order the entire data frame:

```
Wages[order(Wages$lwage), ]
```

```
     exp wks bluecol ind south smsa married    sex union ed black lwage
3279  46  39     yes   0   yes  yes      no female    no  9   yes 4.605
1163  10  20     yes   0    no   no      no female    no 10    no 5.011
1164  11  30     yes   0    no   no      no female    no 10    no 5.011
 [ reached getOption("max.print") -- omitted 4162 rows ]
```

In the same way we can order according to several variables:

```
Wages[order(Wages$exp, Wages$lwage), ]
```

```
     exp wks bluecol ind south smsa married    sex union ed black lwage
4110   1  51      no   0   yes   no     yes   male    no 16    no 5.438
4159   1  52      no   0    no  yes      no female    no 12    no 5.687
1548   1  45     yes   0    no  yes      no   male    no 11    no 5.704
 [ reached getOption("max.print") -- omitted 4162 rows ]
```

Functions

- `cbind`, `rbind`, ...
- `dim`
- `subset`
- `order`
- `table`

4.4 Lists

Lists

Lists = a collection of (potentially) different types.

```
x <- list(3, "abc")
x

[[1]]
[1] 3

[[2]]
[1] "abc"
```

Elements of a list can be accessed with `[[ .. ]]`

```
x[[1]]

[1] 3

x[[2]]

[1] "abc"
```

Inspecting the structure of lists

Lists can be inspected with `str`:

```
str(x)

List of 2
 $ : num 3
 $ : chr "abc"
```

Lists with named elements

Elements of a list can (but need not) have names. We relate names to values with =

```
x <- list(value = 123, name = "abc")
x

$value
[1] 123

$name
[1] "abc"
```

When we inspect `x` with the command `str` we see that the elements have names.

```
str(x)

List of 2
 $ value: num 123
 $ name : chr "abc"
```

Accessing elements of lists

Elements of a list can be accessed by their position or by their name, with `[[ .. ]]` or `$`:

```
x[[1]]

[1] 123

x[["value"]]

[1] 123

x$value

[1] 123
```

More complex list¡s

Elements of a list can be any data type. In particular they can themselves be lists.

```
x <- list(x = 7, y = c("abc", "def"), z = cbind(1:2, 3:4))
str(x)

List of 3
 $ x: num 7
 $ y: chr [1:2] "abc" "def"
 $ z: int [1:2, 1:2] 1 2 3 4
```

`unlist` tries to make a vector out of a list.

```
unlist(x)

    x    y1    y2    z1    z2    z3    z4
  "7" "abc" "def"   "1"   "2"   "3"   "4"
```

Functions

- `str`

- `[[`
- `$`
- `unlist`

4.5 Summary datatypes

Summary data types

- Vector (1 dimension)
- Matrix (2 dimensions)
- Array (n dimensions)
- Data frame (2 dimensions, different types per column)
- List (not rectangular, different types per entry)

5 Reading and writing data

5.1 Using built in data sets

`date` makes a data set from a package accessible.

Using built in data sets

```
data(Wages, package = "Ecdat")
```

Now we can find out the names of the variables, the dimension, and statistics of some variables

```
names(Wages)

 [1] "exp"     "wks"     "bluecol" "ind"     "south"   "smsa"    "married"
 [8] "sex"     "union"   "ed"      "black"   "lwage"

dim(Wages)

[1] 4165   12

mean(Wages$exp)

[1] 19.85
```

`str` gives a brief overview of the types and the first few values of each variable in the dataset.

Using built in data sets

```
str(Wages)

'data.frame': 4165 obs. of  12 variables:
 $ exp    : int  3 4 5 6 7 8 9 30 31 32 ...
 $ wks    : int  32 43 40 39 42 35 32 34 27 33 ...
 $ bluecol: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 2 2 2 ...
 $ ind    : int  0 0 0 0 1 1 1 0 0 1 ...
 $ south  : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 1 1 1 ...
 $ smsa   : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
 $ married: Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
 $ sex    : Factor w/ 2 levels "female","male": 2 2 2 2 2 2 2 2 2 2 ...
 $ union  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 2 ...
 $ ed     : int  9 9 9 9 9 9 9 11 11 11 ...
 $ black  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
 $ lwage  : num  5.56 5.72 6 6 6.06 ...
```

summary gives some basic statistics:

Using built in data sets

```
summary(Wages)

      exp             wks         bluecol          ind          south        smsa
 Min.   : 1.0   Min.   : 5.0   no :2036   Min.   :0.000   no :2956   no :1442
 1st Qu.:11.0   1st Qu.:46.0   yes:2129   1st Qu.:0.000   yes:1209   yes:2723
 Median :18.0   Median :48.0              Median :0.000
 married        sex         union             ed          black          lwage
 no : 773   female: 469   no :2649   Min.   : 4.0   no :3864   Min.   :4.61
 yes:3392   male  :3696   yes:1516   1st Qu.:12.0   yes: 301   1st Qu.:6.39
                                     Median :12.0              Median :6.68
 [ reached getOption("max.print") -- omitted 3 rows ]
```

5.2 Reading and writing csv files

Writing csv files

Let us first export a data frame as csv:

```
write.csv(Wages, file = "wages.csv", row.names = FALSE)
```

The file wages.csv looks like this:

```
"exp","wks","bluecol","ind","south","smsa","married","sex","union","ed","black","lwage"
3,32,"no",0,"yes","no","yes","male","no",9,"no",5.56068
4,43,"no",0,"yes","no","yes","male","no",9,"no",5.72031
5,40,"no",0,"yes","no","yes","male","no",9,"no",5.99645
6,39,"no",0,"yes","no","yes","male","no",9,"no",5.99645
7,42,"no",1,"yes","no","yes","male","no",9,"no",6.06146
8,35,"no",1,"yes","no","yes","male","no",9,"no",6.17379
9,32,"no",1,"yes","no","yes","male","no",9,"no",6.24417
...
```

Reading csv files

Now let us read the file back into a variable:

```
w <- read.csv("wages.csv")
w

     exp wks bluecol ind south smsa married   sex union ed black lwage
1      3  32      no   0   yes   no     yes  male    no  9    no 5.561
2      4  43      no   0   yes   no     yes  male    no  9    no 5.720
3      5  40      no   0   yes   no     yes  male    no  9    no 5.996
 [ reached getOption("max.print") -- omitted 4162 rows ]
```

5.3 Reading z-Tree Output

Reading z-Tree Output

```
source("http://www.kirchkamp.de/lab/zTree.R")

Loading required package:  plyr
```

The function
`zTreeTables(...vector of filenames...[,vector of tables])`
reads zTree .xls files and returns a list of tables. Here we use `list.files` to find all files that match the typical z-Tree pattern. If we ever get more experiments our command will find them and use them.

```
setwd("rawdata/Trust/")
(files <- list.files(pattern = "[0-9]{6}_[0-9]{4}.xls", recursive = TRUE))

[1] "090722_0601.xls" "090722_0602.xls" "090722_0603.xls" "090722_0604.xls"

trustGS <- zTreeTables(files)
```

```
reading  090722_0601.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 090722_0602.xls is file 2 / 4 ***
reading  090722_0602.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 090722_0603.xls is file 3 / 4 ***
reading  090722_0603.xls ...
Skipping:
Doing: globals
Doing: subjects
*** 090722_0604.xls is file 4 / 4 ***
reading  090722_0604.xls ...
Skipping:
Doing: globals
Doing: subjects

setwd("../..")
```

The structure of the z-Tree object

```
str(trustGS)

List of 2
 $ globals :'data.frame': 24 obs. of  5 variables:
  ..$ Date           : chr [1:24] "090722_0601" "090722_0601" "090722_0601" "090722_0601"
  ..$ Treatment      : num [1:24] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Period         : num [1:24] 1 2 3 4 5 6 1 2 3 4 ...
  ..$ NumPeriods     : num [1:24] 6 6 6 6 6 6 6 6 6 6 ...
  ..$ RepeatTreatment: num [1:24] 0 0 0 0 0 0 0 0 0 0 ...
 $ subjects:'data.frame': 432 obs. of  14 variables:
  ..$ Date     : chr [1:432] "090722_0601" "090722_0601" "090722_0601" "090722_0601" ...
  ..$ Treatment: num [1:432] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Period   : num [1:432] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Subject  : num [1:432] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ Pos      : num [1:432] 2 2 1 2 1 1 2 1 2 2 ...
  ..$ Group    : num [1:432] 1 4 5 2 4 3 5 2 9 7 ...
  ..$ Offer    : num [1:432] 0 0 0.495 0 0.558 ...
  ..$ Receive  : num [1:432] 1.53 1.67 0 2.53 0 ...
  ..$ Return   : num [1:432] 0.586 1.132 0 1.471 0 ...
```

```
  ..$ GetBack  : num [1:432] 0 0 0.425 0 1.132 ...
  ..$ country  : num [1:432] 6 15 8 16 17 1 18 12 7 98 ...
  ..$ siblings : num [1:432] 1 3 3 3 0 0 3 1 2 3 ...
  ..$ sex      : num [1:432] 1 1 1 99 1 2 2 2 2 2 ...
  ..$ age      : num [1:432] 27 19 18 28 30 21 25 17 20 99 ...
```

Using the z-Tree object

As long as we need only a single table, we can access, e.g. the subjects table with `$subjects`.

