

# Introduction to R

---

Dr. Willi Mutschler

# Introduction

---

# Aims and prerequisites

- Objective: Learn how to use R for econometrics and statistics
- Prerequisites:
  1. Basics in probability theory and statistical inference
  2. Multiple linear regression

**Essential:** Andreas Behr and Ulrich Pötter (2011): *Einführung in die Statistik mit R*

**Additional:** Muenchen, Hilbe (2012): *R for Stata Users*

Introductory courses on [datacamp.com](https://datacamp.com)

Introductory tutorials on [r-bloggers.com](https://r-bloggers.com)

1. What is R?
2. R-Studio Basics
3. Managing workspace and packages
4. Get help and understand the documentation
5. Programming language basics
6. Controlling functions
7. Data structures and acquisition
8. Selection and transformations of variables and observations
9. Treatment of missing values
10. Data visualization (basic and using grammar)
11. Basic statistics
12. Linear modeling: regression analysis

# What is R?

---

# What is R?

**“The most powerful statistical computing language on the planet...”**

**...It all depends on the use and user**

# What is R?

- The language S (an object-oriented statistical computing language) is implemented as S-Plus (commercial) and R (OpenSource)
- R is a
  1. language
  2. package
  3. environmentfor graphics and data analysis
- R is FOSS (think about collaborations!) and easily extendable
- Similar programming languages: Matlab, GAUSS, Julia



The differences between S-Plus and R are minimal

# Comparison to STATA

Basically five independent parts of a software

- Data input and management (language)
- Statistics and graphics procedures / commands
- Output management systems
- Macro language
- Matrix language

↪ In other softwares, e.g. Stata, these are standalone and developed separately

↪ In R all five were planned to be unified from the beginning

Output management systems: Stata return codes, postestimation, Matrix language in Stata is Mata

# Comparison to STATA

- Base R plus recommended packages contains over 1600 functions similar to e.g. STATA
- Tested via extensive validation programs
  - ↳ R is accurate even though no company behind it, R responds very quickly to bugs etc.
  - ↳ Source code of R (*scripts*) are similar to *Do Files* in STATA
- Note that new statistical methods are nowadays often first published in R, and only later included by PROGRAMMERS (not original scientists) into STATA

This has pro and cons

# R Console

The basic command window is called the R Console

Prompt: >

You can input commands and execute them (by pressing the RETURN key)

```
> 1+1
> 1+1 # This is a comment: 1+1
> (1+2)*3
> (5/3)^4.5
> 5+2; 7+3; 2*5
> pi
> Pi
> PI
> 2*((1+2)*(1-2))
```

This will: (1) evaluate it, (2) print the result, (3) count the lines with [n] and (4) delete the result

That a plus sign appears if we forget to close paranthesis.

Upper and lower cases, dot and comma: R differentiates between upper and lower cases, try: pi, Pi und PI. Keep that in mind when calling functions and commands, and also when naming variables.

The decimal point is the dot.

- Long computations should not be done interactively in the command window
- Use an editor to write a program and then execute it in R
- There is a built-in editor in R: `Datei - Neues Skript`
- External editors:
  - **R-Studio [RECOMMENDED!]**
  - Tinn-R, Notepad++, Atom, Emacs, etc. are also possible



- Overview of four panels in R-Studio:
  1. Script editor
  2. Environment|History|Connections,
  3. Console|Terminal
  4. Files|Plots|Packages|Help|Viewer
- Important shortcuts (see also the Magic Wand)
  - [CTRL+ENTER] for Run
  - [CTRL+SHIFT+C] for commenting a section
  - [CTRL+L] clears command windows
  - [TAB] Function completion
  - [ARROWS UP AND DOWN] in command window: scroll through history
  - [F1] gets you help

One can change this layout as one sees fit in the options interface

Actually R-Studio can also be run on a server easily such that one could call `rstudio.mutschler.eu`

## Concatenation and Assignments

Open a new Script in R-Studio and try out the following:

```
c(1,4,7)
a <- c(1,4,7)
print(a)
a
A
b <- c(1,a,3)
(b <- c(1,a,3))
mean(b)
demo("graphics")
```