If we need, e.g. the globals table together with the subjects table, we can merge them:

```
x <- with(trustGS, merge(globals, subjects))
str(x)

'data.frame': 432 obs. of  16 variables:
 $ Date           : chr  "090722_0601" "090722_0601" "090722_0601" "090722_0601" ...
 $ Treatment      : num  1 1 1 1 1 1 1 1 1 1 ...
 $ Period         : num  1 1 1 1 1 1 1 1 1 1 ...
 $ NumPeriods     : num  6 6 6 6 6 6 6 6 6 6 ...
 $ RepeatTreatment: num  0 0 0 0 0 0 0 0 0 0 ...
 $ Subject        : num  1 2 3 4 5 6 7 8 9 10 ...
 $ Pos            : num  2 2 1 2 1 1 2 1 2 2 ...
 $ Group          : num  1 4 5 2 4 3 5 2 9 7 ...
 $ Offer          : num  0 0 0.495 0 0.558 ...
 $ Receive        : num  1.53 1.67 0 2.53 0 ...
 $ Return         : num  0.586 1.132 0 1.471 0 ...
 $ GetBack        : num  0 0 0.425 0 1.132 ...
 $ country        : num  6 15 8 16 17 1 18 12 7 98 ...
 $ siblings       : num  1 3 3 3 0 0 3 1 2 3 ...
 $ sex            : num  1 1 1 99 1 2 2 2 2 2 ...
 $ age            : num  27 19 18 28 30 21 25 17 20 99 ...
```

5.4 Reading and writing R-Files

Reading and writing R-Files

If we want to save one or more R objects in a file, we use `save`

```
save(trustGS, zTreeTables, file = "120722_060x.Rdata")
```

To retrieve them, we use `load`

```r
load("120722_060x.Rdata")
```

Advantages:

- Rdata is very compact, files are small
- All attributes are saved together with the data
- We can save functions together with data

5.5 Reading and writing Stata files

Stata files

Let us first create a Stata file:

```r
library(foreign)
write.dta(Wages, file = "Wages.dta")
```

Now we do the following in Stata:

```
> pwd
> use Wages
> sum
> table ed
> save  Wages2
```

Reading back from Stata

```r
read.dta(file = "Wages2.dta")

    exp wks bluecol ind south smsa married   sex union ed black lwage
1     3  32      no   0   yes   no     yes  male    no  9    no 5.561
2     4  43      no   0   yes   no     yes  male    no  9    no 5.720
3     5  40      no   0   yes   no     yes  male    no  9    no 5.996
 [ reached getOption("max.print") -- omitted 4162 rows ]

Wa2 <- read.dta(file = "Wages2.dta")
summary(Wa2$ed)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0    12.0    12.0    12.8    16.0    17.0

summary(Wages$ed)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0    12.0    12.0    12.8    16.0    17.0
```

```
table(Wa2$ed)


   4    5    6    7    8    9   10   11   12   13   14   15   16   17
  14   21   28   77  175  161  231  210 1498  182  343   84  637  504

table(Wages$ed)


   4    5    6    7    8    9   10   11   12   13   14   15   16   17
  14   21   28   77  175  161  231  210 1498  182  343   84  637  504

summary(Wa2[, c("ed", "exp", "sex")])

       ed             exp            sex
 Min.   : 4.0   Min.   : 1.0   female: 469
 1st Qu.:12.0   1st Qu.:11.0   male  :3696
 Median :12.0   Median :18.0
 Mean   :12.8   Mean   :19.9
 3rd Qu.:16.0   3rd Qu.:29.0
 Max.   :17.0   Max.   :51.0
```

The two data frames (`Wa2` and `Wages`) seem to be the same (as they should).

5.6 Other formats

foreign

The `foreign` package contains functions to read and (often) write the following:

	read	write
S3	r	-
SAS	r	-
ARFF	r	w
DBF	r	w
Stata Files	r	w
Epi	r	-
Minitab	r	-
Octave	r	-
SPSS	r	w
Systat	r	-

gnumeric

`read.gnumeric.sheet` from the `gnumeric` package reads the following:

Applix	*.as
Data Interchange Format	*.dif
Gnumeric XML	*.gnumeric
GNU Oleo	*.oleo
HTML	*.html, *.htm
Linear and integer program expression	*.mps
Lotus 123	*.wk1, *.wks, *.123
MS Excel	*.xls, *.xlsx
MultiPlan	*.slk
Open Document Format	*.sxc, *.ods
Plan Perfect Format	*.pln
Quattro Pro	*.wb1, *.wb2, *.wb3
xspread	*.sc
Xbase	*.dbf

Reading from the clipboard
We can also read data from the clipboard (e.g. select some data in a spreadsheet and use this in R. However, this workflow will make it impossible to document what exactly *was* on the clipboard. It is always better to really import the spreadsheet

```
read.table("clipboard")
read.table("clipboard", header = TRUE)
```

Functions

- `data`, `names`, `rownames`, `colnames`
- `read.csv`, `write.csv`, `read.table`, `write.table`
- `read.dta`, `write.dta`, `zTreeTables`
- `save`, `save.image`, `load`
- `setwd`, `getwd`

5.7 Which data frame is actually used?

When use names of variables, R searches this name in different contexts.

Which data frame is actually used?

- `Wages$exp`
- `(...,data=Wages,...)`
- `attach(Wages),...,detach(Wages)`

- `with(Wages,...)`
- `within(Wages,...)`

Which data frame is actually used

- We can always explicitely prefix our variables with the name of the data frame. However, this can be a bit clumsy and is not necessary if we want to use always the same data frame.

  ```
  lm(Wages$lwage ~ Wages$exp + Wages$sex + Wages$ed)
  ```

- Many commands have a parameter `data` which can be used to specify a data frame.

  ```
  lm(lwage ~ exp + sex + ed,data=Wages)
  ```

- A data frame can also be brought into the current context with `attach`. So, after `attach(Wages)` we can simply say `exp` and this variable will first be searched in the dataset `Wages`. Note: Changing `exp` (or any other variable) does only change in the local context, not in the original data frame.

  ```
  attach(Wages)
  lm(lwage ~ exp + sex + ed)
  ...
  detach(Wages)
  ```

- We can surround a command with `with(`≪*data frame*≫`,`≪*command*≫ `)`. Then ≪*command*≫ is executed within the context of the data frame.

  ```
  with(Wages,lm(lwage ~ exp + sex + ed))
  ```

- `within(...)` solves a different problem: It performs (similar to `with(...)`) all calculations within the context of a given data frame, and then returns this data frame with all changes that have been caculated and with all new variables that have been generated.

  ```
  Wages2 <- within(Wages,{wage   <- exp(lwage);
                          lwage  <- NULL;
                          female <- sex=="female";
                          wage[female] <- wage[female]*2;
                         })
  ```

  ...creates a copy of `Wages` with two more added (`wage` and `female`), one removed (`lwage`), and the `wage` of female workers multiplied by 2.

Functions

- `as.data.frame`, `is.data.frame`
- `with`
- `within`
- `attach`, `detach`

6 Regressions

6.1 Regression objects

A simple regression

The function `lm` estimates OLS models. We use the *formula* notation to specify the estimated model.

```
lm(lwage ~ exp + sex + ed, data = Wages)


Call:
lm(formula = lwage ~ exp + sex + ed, data = Wages)

Coefficients:
(Intercept)          exp      sexmale           ed
     5.0876       0.0118       0.4358       0.0753
```

We can write the result of a regression into a variable:

```
est <- lm(lwage ~ exp + sex + ed, data = Wages)
```

This variable has not the class `lm` (linear model).

```
class(est)

[1] "lm"
```

Regression objects - structure

Actually, the regression object is a long list. The elements of this list are the estimated coefficients, the estimated residuals, etc.

```
str(est)
```

```
List of 13
 $ coefficients : Named num [1:4] 5.0876 0.0118 0.4358 0.0753
  ..- attr(*, "names")= chr [1:4] "(Intercept)" "exp" "sexmale" "ed"
 $ residuals    : Named num [1:4165] -0.676 -0.528 -0.264 -0.276 -0.222 ...
  ..- attr(*, "names")= chr [1:4165] "1" "2" "3" "4" ...
 $ effects      : Named num [1:4165] -430.87 6.234 -9.143 13.217 -0.195 ...
  ..- attr(*, "names")= chr [1:4165] "(Intercept)" "exp" "sexmale" "ed" ...
 $ rank         : int 4
 $ fitted.values: Named num [1:4165] 6.24 6.25 6.26 6.27 6.28 ...
  ..- attr(*, "names")= chr [1:4165] "1" "2" "3" "4" ...
 $ assign       : int [1:4] 0 1 2 3
 $ qr           :List of 5
  ..$ qr   : num [1:4165, 1:4] -64.5368 0.0155 0.0155 0.0155 0.0155 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:4165] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:4] "(Intercept)" "exp" "sexmale" "ed"
  .. ..- attr(*, "assign")= int [1:4] 0 1 2 3
  .. ..- attr(*, "contrasts")=List of 1
  .. .. ..$ sex: chr "contr.treatment"
  ..$ qraux: num [1:4] 1.02 1.02 1.01 1.03
  ..$ pivot: int [1:4] 1 2 3 4
  ..$ tol  : num 1e-07
  ..$ rank : int 4
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 4161
 $ contrasts    :List of 1
  ..$ sex: chr "contr.treatment"
 $ xlevels      :List of 1
  ..$ sex: chr [1:2] "female" "male"
 $ call         : language lm(formula = lwage ~ exp + sex + ed, data = Wages)
 $ terms        :Classes 'terms', 'formula' length 3 lwage ~ exp + sex + ed
  .. ..- attr(*, "variables")= language list(lwage, exp, sex, ed)
  .. ..- attr(*, "factors")= int [1:4, 1:3] 0 1 0 0 0 0 1 0 0 0 ...
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:4] "lwage" "exp" "sex" "ed"
  .. .. .. ..$ : chr [1:3] "exp" "sex" "ed"
  .. ..- attr(*, "term.labels")= chr [1:3] "exp" "sex" "ed"
  .. ..- attr(*, "order")= int [1:3] 1 1 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(lwage, exp, sex, ed)
  .. ..- attr(*, "dataClasses")= Named chr [1:4] "numeric" "numeric" "factor" "numeric"
```

```
 .. .. ..- attr(*, "names")= chr [1:4] "lwage" "exp" "sex" "ed"
$ model        :'data.frame': 4165 obs. of  4 variables:
 ..$ lwage: num [1:4165] 5.56 5.72 6 6 6.06 ...
 ..$ exp  : int [1:4165] 3 4 5 6 7 8 9 30 31 32 ...
 ..$ sex  : Factor w/ 2 levels "female","male": 2 2 2 2 2 2 2 2 2 2 ...
 ..$ ed   : int [1:4165] 9 9 9 9 9 9 9 11 11 11 ...
 ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 lwage ~ exp + sex + ed
 .. .. ..- attr(*, "variables")= language list(lwage, exp, sex, ed)
 .. .. ..- attr(*, "factors")= int [1:4, 1:3] 0 1 0 0 0 0 1 0 0 0 ...
 .. .. .. ..- attr(*, "dimnames")=List of 2
 .. .. .. .. ..$ : chr [1:4] "lwage" "exp" "sex" "ed"
 .. .. .. .. ..$ : chr [1:3] "exp" "sex" "ed"
 .. .. ..- attr(*, "term.labels")= chr [1:3] "exp" "sex" "ed"
 .. .. ..- attr(*, "order")= int [1:3] 1 1 1
 .. .. ..- attr(*, "intercept")= int 1
 .. .. ..- attr(*, "response")= int 1
 .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
 .. .. ..- attr(*, "predvars")= language list(lwage, exp, sex, ed)
 .. .. ..- attr(*, "dataClasses")= Named chr [1:4] "numeric" "numeric" "factor" "numeri
 .. .. .. ..- attr(*, "names")= chr [1:4] "lwage" "exp" "sex" "ed"
- attr(*, "class")= chr "lm"
```

6.2 What to do with a regression object?

Regression objects

We can do different things with such a variable (a regression object):

- `str(est)`
- `coef(est)`
- `logLik(est)`
- `AIC(est)`
- `summary(est)`
- `plot(est)`
- $\vdots$

Regression objects - summary

`summary` produces an overview with some statistics of the regression object.

```r
summary(est)
```

```
Call:
lm(formula = lwage ~ exp + sex + ed, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max 
-2.1883 -0.2370  0.0042  0.2521  1.9558 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 5.087573   0.035488   143.4   <2e-16 ***
exp         0.011828   0.000548    21.6   <2e-16 ***
sexmale     0.435815   0.018538    23.5   <2e-16 ***
ed          0.075295   0.002145    35.1   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.376 on 4161 degrees of freedom
Multiple R-squared:  0.335,Adjusted R-squared:  0.335 
F-statistic:  699 on 3 and 4161 DF,  p-value: <2e-16
```

Regression and heteroscedasticity

The core of the above table can be shown with `coeftest`.

```r
coeftest(est)
```

```
t test of coefficients:

            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 5.087573   0.035488   143.4   <2e-16 ***
exp         0.011828   0.000548    21.6   <2e-16 ***
sexmale     0.435815   0.018538    23.5   <2e-16 ***
ed          0.075295   0.002145    35.1   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Regression and heteroscedasticity

As a default R assumes homoscedasticity. For the heteroscedastic case we specify `vcov=hccm`.

```
coeftest(est, vcov = hccm)


t test of coefficients:

            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 5.087573   0.037168   136.9   <2e-16 ***
exp         0.011828   0.000618    19.1   <2e-16 ***
sexmale     0.435815   0.017543    24.8   <2e-16 ***
ed          0.075295   0.002277    33.1   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Regression objects - confidence intervals

Different from Stata, confidence intervals are not included in the summary. We calculate them separately.

```
confint(est)

              2.5 % 97.5 %
(Intercept) 5.01800 5.1571
exp         0.01075 0.0129
sexmale     0.39947 0.4722
ed          0.07109 0.0795
```

However, if we need the results side by side, we can do this:

```
cbind(coeftest(est), confint(est))

            Estimate Std. Error t value   Pr(>|t|)   2.5 % 97.5 %
(Intercept)  5.08757  0.0354883  143.36  0.000e+00 5.01800 5.1571
exp          0.01183  0.0005476   21.60  3.691e-98 0.01075 0.0129
sexmale      0.43582  0.0185376   23.51 7.340e-115 0.39947 0.4722
ed           0.07530  0.0021448   35.11 1.022e-236 0.07109 0.0795
```

Regression objects - coefficients and other statistics

Here are some *extractor-functions* that extract statistics from the regression object.

```
coef(est)

(Intercept)         exp     sexmale          ed 
    5.08757     0.01183     0.43582     0.07530 
```

```r
logLik(est)
```

```
'log Lik.' -1839 (df=5)
```

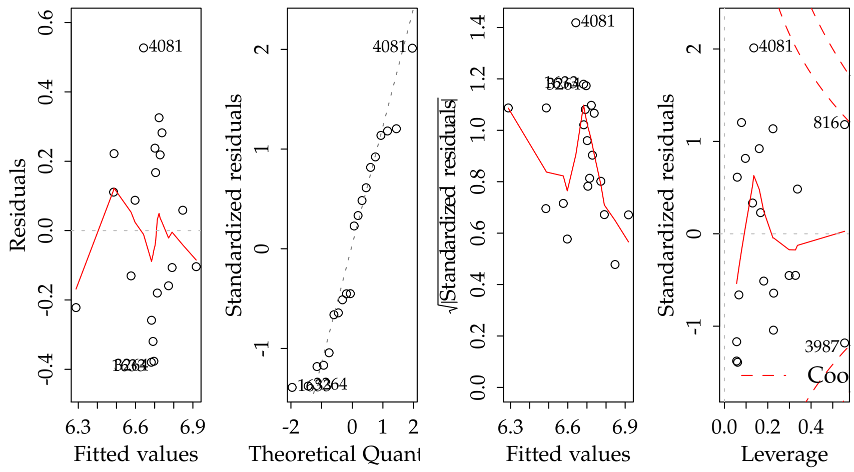
```r
AIC(est)
```

```
[1] 3688
```

Regression objects - plots

If we simply `plot` a regression object we get four diagnostic plots.

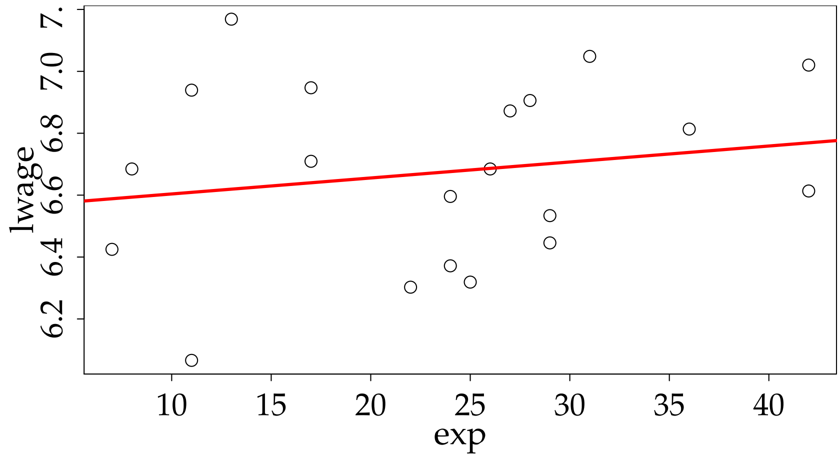
```r
plot(est)
```



Regression objects - lines

We can also draw the regression as a line:

```r
est2 <- lm(lwage ~ exp, data = w)
plot(lwage ~ exp, data = w)
abline(est2, col = "red", lwd = 3)
```



6.3 Hypothesis tests

Regression objects - hypothesis tests

```
library(car)
```

The function `linearHypothesis` from the package `car` allows us to write linear restrictions in a simple way.

```
linearHypothesis(est, "6*exp=ed")

Linear hypothesis test

Hypothesis:
6 exp - ed = 0

Model 1: restricted model
Model 2: lwage ~ exp + sex + ed

  Res.Df  RSS Df Sum of Sq    F Pr(>F)
1     17 1.27
2     16 1.27  1   0.00238 0.03   0.86
```

Again, this can also be done for the heteroscedastic case.

```
linearHypothesis(est, "6*exp=ed", vcov = hccm)

Linear hypothesis test

Hypothesis:
```

```
6 exp - ed = 0

Model 1: restricted model
Model 2: lwage ~ exp + sex + ed

Note: Coefficient covariance matrix supplied.

  Res.Df Df    F Pr(>F)
1     17
2     16  1 0.03   0.87
```

Regression objects - hypothesis tests
A Breusch-Pagan test against heteroskedasticity.

```
bptest(est)


studentized Breusch-Pagan test

data:  est
BP = 1.539, df = 3, p-value = 0.6734
```

6.4 Regression with dummies

Dummies
Dummy variables will be automagially coerced into integers. R will also create a variable name the shows the meaning (here `blackyes`).

```
summary(lm(lwage ~ exp + black, data = Wages))


Call:
lm(formula = lwage ~ exp + black, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max
-1.9825 -0.2872  0.0218  0.2737  1.9512

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  6.520275   0.014237   458.0   <2e-16 ***
exp          0.009147   0.000625    14.6   <2e-16 ***
blackyes    -0.353412   0.026474   -13.3   <2e-16 ***
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.442 on 4162 degrees of freedom
Multiple R-squared:  0.0831,Adjusted R-squared:  0.0826
F-statistic:  189 on 2 and 4162 DF,  p-value: <2e-16
```

Interactions
Interactions can be specified as just the product (*) of several terms. By default R will add *all* possible interactions to the model.

```
summary(lm(lwage ~ exp * black, data = Wages))


Call:
lm(formula = lwage ~ exp * black, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.8917 -0.2860  0.0222  0.2745  1.9559 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept)   6.513979   0.014688  443.50  < 2e-16 ***
exp           0.009467   0.000651   14.53  < 2e-16 ***
blackyes     -0.267818   0.055908   -4.79  1.7e-06 ***
exp:blackyes -0.004017   0.002311   -1.74    0.082 .  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.442 on 4161 degrees of freedom
Multiple R-squared:  0.0838,Adjusted R-squared:  0.0831
F-statistic:  127 on 3 and 4161 DF,  p-value: <2e-16
```

Specific interactions
Here we include only some interactions with the operator :

```
summary(lm(lwage ~ exp + black + sex + black:sex:exp, data = Wages))


Call:
lm(formula = lwage ~ exp + black + sex + black:sex:exp, data = Wages)
```

```
Residuals:
   Min     1Q Median     3Q    Max
-1.830 -0.264  0.017  0.261  1.895

Coefficients: (1 not defined because of singularities)
                      Estimate Std. Error t value Pr(>|t|)
(Intercept)            6.13105    0.04081  150.22  < 2e-16 ***
exp                    0.01002    0.00244    4.11  4.1e-05 ***
blackyes              -0.24434    0.05482   -4.46  8.5e-06 ***
sexmale                0.45183    0.04274   10.57  < 2e-16 ***
exp:blackno:sexfemale  0.00251    0.00369    0.68     0.50
exp:blackyes:sexfemale -0.00399   0.00254   -1.57     0.12
exp:blackno:sexmale   -0.00223    0.00249   -0.90     0.37
exp:blackyes:sexmale        NA         NA      NA       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.424 on 4158 degrees of freedom
Multiple R-squared:  0.159,Adjusted R-squared:  0.158
F-statistic:  131 on 6 and 4158 DF,  p-value: <2e-16
```

Factors in the formula

Factors with more than only two values will be included in the model as one dummy for each value of the factor.

```r
summary(lm(lwage ~ as.factor(ed) + black, data = Wages))
```

```

Call:
lm(formula = lwage ~ as.factor(ed) + black, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max
-1.9735 -0.2722  0.0039  0.2794  1.9142

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)       6.0381     0.1116   54.11  < 2e-16 ***
as.factor(ed)5    0.3981     0.1441    2.76  0.00575 **
as.factor(ed)6    0.5183     0.1373    3.78  0.00016 ***
as.factor(ed)7    0.2536     0.1213    2.09  0.03666 *
as.factor(ed)8    0.4016     0.1160    3.46  0.00054 ***
```

```
as.factor(ed)9    0.4463     0.1163    3.84  0.00013 ***
as.factor(ed)10   0.4780     0.1149    4.16  3.3e-05 ***
as.factor(ed)11   0.5347     0.1154    4.64  3.7e-06 ***
 [ reached getOption("max.print") -- omitted 7 rows ]
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.418 on 4150 degrees of freedom
Multiple R-squared:  0.184,Adjusted R-squared:  0.181
F-statistic: 66.9 on 14 and 4150 DF,  p-value: <2e-16
```