Execute a single line (or multiple lines by marking them) by pressing CTRL-ENTER

Variables are case-sensitive, e.g. pi, Pi, PI

Assignments: evaluate expression and store result <- ("gets"), = works too but avoid it you can enter command with semicolon or separate commands, but usually just new line

a + at the command line indicates that a command is not complete, either complete it or hit ESC, - if you did not complete commands, you get a +, mostly ), or hit ESC key or STRG/CMD+C

## Managing workspace

---

# Managing workspace

## Listing Objects

- `ls()` or `objects()` lists the objects in your workspace
- `list=ls()` clears workspace

## Working Directory

- Easiest: Use GUI, i.e. Session - Set Working Directory
- alternatively: `getwd()` and `setwd("c:/temp")`
- Note that the path name is structured by slashes (/), not backslashes (\)
- some special hidden files are in your working directory

Special R Files: Rproj that remembers the open files and working directory, stores settings in a folder named .Rproj.user R looks for the following files: .Rprofile (like profile.do) commands are automatically executed at startup (not a good practice) .Rhistory history of all statements! click on history tab, or history() .Rdata (not good)

Save as save.image(file="file.RData") save(mydata,mylist,mymatrix,file="mypractice.R")

Save Workspace image -> creates .RData in the current wd, when you start it will load it automatically, (TOOLS OPTION GENERAL, what you want)

## Quitting

- Quit R by the command `q()`
- Quit RStudio by using the GUI or [CTRL+Q]
- In general, do *not* save your workspace



## Misc

- if you did not complete commands, you get a +. Most of the times close a ), or hit ESC key or CTRL+C
- comments go from # to the end of line, can be between functions or in the middle with line breaks
- there is no block comment features, simply use [CTRL-SHIFT-C] in R-Studio to (un)comment sections

```
d <- #output name
```

```
2 * #first param
```

```
3 #second param
```

```
print(d)
```

# Packages

---

# Packages

- One of the strengths of R is the large and growing collection of packages that can be downloaded from CRAN (or Github, etc)
- Installation (only once!)
  - Use R-Studio interface for Packages
  - `install.packages("packagename")`
- Installed packages are activated by `library("packagename")`
- Help about packages: `library(help="packagename")`

Actually, library does not need quotations, but who cares.

# Common problems

- `install.packages("ggplot")`
  - ↳ either not available or wrong package name
- `install.packages(ggplot2)`
  - ↳ forgot quotes
- `library("prettyR")`
  - ↳ forgot to install it
- `library("dplyr")`
  - ↳ masked or covered up, that's okay
- `detach("package:dplyr")` is opposite of `library`, which might prevent conflicts in function names and save on memory

## Which packages to use?

- The Comprehensive R Archive Network (CRAN): can be searched by name or via Task Views for key programs on `cran.r-project.org`
- `crantastic.org`: rated software
- `rdocumentation.org`: top packages
- `r-bloggers.com`
- Git[hu|la]b, ...

# Exercise 1



## **Help and documentation**

---

# Help and documentation

- To obtain details about a command, type  
`?command` or `help(command)`

```
?mean
help(mean)
help("for")
help("while")
?"while"
help(package = "prettyR")
methods(plot) #gives you overview of extra
               functions, e.g.
help(plot.lm)
help.start()
```

- R-Studio: select/click on a command and hit F1

Some commands belong to a certain package or base, and need to be asked for with quotation marks

# Programming Language Basics

---

# R objects

- R is object oriented
- An object can be anything: scalar, vector, matrix, string, table, factor, list, data frame, regression results, model, . . .
- object name should begin with letter, no numbers, no underscores, no special characters, case matters
- The object type determines how some commands work (e.g. `plot`, `summary`)
- Every object has a unique name

# Parenthesis

(Parenthesis)

- control math order as usual in algebra, e.g.

```
1+1*10  
(1+1)*10
```

- print assignment values:

```
(x<-12)
```

- provide options to functions, e.g.