Remove the regression constant with `- 1`

To estimate the model without a constant we add `- 1` to the formula.

```
summary(lm(lwage ~ ed + black - 1, data = Wages))


Call:
lm(formula = lwage ~ ed + black - 1, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max
-1.9367 -0.2716  0.0106  0.2747  1.8126

Coefficients:
         Estimate Std. Error t value Pr(>|t|)
ed        0.06234    0.00235    26.6   <2e-16 ***
blackno   5.89412    0.03110   189.5   <2e-16 ***
blackyes  5.63662    0.03648   154.5   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.419 on 4162 degrees of freedom
Multiple R-squared:  0.996,Adjusted R-squared:  0.996
F-statistic: 3.53e+05 on 3 and 4162 DF,  p-value: <2e-16
```

Using `update`

The same could have been achieved with `update`:

```
est <- lm(lwage ~ ed + black, data = Wages)
update(est, . ~ . - 1)
```

```
Call:
lm(formula = lwage ~ ed + black - 1, data = Wages)

Coefficients:
      ed   blackno   blackyes
  0.0623    5.8941     5.6366
```

We can as well add variables (here we add `exp`):

```
update(est, . ~ . + exp)


Call:
lm(formula = lwage ~ ed + black + exp, data = Wages)

Coefficients:
(Intercept)           ed     blackyes          exp
     5.4900       0.0735      -0.2656       0.0131
```

6.5 Nonlinear models

A log-log model

Transformations of variables can be done in the formula:

```
summary(lm(lwage ~ ed + log(ed) + black, data = Wages))


Call:
lm(formula = lwage ~ ed + log(ed) + black, data = Wages)

Residuals:
    Min      1Q  Median      3Q     Max
-1.9602 -0.2738  0.0032  0.2815  1.8261

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.6454     0.2145   30.98  < 2e-16 ***
ed            0.1067     0.0127    8.37  < 2e-16 ***
log(ed)      -0.5228     0.1477   -3.54  0.00041 ***
blackyes     -0.2548     0.0252  -10.10  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.419 on 4161 degrees of freedom
Multiple R-squared:  0.178,Adjusted R-squared:  0.178
F-statistic:  301 on 3 and 4161 DF,  p-value: <2e-16
```

A logit model

Here is an example for logistic regression:
The Boston HDMA Data Set:

- `dir`: debt to income ratio
- `hir`: housing expenses to income ratio
- `self`: self employed
- `deny`: mortgage applicationd denied

```
data(Hdma, package = "Ecdat")
summary(Hdma[, c("dir", "hir", "self", "deny")])

      dir              hir             self         deny
 Min.   :0.000   Min.   :0.000   no  :2103   no :2096
 1st Qu.:0.280   1st Qu.:0.214   yes : 277   yes: 285
 Median :0.330   Median :0.260   NA's:   1
 Mean   :0.331   Mean   :0.255
 3rd Qu.:0.370   3rd Qu.:0.299
 Max.   :3.000   Max.   :3.000
```

A logit model

The estimation is done with `glm`. The parameter `family` specifies the "family" of the model (binomial, gaussian, Gamma, inverse.gaussian, poisson, quasi, quasibinomial, quasipoisson). Hence, `glm` covers a range of estimation problems.

```
glm(deny ~ dir + hir + self, family = binomial, data = Hdma)


Call:  glm(formula = deny ~ dir + hir + self, family = binomial, data = Hdma)

Coefficients:
(Intercept)          dir          hir      selfyes
     -4.017        6.082       -0.466        0.346

Degrees of Freedom: 2379 Total (i.e. Null);  2376 Residual
  (1 observation deleted due to missingness)
Null Deviance:      1740
Residual Deviance: 1660  AIC: 1660
```

The logit model object

Similar to lm, also glm creates an object that we can write into a variable and that we can summarise.

```
est <- glm(deny ~ dir + hir + self, family = binomial, data = Hdma)
summary(est)


Call:
glm(formula = deny ~ dir + hir + self, family = binomial, data = Hdma)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-2.755  -0.525  -0.463  -0.388   2.761

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -4.017      0.281  -14.29  < 2e-16 ***
dir            6.082      0.916    6.64  3.1e-11 ***
hir           -0.466      1.039   -0.45    0.654
selfyes        0.346      0.188    1.84    0.066 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1744.2  on 2379  degrees of freedom
Residual deviance: 1656.6  on 2376  degrees of freedom
  (1 observation deleted due to missingness)
AIC: 1665

Number of Fisher Scoring iterations: 5
```

The logit model object and heteroscedasticity

To see estimated standard errors for the heteroscedastic case we use vcov=vcovHC from the sandwich package.

```
library(sandwich)
coeftest(est, vcov = vcovHC)


z test of coefficients:
```

```
             Estimate Std. Error z value Pr(>|z|)
(Intercept)   -4.017      0.378  -10.62   <2e-16 ***
dir            6.082      1.128    5.39    7e-08 ***
hir           -0.466      1.075   -0.43    0.665
selfyes        0.346      0.196    1.76    0.078 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Linear hypotheses

Linear hypotheses can be specified as in `lm`.

```
linearHypothesis(est, "dir=hir")

Linear hypothesis test

Hypothesis:
dir - hir = 0

Model 1: restricted model
Model 2: deny ~ dir + hir + self

  Res.Df Df Chisq Pr(>Chisq)
1   2377
2   2376  1    14    0.00018 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Linear hypotheses and heteroscedasticity

```
linearHypothesis(est, "dir=hir", vcov = vcovHC)

Linear hypothesis test

Hypothesis:
dir - hir = 0

Model 1: restricted model
Model 2: deny ~ dir + hir + self

Note: Coefficient covariance matrix supplied.

  Res.Df Df Chisq Pr(>Chisq)
```

```
1   2377
2   2376  1  12.2    0.00049 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Maximum likelihood manually

Although R comes with a wide range of ML estimators, we can, for special estimators, do our own likelihood maximisation. Here is the logistic model as an example:

$$\ln L = \sum_i \left(y_i \ln F(\boldsymbol{x}'\boldsymbol{\beta}) + (1-y_i)\ln\left(1-F(\boldsymbol{x}'\boldsymbol{\beta})\right)\right)$$

The logistic regression above can also be done "manually" as follows.

`optim` minimises just any function, here the function that we specify: `logli`. (this is the negative of the above function, since we want to maximise but `optim` rather minimises).

```
logli<-function(beta) {
  n<-length(x)
  theta<-x*beta[2]+rep(1,n)*beta[1]
  -(sum(y*(plogis(theta,log.p=TRUE)))+
    sum((1-y)*plogis(theta,lower.tail=FALSE,log.p=TRUE)))
}
x <- 1:8
y <- c(0,1,0,0,1,1,0,1)
optim(c(0,0),logli)

$par
[1] -1.3759  0.3057

$value
[1] 5.102

$counts
function gradient
      63       NA

$convergence
[1] 0

$message
NULL
```

`glm` obtains the same result:

```
glm(y ~ x, family = binomial)


Call:  glm(formula = y ~ x, family = binomial)

Coefficients:
(Intercept)            x
     -1.376        0.306

Degrees of Freedom: 7 Total (i.e. Null);  6 Residual
Null Deviance:      11.1
Residual Deviance: 10.2  AIC: 14.2
```

Count data

Here is the poisson model with doctors visits:

```
data(DoctorAUS, package = "Ecdat")
summary(glm(doctorco ~ sex + age + I(age^2) + income, family = poisson, data = DoctorAUS)


Call:
glm(formula = doctorco ~ sex + age + I(age^2) + income, family = poisson,
    data = DoctorAUS)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-1.020  -0.815  -0.672  -0.597   6.313

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -1.9777     0.1841  -10.74  < 2e-16 ***
sex           0.2146     0.0559    3.84  0.00012 ***
age           2.8565     0.9893    2.89  0.00388 **
I(age^2)     -1.7900     1.0847   -1.65  0.09889 .
income       -0.3207     0.0837   -3.83  0.00013 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 5634.8  on 5189  degrees of freedom
Residual deviance: 5432.2  on 5185  degrees of freedom
AIC: 7774
```

```
Number of Fisher Scoring iterations: 6
```

Count data with overdispersion

Now we do the same exercise with negative binomial

```
library(MASS)
summary(glm.nb(doctorco ~ sex + age + I(age^2) + income, data = DoctorAUS))


Call:
glm.nb(formula = doctorco ~ sex + age + I(age^2) + income, data = DoctorAUS,
    init.theta = 0.4281145934, link = log)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-0.827  -0.706  -0.601  -0.543   3.573

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -2.0252     0.2399   -8.44  < 2e-16 ***
sex           0.2446     0.0726    3.37  0.00076 ***
age           2.9328     1.3251    2.21  0.02687 *
I(age^2)     -1.8549     1.4656   -1.27  0.20566
income       -0.2985     0.1076   -2.77  0.00554 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Negative Binomial(0.4281) family taken to be 1)

    Null deviance: 3110.1  on 5189  degrees of freedom
Residual deviance: 2989.9  on 5185  degrees of freedom
AIC: 7067

Number of Fisher Scoring iterations: 1


              Theta:  0.4281
          Std. Err.:  0.0313

 2 x log-likelihood:  -7054.9350
```

Ordered probit

This is an ordered probit model:

```
data(Mathlevel, package = "Ecdat")
summary(polr(mathlevel ~ sat + sex + mathcourse + major, data = Mathlevel, method = "prob

Re-fitting to get Hessian

Call:
polr(formula = mathlevel ~ sat + sex + mathcourse + major, data = Mathlevel,
    method = "probit")

Coefficients:
              Value Std. Error t value
sat         0.00526   0.000811    6.48
sexfemale   0.21143   0.093070    2.27
mathcourse  0.56334   0.077317    7.29
majoreco   -0.11311   0.121649   -0.93
majoross   -0.28797   0.144411   -1.99
majorns     0.44182   0.135447    3.26
majorhum   -0.45335   0.198786   -2.28

Intercepts:
          Value  Std. Error t value
170|171a   3.420  0.508      6.738
171a|172a  3.687  0.509      7.240
172a|171b  3.743  0.510      7.345
171b|172b  4.910  0.521      9.431
172b|221a  5.184  0.523      9.908
221a|221b  5.423  0.525     10.322

Residual Deviance: 1792.06
AIC: 1818.06
```

Multinomial

This is an example for a multinomial model:

```
data(Mode, package = "Ecdat")
summary(Mode)

     choice         cost.car        cost.carpool        cost.bus         cost.rail
 car     :218   Min.   :0.41   Min.   :0.129   Min.   :1.01   Min.   :1.27
 carpool: 32   1st Qu.:3.70   1st Qu.:0.952   1st Qu.:1.78   1st Qu.:1.95
 bus     : 81   Median :4.88   Median :1.666   Median :2.03   Median :2.20
 rail    :122   Mean   :4.87   Mean   :1.686   Mean   :2.04   Mean   :2.21
```

```
   time.car      time.carpool       time.bus        time.rail    
 Min.    : 2.4   Min.    : 8.39   Min.    : 1.97   Min.    : 4.62  
 1st Qu.:21.8   1st Qu.:28.39   1st Qu.:25.46   1st Qu.:28.14  
 Median :37.5   Median :40.64   Median :41.41   Median :40.03  
 Mean   :37.0   Mean   :39.77   Mean   :39.92   Mean   :39.50  
 [ reached getOption("max.print") -- omitted 2 rows ]
```

Multinomial

```
multinom(choice ~ cost.car + cost.carpool + cost.bus + cost.rail + time.car + time.carpo
    time.bus + time.rail, data = Mode)

# weights:  40 (27 variable)
initial  value 627.991346 
iter  10 value 394.826983
iter  20 value 360.160469
iter  30 value 342.077240
final  value 342.073501 
converged
Call:
multinom(formula = choice ~ cost.car + cost.carpool + cost.bus + 
    cost.rail + time.car + time.carpool + time.bus + time.rail, 
    data = Mode)

Coefficients:
        (Intercept) cost.car cost.carpool cost.bus cost.rail time.car
carpool      -4.106   0.6361      -0.4473  0.04506   -0.5501  0.12367
bus          -4.789   0.8461       0.2163  0.01013   -0.5277  0.02285
rail         -4.300   0.8901       0.2058  0.56601   -1.2753  0.03639
        time.carpool  time.bus time.rail
carpool     -0.06923  0.007442 -0.027733
bus          0.09462 -0.107751 -0.006393
rail         0.07297 -0.018132 -0.075283

Residual Deviance: 684.1 
AIC: 738.1 
```

Interval regression

In this example for an interval regression the dependent variable `stobacco` is censored. Many observations are zero. We can imagine that the underlying latent variable could be negative in these cases.

```r
data(Tobacco, package = "Ecdat")
summary(Tobacco[, c("stobacco", "nkids", "nkids2", "lnx")])
```

```
    stobacco            nkids            nkids2             lnx
 Min.   :0.0000   Min.   :0.000   Min.   :0.0000   Min.   :11.8
 1st Qu.:0.0000   1st Qu.:0.000   1st Qu.:0.0000   1st Qu.:13.4
 Median :0.0000   Median :0.000   Median :0.0000   Median :13.8
 Mean   :0.0122   Mean   :0.565   Mean   :0.0448   Mean   :13.7
 3rd Qu.:0.0138   3rd Qu.:1.000   3rd Qu.:0.0000   3rd Qu.:14.1
 Max.   :0.1928   Max.   :5.000   Max.   :2.0000   Max.   :15.3
```

- `stobacco`: budgetshare of tobacco
- `nkids`: number of kids of more than two years old
- `nkids2`: number of kids of less than two years old
- `lnx`: log of total expenditures

Interval regression

For each observation we define an interval. The right boundary of the interval is `stobacco`. The left boudary is the same value for the uncensored observations. For the censored observations the left boundary is $-$ which is coded as `NA`.

```r
library(survival)
Tobacco <- within(Tobacco, stobaccoMin <- ifelse(stobacco == 0, NA, stobacco))
summary(survreg(Surv(stobaccoMin, stobacco, type = "interval2") ~ nkids + nkids2 +
    lnx, dist = "gaussian", data = Tobacco))
```

```


Call:
survreg(formula = Surv(stobaccoMin, stobacco, type = "interval2") ~
    nkids + nkids2 + lnx, data = Tobacco, dist = "gaussian")
               Value Std. Error       z        p
(Intercept)  0.24887    0.03291    7.56 3.95e-14
nkids        0.00642    0.00120    5.35 8.79e-08
nkids2      -0.00671    0.00542   -1.24 2.15e-01
lnx         -0.01958    0.00242   -8.09 5.75e-16
Log(scale)  -3.01205    0.02474 -121.76 0.00e+00

Scale= 0.0492

Gaussian distribution
Loglik(model)= 711.8   Loglik(intercept only)= 673.6
```

```
Chisq= 76.35 on 3 degrees of freedom, p= 2.2e-16
Number of Newton-Raphson Iterations: 3
n= 2724
```

7 Functions

7.1 Writing functions

Writing functions

So far assign numbers to variables:

```
x <- 2
class(x)

[1] "numeric"
```

Functions are just a datatype. As such they are just assigned to a variable:

```
x <- function(z) {
    z^2
}
class(x)

[1] "function"
```

The value of functions

```
x

function(z) {
    z^2
}
```

Of course, we can simply use the function:

```
x(10)

[1] 100
```

7.2 More on parameters

Positional and named parameters

```
z <- function(a, b = 3) {
    a/b
}
```

We can call z with named parameters:

```
z(a = 10, b = 2)

[1] 5
```

With named parameters the order does not matter:

```
z(b = 2, a = 10)

[1] 5
```

If names are omitted, the order matters:

```
z(10, 2)

[1] 5
```

Parameters with a default (b=3) can be omitted.

```
z(10)

[1] 3.333
```

The ... argument to functions

```
z <- function(data, ...) {
    est <- lm(lwage ~ exp + sex + ed, data = data)
    linearHypothesis(est, "6*exp=ed", ...)[["Pr(>F)"]][2]
}
```

```
z(Wages)

[1] 0.2176

z(Wages, vcov = hccm)

[1] 0.2459
```

The ... argument can be used to pass on a variable number of arguments to another function. This is also very useful for `plot`.

7.3 Return values

Return values of functions

Functions always return the result of the last statement:

```
z <- function(x) {
    q <- x + 1
    r <- x - 1
    r^2
}
z(10)

[1] 81
```

Return values of functions

If we can to return more than one value, then we can use lists:

```
z <- function(x) {
    list(x = x, y = x^2, z = x^3)
}
result <- z(10)
str(result)

List of 3
 $ x: num 10
 $ y: num 100
 $ z: num 1000

result$z

[1] 1000
```

Most higher level statistical functions return lists. This makes it easy to access their results.

Scope of variables

Assignments to variables only change these variables within a function.

```
q <- 7
z <- function() {
    q <- 3
    q
}
z()
```

```
[1] 3

q

[1] 7
```

If we really want to change variables *outside* the scope of a function, we have to use `<<-`.

Scope of variables

```
q <- 7
z <- function() {
    q <<- 3
    q
}
z()

[1] 3

q

[1] 3
```

`z()` is now a function with a *side effect*, it changes the value of `q`.
Most of the time this is confusing and should be avoided.

7.4 Debugging functions

Debugging

The following will not work:

```
bad <- function(x) {
    lm(x ~ y, data = doesnotexist)
}
bad()

Error:  object 'doesnotexist' not found
```

Debugging

```
options(error = recover)
bad()
```

```
Error in inherits(x, "data.frame") : object "doesnotexist" not found

Enter a frame number, or 0 to exit

1: bad()
2: lm(x ~ y, data = doesnotexist)
3: eval(mf, parent.frame())
4: eval(expr, envir, enclos)
5: model.frame(formula = x ~ y, data = doesnotexist, drop.unused.levels = TRUE)
6: model.frame.default(formula = x ~ y, data = doesnotexist, drop.unused.levels = TRUE)
7: is.data.frame(data)
8: inherits(x, "data.frame")

Selection:
```

We can now choose a *frame* to inspect (here we choose frame 2):

Debugging

```
 Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "method"
 [9] "na.action"   "offset"      "qr"          "ret.x"       "ret.y"       "singular.ok"
[17] "x"           "y"
Browse[1]> Q
```

Functions

- `options(error=recover)`, `options(error=stop)`
- `debug`, `debugonce`, `undebug`, `trace`

8 Control and repetition

8.1 if and else

if and else

```r
x <- 7
if (x > 5) print("x is larger than five") else print("x is small")

[1] "x is larger than five"
```

`ifelse` does something different:

```
x <- 31:40
x

 [1] 31 32 33 34 35 36 37 38 39 40

ifelse(x > 35, x + 10, x - 10)

 [1] 21 22 23 24 25 46 47 48 49 50
```

8.2 for, while, repeat

for, while, repeat

```
for (i in 1:3) print(i^2)

[1] 1
[1] 4
[1] 9
```

```
i <- 1
while (i < 4) {
    print(i^2)
    i <- i + 1
}

[1] 1
[1] 4
[1] 9
```

for, while, repeat

```
i <- 1
repeat {
    print(i^2)
    i <- i + 1
    if (i > 3)
        break
    next
}

[1] 1
[1] 4
[1] 9
```

We should avoid `for`, `while`, and `repeat`. They are slow and clumsy.

8.3 sapply and apply

sapply

`sapply` applies a function to a vector (and returns a vector).

```
sapply(1:3, function(x) x^2)

[1] 1 4 9
```

```
sapply(1:3, log)

[1] 0.0000 0.6931 1.0986
```

```
sapply(1:3, log, base = 3)

[1] 0.0000 0.6309 1.0000
```

```
sapply(1:3, function(x) c(x = x, y = x^2, r = 1/x))

  [,1] [,2]    [,3]
x    1  2.0 3.0000
y    1  4.0 9.0000
r    1  0.5 0.3333
```

apply

`apply` applies a function (here `sum`) along a dimension of an array (here `1` and `2`) and returns an array of lower dimensionality.

```
(x <- rbind(c(1, 2, 3), c(4, 5, 6)))

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

apply(x, 1, sum)

[1]  6 15

apply(x, 2, sum)

[1] 5 7 9
```

8.4 Wide and long arrays

Wide and long regular arrays:

wide form of a regular array:

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

long form of the same regular array:

row	column	value
1	1	a
1	2	b
1	3	c
2	1	d
2	2	e
2	3	f

Wide and long ragged arrays:

wide form of a *ragged* array:

$$\begin{pmatrix} a & b & \\ d & & f \end{pmatrix}$$

long form of the same *ragged* array:

row	column	value
1	1	a
1	2	b
2	1	d
2	3	f

We apply functions along the dimensions of a regular or ragged array in *long* form with `aggregate` and `by`.

8.5 aggregate and by

aggregate and by

index$_1$	index$_2$	x	y	z
1	1	x_1	y_1	z_1
1	1	x_2	y_2	z_2
1	1	x_3	y_3	z_3
1	2	x_4	y_4	z_4
1	2	x_5	y_5	z_5
1	2	x_6	y_6	z_6
1	2	x_7	y_7	z_7
2	1	x_8	y_8	z_8
2	1	x_9	y_9	z_9
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

`aggregate(...)`

index$_1$	index$_2$	x	y	z
1	1	x_1	y_1	z_1
1	1	x_2	y_2	z_2
1	1	x_3	y_3	z_3
1	2	x_4	y_4	z_4
1	2	x_5	y_5	z_5
1	2	x_6	y_6	z_6
1	2	x_7	y_7	z_7
2	1	x_8	y_8	z_8
2	1	x_9	y_9	z_9
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

`by(...)`

aggregate

```r
aggregate(Wages, list(ed = Wages$ed), mean)
```

ed	exp	wks	bluecol	ind	south	smsa	married	sex	union	ed	black	lwage
4	22.00	46.00		0.64						4.00		6.04
5	37.67	45.71		0.33						5.00		6.44
6	31.00	45.86		0.93						6.00		6.42
7	23.91	47.10		0.73						7.00		6.27
8	27.84	45.35		0.64						8.00		6.41
9	22.26	46.67		0.50						9.00		6.47
10	24.30	46.50		0.53						10.00		6.50
11	22.83	46.43		0.48						11.00		6.52
12	19.49	47.11		0.44						12.00		6.60
13	15.38	48.05		0.30						13.00		6.59
14	17.29	47.62		0.27						14.00		6.73
15	20.83	48.10		0.57						15.00		6.88
16	17.03	47.10		0.27						16.00		6.89
17	18.85	45.29		0.23						17.00		6.98

by

```r
by(Wages, list(ed = Wages$ed), function(x) coef(lm(lwage ~ exp, data = x)))

ed: 4
(Intercept)         exp
    5.49710     0.02459
------------------------------------------------------------
ed: 5
```

```
(Intercept)          exp
    7.78186    -0.03573
-------------------------------------------------------------
ed: 6
(Intercept)          exp
   6.299444    0.004007
-------------------------------------------------------------
ed: 7
(Intercept)          exp
    5.85985     0.01705
-------------------------------------------------------------
ed: 8
(Intercept)          exp
    5.64363     0.02745
-------------------------------------------------------------
ed: 9
(Intercept)          exp
   6.380162    0.004161
-------------------------------------------------------------
ed: 10
(Intercept)          exp
   6.342956    0.006462
-------------------------------------------------------------
ed: 11
(Intercept)          exp
    5.99853     0.02282
-------------------------------------------------------------
ed: 12
(Intercept)          exp
    6.40168     0.01036
-------------------------------------------------------------
ed: 13
(Intercept)          exp
    6.37998     0.01383
-------------------------------------------------------------
ed: 14
(Intercept)          exp
    6.48491     0.01445
-------------------------------------------------------------
ed: 15
(Intercept)          exp
  6.8984666  -0.0007323
-------------------------------------------------------------
```

```
ed: 16
(Intercept)         exp
     6.5643      0.0193
------------------------------------------------------------
ed: 17
(Intercept)         exp
    6.74654     0.01257
```

by, decluttered

```
x <- by(Wages, list(ed = Wages$ed), function(x) coef(lm(lwage ~ exp, data = x)))
```

```
sapply(x, c)
```

```
                  4        5        6        7       8        9       10      11
(Intercept) 5.49710  7.78186 6.299444 5.85985 5.64363 6.380162 6.342956 5.99853
exp         0.02459 -0.03573 0.004007 0.01705 0.02745 0.004161 0.006462 0.02282
                 12      13      14         15     16      17
(Intercept) 6.40168 6.37998 6.48491  6.8984666 6.5643 6.74654
exp         0.01036 0.01383 0.01445 -0.0007323 0.0193 0.01257
```

```
t(sapply(x, c))
```

```
   (Intercept)          exp
4        5.497   0.0245893
5        7.782  -0.0357275
6        6.299   0.0040072
7        5.860   0.0170517
8        5.644   0.0274480
9        6.380   0.0041614
10       6.343   0.0064618
11       5.999   0.0228244
12       6.402   0.0103584
13       6.380   0.0138323
14       6.485   0.0144524
15       6.898  -0.0007323
16       6.564   0.0193044
17       6.747   0.0125693
```

9 Organising data

9.1 Merge + Append

merge as append

```
W1 <- subset(Wages, exp == 1)
W2 <- subset(Wages, exp == 2)
merge(W1, W2, all = TRUE)

   exp wks bluecol ind south smsa married    sex union ed black lwage
1    1  30      no   0    no  yes     yes   male    no 17    no 6.215
2    1  35     yes   1    no  yes      no   male    no 14    no 5.704
3    1  45     yes   0    no  yes      no   male    no 11    no 5.704
 [ reached getOption("max.print") -- omitted 23 rows ]
```

merge

```
(X <- as.data.frame(list(ed = c(9, 11, 12), type = c("A", "B", "C"))))

  ed type
1  9    A
2 11    B
3 12    C

merge(Wages, X)[, c("ed", "exp", "wks", "type")]

     ed exp wks type
1     9  19  45    A
2     9  22  50    A
3     9  31  49    A
4     9  45  51    A
5     9  13  49    A
6     9  33  48    A
7     9  16  52    A
8     9  28  47    A
9     9  35  52    A
10    9   6  50    A
 [ reached getOption("max.print") -- omitted 1859 rows ]
```

merge

```
merge(Wages, X, all = TRUE)[, c("ed", "exp", "wks", "type")]

     ed exp wks type
1     4  28  42 <NA>
2     4  29  46 <NA>
3     4  30  49 <NA>
4     4  31  49 <NA>
5     4  32  48 <NA>
6     4  33  45 <NA>
7     4  34  42 <NA>
8     4  10  49 <NA>
9     4  12  39 <NA>
10    4  13  47 <NA>
 [ reached getOption("max.print") -- omitted 4155 rows ]
```

9.2 Strings and Renaming

Let us, for the sake of the example, assume that we want to combine several variables (of different types) into one character variable (sometimes we do this to create a factor, think of combining the `Date` and `Subject` in a z-Tree file to create a unique subject-id).

sprintf: From numbers to strings

exp	wks	bluecol	ind	south	smsa	married	sex	union	ed	black	lwage
3	32	no	0	yes	no	yes	male	no	9	no	5.56
4	43	no	0	yes	no	yes	male	no	9	no	5.72
5	40	no	0	yes	no	yes	male	no	9	no	6.00
6	39	no	0	yes	no	yes	male	no	9	no	6.00
7	42	no	1	yes	no	yes	male	no	9	no	6.06
8	35	no	1	yes	no	yes	male	no	9	no	6.17
9	32	no	1	yes	no	yes	male	no	9	no	6.24
30	34	yes	0	no	no	yes	male	no	11	no	6.16
31	27	yes	0	no	no	yes	male	no	11	no	6.21
32	33	yes	1	no	no	yes	male	yes	11	no	6.26
33	30	yes	1	no	no	yes	male	no	11	no	6.54
34	30	yes	1	no	no	yes	male	no	11	no	6.70
35	37	yes	1	no	no	yes	male	no	11	no	6.79
36	30	yes	1	no	no	yes	male	no	11	no	6.82

sprintf: From numbers to strings

```
with(Wages, sprintf("%s-%d-%.2f", sex, ed, lwage))

 [1] "male-9-5.56"    "male-9-5.72"    "male-9-6.00"    "male-9-6.00"
```

```
 [5] "male-9-6.06"    "male-9-6.17"    "male-9-6.24"    "male-11-6.16"
 [9] "male-11-6.21"   "male-11-6.26"   "male-11-6.54"   "male-11-6.70"
[13] "male-11-6.79"   "male-11-6.82"   "male-12-5.65"   "male-12-6.44"
[17] "male-12-6.55"   "male-12-6.60"   "male-12-6.70"   "male-12-6.78"
[21] "male-12-6.86"   "female-10-6.16" "female-10-6.24" "female-10-6.30"
[25] "female-10-6.36" "female-10-6.47" "female-10-6.56" "female-10-6.62"
[29] "male-16-6.44"   "male-16-6.62"   "male-16-6.63"   "male-16-6.98"
[33] "male-16-7.05"   "male-16-7.31"   "male-16-7.30"   "male-12-6.91"
[37] "male-12-6.91"   "male-12-6.91"   "male-12-7.00"   "male-12-7.07"
 [ reached getOption("max.print") -- omitted 4125 entries ]
```

sprintf: From numbers to strings

`%s`	string
`%10s`	string of width 10
`%d`	integer
`%8d`	integer of width 8
`%f`	floating point number
`%10.2f`	floating point number of width 10 with 2 digits precision

Manipulating strings

- Strings as data which we want to manipulate.
- Names of variables which we want to change in a systematic way.

```
W <- Wages
names(W)

 [1] "exp"     "wks"     "bluecol" "ind"     "south"   "smsa"    "married"
 [8] "sex"     "union"   "ed"      "black"   "lwage"
```

```
names(W)[3]

[1] "bluecol"
```

```
names(W)[3] <- "Worker"
```

```
names(W)

 [1] "exp"     "wks"     "Worker"  "ind"     "south"   "smsa"    "married"
 [8] "sex"     "union"   "ed"      "black"   "lwage"
```

```
W$Worker[1:13]

 [1] no  no  no  no  no  no  no  yes yes yes yes yes yes
Levels: no yes
```

Manipulating strings

- Can we change the name in a safer way?

```
names(W)[names(W) == "Worker"] <- "bluecollar"
names(W)

 [1] "exp"        "wks"        "bluecollar" "ind"        "south"
 [6] "smsa"       "married"    "sex"        "union"      "ed"
[11] "black"      "lwage"
```

Changing names/strings in a systematic way

Assume that we have the following names (of variables):

```
n <- c("Date", "Subject", "Period", "Invest", "Time.A", "Time.B", "Time.C")
```

How do we drop all the variables that start with `Time` ?
`grep` finds strings that match a pattern:

```
grep("Time", n)

[1] 5 6 7
```

If we want to see what we have found:

```
grep("Time", n, value = TRUE)

[1] "Time.A" "Time.B" "Time.C"
```

We can also invert the selection (we want to keep the variables who do not match):

```
grep("Time", n, invert = TRUE)

[1] 1 2 3 4

n[grep("Time", n, invert = TRUE)]

[1] "Date"    "Subject" "Period"  "Invest"
```

Even safer would be to say the following:

```
n[grep("^Time", n, invert = TRUE)]

[1] "Date"    "Subject" "Period"  "Invest"
```

This would make sure that only strings that *start* with `Time` match (and not `SomeTime`).

Changing names/strings in a systematic way

```
n <- c("Date", "Subject", "Period", "A_invest", "B_invest", "C_invest", "other")
```

How can we translate `A_invest` into `Inv_A` ?