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

# Parenthesis

## {Curly Braces}

- Can combine many commands into one
- Executes all assignments but returns only value of last expression
- Useful for writing own functions

```
{x<-2; y<-1  
z<-x+y; z2<- z^2  
z  
z2}
```

## [Square Braces] and [[Double Square Braces]]

- Used for selecting/indexing elements within objects
- Double squares are used in lists (one of the most general data structure)

All kinds of values can be stored in a variable (as we are object-oriented):

- numbers
- letters
- words
- dates
- logical TRUE/FALSE values
- data structures
- ....



## Mode, Class, Dimension and Length of Vectors

```
x <- c(1, 2, 1, 2, 1, 2, 1, 2)
print(x)
mode(x)
class(x)
length(x)
dim(x)
x + x
2*x
x + 10
x + c(10, 100)
```

- `mode`: a variable's type
- `class`: vectors have a class of character or numeric (or many other things), dimension (`dim`) of a vector is `NULL`
- `length`: number of elements it contains (including (!) missings)
- Note: If one vector is shorter, its values are **recycled** until the lengths match

Advantage: saves you typing,

Attention: a vector is NOT a 1D-vector!

# Operators and functions

## Logical operators

& and

| or

! not

NA not available or no answer

== equal (do *not* use =)

>, >= greater than, greater than or equal

<, <= less than, less than or equal

!= not equal

# Operators and functions

## Examples logical operators

```
5 < 7
1+1 == 3
a <- c(-1,4,9)
a >= 2 \& a < 8
b <- c(NA,1,2,3)
b > 0
is.na(b)
a[a>2]
a == 4
a = 4
```

## Exercise 2

# Operators and functions

## Arithmetic operators and mathematical functions

`+`, `-` plus, minus

`*`, `/` multiplication and division

`^` power (exponentiation)

`Inf`, `-Inf` infinity (plus or minus)

`NaN` not a number

`abs` absolute value

`sqrt` square root

`exp`, `log` exponential function and natural logarithm (*not* `ln`)

`sin` sinus (other trigonometric functions as well)

`sum` sum

# Operators and functions

Examples arithmetic operators and mathematical functions

```
x <- c(-1,0,2,9,3)
abs(x)
sqrt(x)
1/x
-1/x
0/x
log(x)
x^c(2,3,2,3,2)
x^c(2,3)
log(x)<0
```

## Exercise 3



# Operators and functions

## Matrix functions

`matrix` creates a matrix from a vector

`dim` dimensions of a matrix

`t` transpose

`%%` matrix multiplication

`det` determinant

`solve` inverse

`eigen` eigenvalues and eigenvectors

`diag` diagonal

`cbind` merge matrices column-wise

`rbind` merge matrices row-wise

# Operators and functions

## Examples matrix functions

```
X <- matrix(c(2,3,4,5,1,1,9,3,2),3,3)
X
dim(X)
det(X)
solve(t(X)%*%X)
X*c(8,5,1)
X%%c(8,5,1)
diag(X)
diag(X) <- 0
X
solve(X)%*%X
matrix(NA,4,4)
rbind(cbind(X,X),c(0,1))
```

Note the difference between `*` and `%*%!`

## Exercise 4

# Operators and functions

Set operations and special functions

`unique` the set of all unique elements of a vector

`union`  $x \cup y$

`intersect`  $x \cap y$

`setdiff`  $x \setminus y$

`%in%`  $x \in y$

`sort` sort the elements of a vector

`cumsum` cumulated sum of a vector  
(also `cumprod`, `cummin`, `cummax`)

`which(...)` find the index of the vector element for which some condition is true

`which.min` find the index of the smallest vector element  
(also `which.max`)

## Exercise 5

# Operators and functions

## Sequences and replications

**seq** sequence from  $a$  to  $b$  of length  $n$ ,  
seq(from= $a$ , to= $b$ , length= $n$ ),  
or by increments of size  $d$ ,  
seq(from= $a$ , to= $b$ , by= $d$ )