```
sub("(.*)_invest", "Inv_\\1", n)

[1] "Date"    "Subject" "Period"  "Inv_A"   "Inv_B"   "Inv_C"   "other"
```

The pattern `(.*)` follows the syntax of "regular expressions" (see `help(regexp)`).

Regular expressions

- `a-z0-9` matches itself
- `.` matches anything
- `*` matches whatever was before zero or more times (hence, `.*` matches anything)
- `+` matches whatever was before one or more times
- `?` matches whatever was before at most once.
- `[3-5]` matches `3`, or `4`, or `5`
- `[^3-5]` matches everything, except `3`, or `4`, or `5`
- `^` matches the start of the string (e.g. `^abc` matches `abc` at the start of the string)
- `$` matches the end of the string
- `(...)` `(...)` `(...)` the match of each group of parentheses can be used as \1, \2, \3, ... (the letter \ has to be escaped in R, like \\.

For more details, see `help(regexp)`.

Strings and Formulas in Regressions

Sometimes we want to use many variables in a regression. We can extract them with `grep`, paste them together with `paste`, and make them a formular with `as.formula`. Let us assume that we want, for the sake of the example, use all variables that start with the letter `b`:

```
(varnames <- grep("^b", names(Wages), value = TRUE))

[1] "bluecol" "black"
```

```
(myForm <- as.formula(paste("lwage ~", paste(varnames, collapse = "+"))))

lwage ~ bluecol + black
```

```
lm(myForm, data = Wages)


Call:
lm(formula = myForm, data = Wages)

Coefficients:
(Intercept)    bluecolyes     blackyes
      6.841        -0.281       -0.292
```

9.3 Reshape

Reshaping data

How can we reshape a data from from wide to long form and vice versa?

```
x <- as.data.frame(cbind(id = c(1:3), invest.1 = c(100:102), invest.2 = c(200:202)))
x

  id invest.1 invest.2
1  1      100      200
2  2      101      201
3  3      102      202
```

```
reshape(x, idvar = "id", varying = list(2:3), direction = "long")

    id time invest.1
1.1  1    1      100
2.1  2    1      101
3.1  3    1      102
```

```
1.2  1    2      200
2.2  2    2      201
3.2  3    2      202
```

and if we do not want to list the wide variables as numbers we can use `grep`:

```
grep("^invest", names(x))

[1] 2 3

(longx <- reshape(x, idvar = "id", varying = grep("^invest", names(x)), direction = "long

    id time invest
1.1  1    1    100
2.1  2    1    101
3.1  3    1    102
1.2  1    2    200
2.2  2    2    201
3.2  3    2    202
```

Reshape back to wide

```
reshape(longx, v.names = "invest", idvar = "id", timevar = "time", direction = "wide")

    id invest.1 invest.2
1.1  1      100      200
2.1  2      101      201
3.1  3      102      202
```

if the long format misses some ids:

```
longx[-3, ]

    id time invest
1.1  1    1    100
2.1  2    1    101
1.2  1    2    200
2.2  2    2    201
3.2  3    2    202
```

```
reshape(longx[-3, ], v.names = "invest", idvar = "id", timevar = "time", direction = "wi

    id invest.1 invest.2
1.1  1      100      200
2.1  2      101      201
3.2  3       NA      202
```

Functions

- `merge`
- `reshape`

- `sprintfs`
- `grep`
- `sub`
- `regexp`

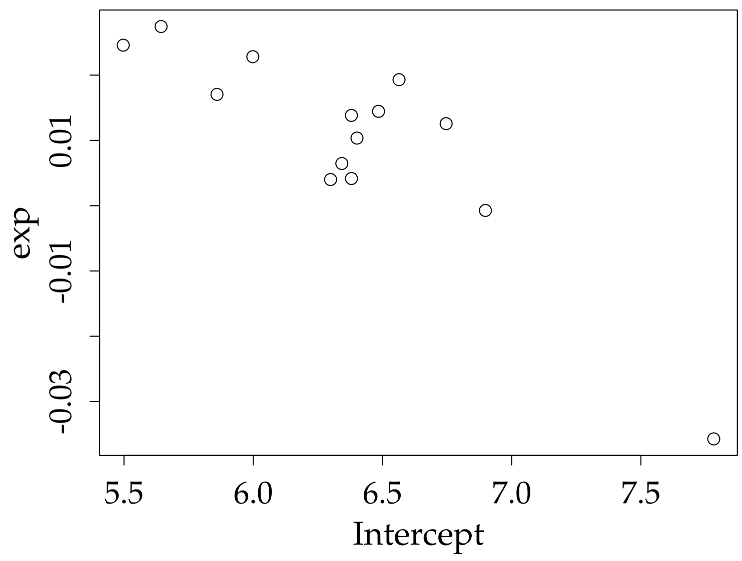
10 Plotting

10.1 The standard plot interface

R his different plotting routines for a large variety of objects. A $n \times 2$ matrix, e.g., is plotted as a scatterplot:

A simple plot

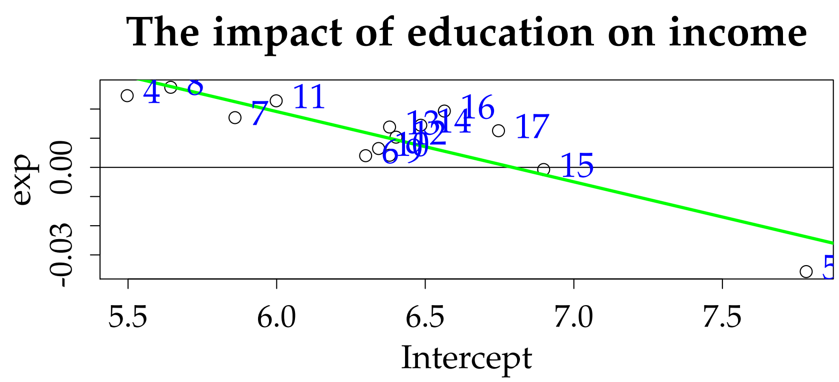
```
plot(x2)
```



Adding elements to a plot

We can add arbitrary elements to a plot:

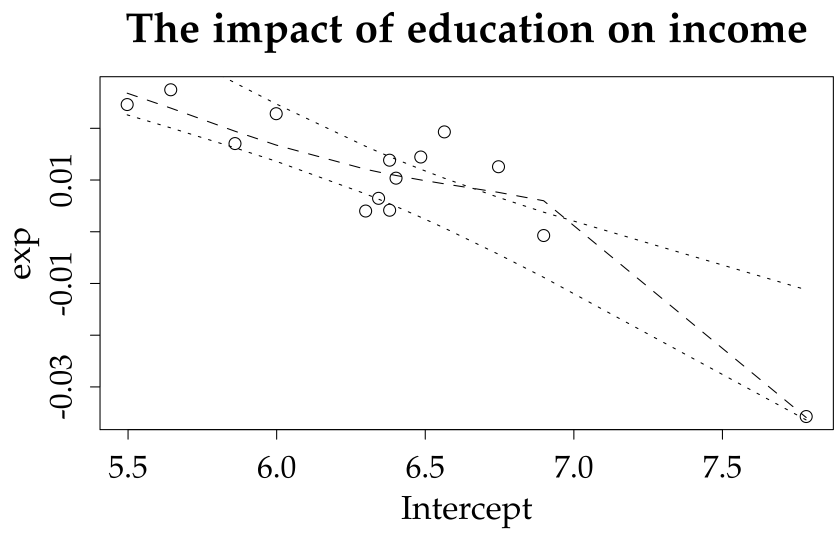
```
plot(x2)
abline(h = 0)
title(main = "The impact of education on income")
est <- lm(exp ~ Intercept, data = x2)
abline(est, col = "green", lwd = 3)
text(x2, labels = rownames(x2), pos = 4, col = "blue")
```



Adding confidence bands

Confidence bands and polynomial approximations can be calculated with `predict`:

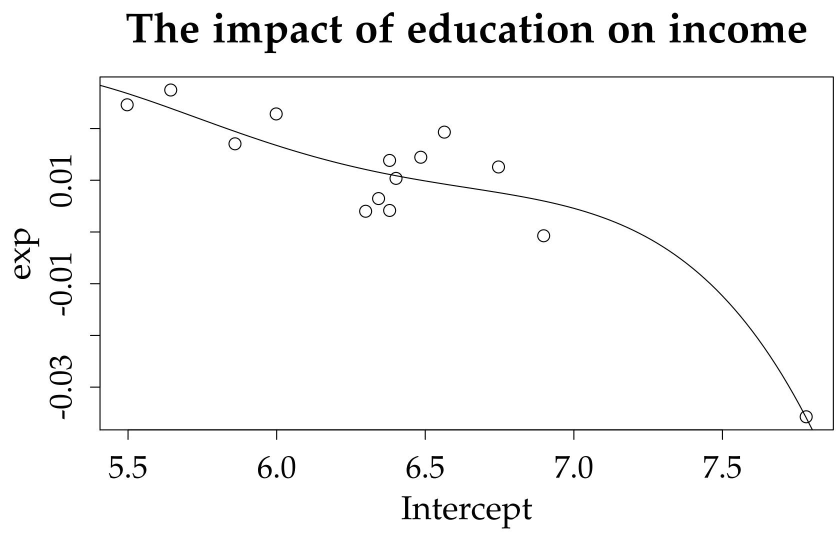
```
par(mar = c(4, 4, 3, 0.1))
plot(x2)
title(main = "The impact of education on income")
est.pred <- as.data.frame(predict(est, interval = "confidence"))
est.pred$Intercept <- x2$Intercept
attach(est.pred)
lines(Intercept, upr, lty = "dotted")
lines(Intercept, lwr, lty = "dotted")
detach(est.pred)
est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = 4), data = x2))
lines(x2$Intercept, est.pred4, lty = "dashed")
```



Adding smoothness

Using only the original data did not produce an entirely satisfactory result. To get a more smooth curve, we produce a new dataset with a finer resolution:

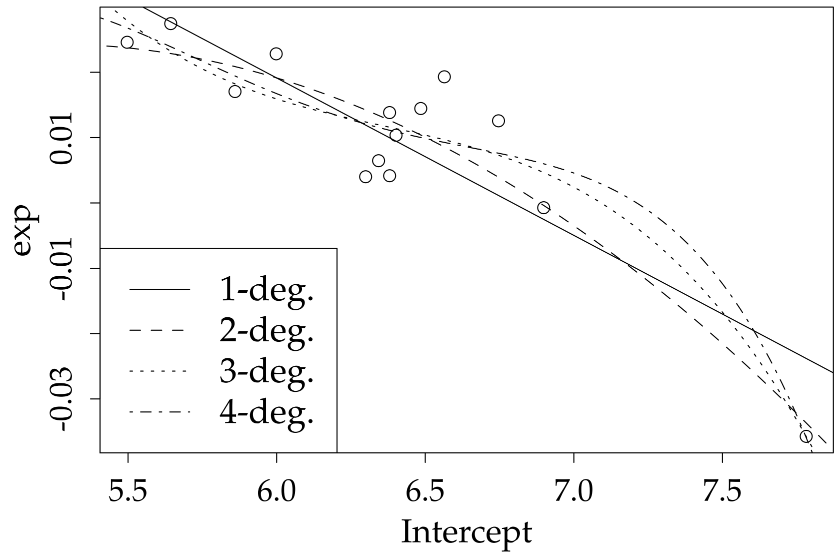
```
plot(x2)
title(main = "The impact of education on income")
x4 <- list(Intercept = seq(5, 8, 0.02))
est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = 4), data = x2), newdata = x4)
lines(x4$Intercept, est.pred4)
```



Several polynomials

If we want to look at more than a single polynomial, we can try a loop:

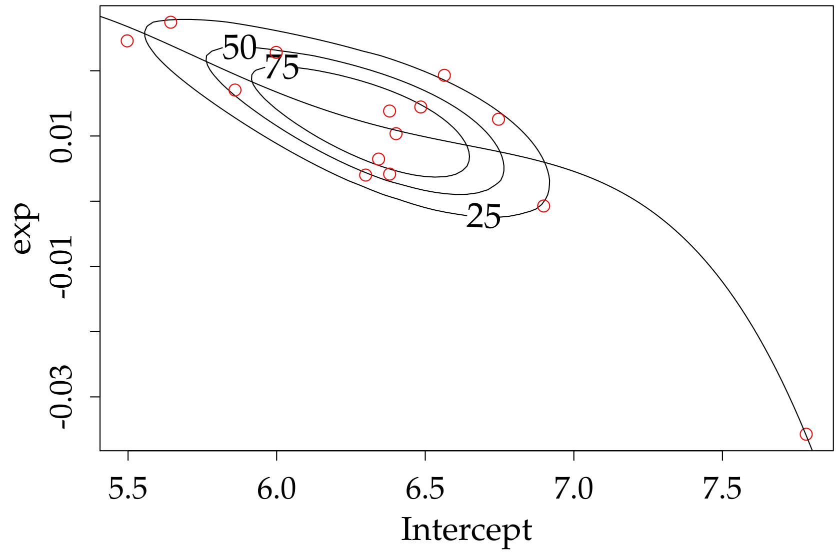
```
plot(x2)
for (i in c(1:4)) {
    est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = i), data = x2), newdata = x4)
    lines(x4$Intercept, est.pred4, lty = i)
}
legend("bottomleft", legend = sprintf("%d-deg.", 1:4), lty = 1:4)
```



Kernel density plot

Here is another way to organise the data with the help of a kernel density plot:

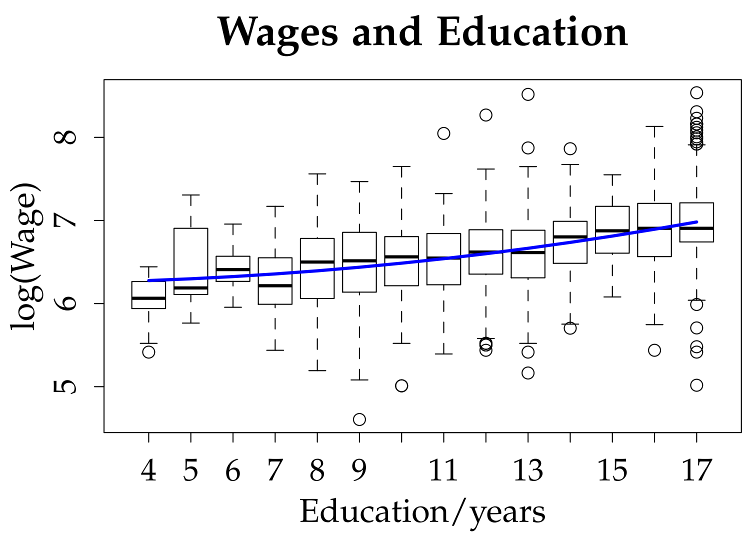
```
library(ks)
H.scv <- Hscv(x2)
kdeObj <- kde(x2, H = H.scv)
plot(kdeObj)
points(x2, col = "red")
lines(x4$Intercept, est.pred4)
```



A boxplot

R can also draw boxplots:

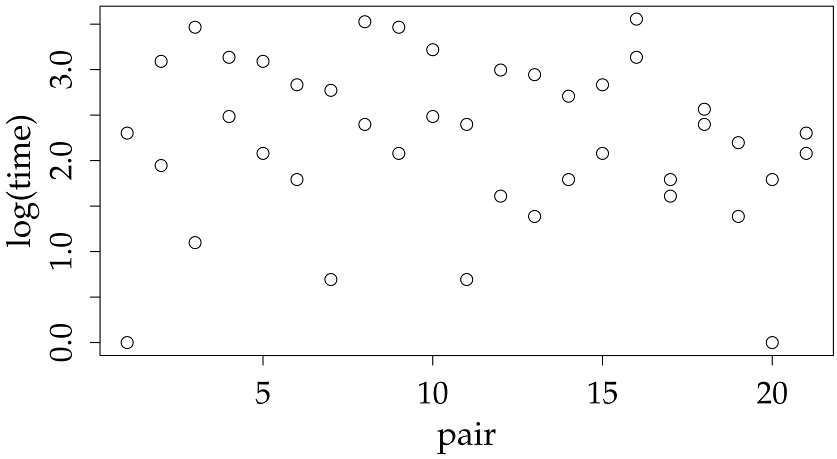
```
boxplot(lwage ~ ed, data = Wages)
title(main = "Wages and Education", xlab = "Education/years", ylab = "log(Wage)")
est <- lm(lwage ~ poly(ed, degree = 2), data = Wages)
lines(1:14, predict(est, newdata = list(ed = c(4:17))), col = "blue", lwd = 3)
```



Survival analysis

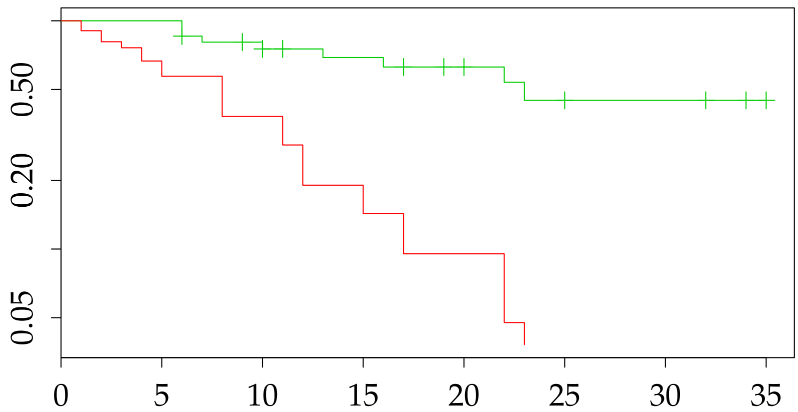
Here is a survival analysis. A `Surv()` object is just a special kind of dependent variable.

```
library(survival)
library(MASS)
plot(log(time) ~ pair, data = gehan)
```



Survival analysis

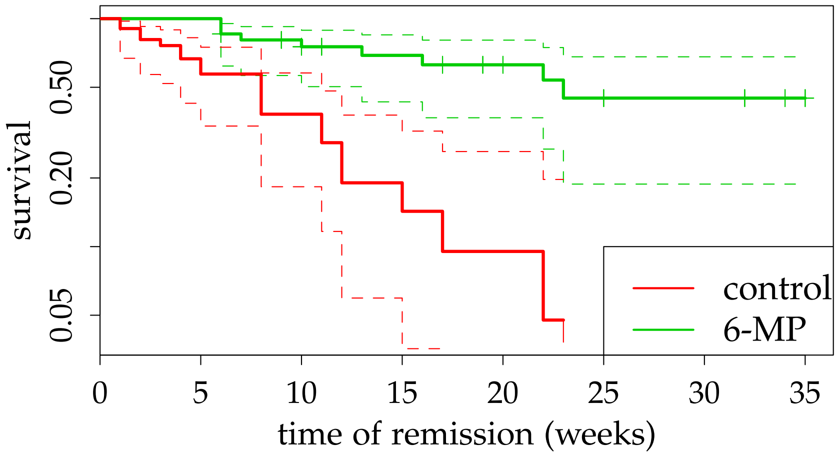
```
qq <- Surv(gehan$time, gehan$cens)
gehan.surv <- survfit(Surv(time, cens) ~ treat, data = gehan, conf.type = "log-log")
plot(gehan.surv, col = 3:2, log = TRUE)

Warning:  1 y value <= 0 omitted from logarithmic plot
```



```
summary(gehan.surv)

Call: survfit(formula = Surv(time, cens) ~ treat, data = gehan, conf.type = "log-log")

                treat=6-MP
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    6     21       3    0.857  0.0764        0.620        0.952
    7     17       1    0.807  0.0869        0.563        0.923
   10     15       1    0.753  0.0963        0.503        0.889
   13     12       1    0.690  0.1068        0.432        0.849
   16     11       1    0.627  0.1141        0.368        0.805
 [ reached getOption("max.print") -- omitted 2 rows ]

                treat=control
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    1     21       2   0.9048  0.0641      0.67005        0.975
    2     19       2   0.8095  0.0857      0.56891        0.924
    3     17       1   0.7619  0.0929      0.51939        0.893
    4     16       2   0.6667  0.1029      0.42535        0.825
    5     14       2   0.5714  0.1080      0.33798        0.749
 [ reached getOption("max.print") -- omitted 7 rows ]
```

Survival analysis

```
plot(gehan.surv, conf.int = TRUE, col = 3:2, log = TRUE)

Warning:  1 y value <= 0 omitted from logarithmic plot

lines(gehan.surv, col = 3:2, lwd = 3, cex = 2)
title(xlab = "time of remission (weeks)", ylab = "survival")
legend(25, 0.1, c("control", "6-MP"), col = 2:3, lwd = 2)
```



Survival analysis

```
survdiff(Surv(time, cens) ~ treat, data = gehan)

Call:
survdiff(formula = Surv(time, cens) ~ treat, data = gehan)

                N Observed Expected (O-E)^2/E (O-E)^2/V
treat=6-MP     21        9     19.3      5.46      16.8
treat=control  21       21     10.7      9.77      16.8

 Chisq= 16.8  on 1 degrees of freedom, p= 4.17e-05
```