**a:b** integer sequence from  $a$  to  $b$

**rep** replicate a vector  $n$  times  
rep(what, times= $n$ ),  
or each element  $n$  times,  
rep(what, each= $n$ )

## Exercise 6

# Controlling Functions

---



# Calling Functions

- R is controlled by functions: when you use an R function you call it and pass values to their arguments
- arguments are listed in (parenthesis) and are separated by comas
- argument values are usually single objects and have a unique name
- function calls **return** a **value** (in help file, output is often called return/value)
- value is a single object, may contain much information, optimized for further analysis not (necessarily) for displaying

Hint: dont abbreviate TRUE AND FALSE by T and F. So be careful  
note that we may use nested functions

## More on function arguments

### Common Error:

```
x1 <-c(1,2,3); x2 <-c(5,6,7); x3 <-c(8,9,0);  
mean(x1,x2,x3) #nope!  
summary(data.frame(x1,x2,x3)) #better!
```

- When calling a function, the order of the arguments is arbitrary, if the argument names are explicitly used:  
`mean(na.rm=FALSE,trim=.1,x=mydata)`
- Without argument names, R assigns the values in the order of the function definition:  
`mean(mydata,.1,FALSE)`
- A function definition may include default values for arguments, e.g.  
`mean(x, trim = 0, na.rm = FALSE)`
- If an argument with a default value is missing in a function call, R uses the default value

# Data Structures

---

Most Statistics programs have only one data structure, R is more flexible

- Factors
- Arrays
- Vectors
- Matrices
- Data frames
- Tables
- Lists
- or make your own

# Factors

Does this make sense?

```
degree <- c(0,      2,    1,   2,    3,    2,    1,    3)
gender  <- c("f", "f", "f", NA, "m", "m", "m", "m")
degree[gender=="f"]
degree[gender=="m"]
table(degree)
table(gender)
summary(gender)
summary(degree)
summary(degree[gender=="m"])
```

- No! We need to tell R that these are categorical variables! This is important as this will
  - print the right statistics and summaries
  - automatically include dummy variables in regression model
- Note that NA is always included

No statistics, bar charts etc.; Es sind noch keine categorical variables festgelegt  
Character elements must be in quotes, missing value NA is not in quotes  
NA's is displayed! As I don't know, I keep it.  
R output is typically very sparse, optimized for further processing

# Factors

Better:

```
degree <- factor(degree)
degree
summary(degree)
degree <- factor(degree,
  levels=c(1,2,3,4),
  labels = c("BA", "MA", "PhD", "Other"))
degree
summary(degree)
gender <- factor(gender,
  levels = c("m", "f"),
  labels = c("Male", "Female"))
summary(gender)
degree[gender=="Female"]
degree[gender=="f"] #note this does not work anymore!
```

Note that values you do not include in `levels` become NA



Values do not have to actually occur, if for instance you collect data in different sources, but you still want to label.

# Data Frames

Why use data frames?

- data frames are rectangular set of variables
- variables are called components (vectors, factors, columns)
- observations are called rows or cases
- mode is list, class is `data.frame`, components must have equal length (same number of observations)
- variable names and row names are stored as attributes
- Almost never required, but...
  - lock values of observations together
  - ensures proper sorting
  - ensures correct NA removal

# Data Frames

```
testscores <- c("1.7", "1.3", "1.0", "1.7", "2.0")
mydata <- data.frame(degree, gender, testscores)
testscores <- c("1.7", "1.3", "1.0", "1.7", "2.0", NA,
               NA, NA)
mydata <- data.frame(degree, gender, testscores)
mydata
names(mydata)
rownames(mydata)
rownames(mydata) <- c("Bart", "Homer", "Maggie", "Marge",
                      "Nelson", "Apu", "Moe", "Krusty")
rownames(mydata)
mydata
class(mydata$testscores)
mydata$testscores <- as.numeric(as.character(mydata$
      testscores))
class(mydata$testscores)
```

- `data.frame` converts character variables to factors unless you add `stringsAsFactor = FALSE`

Changing classes as. coerce objects to change class temporarily when possible:  
as.vector, as.character, as.numeric, as.factor, as.data.frame, as.matrix, as.list

# Large Data Frames

For large data frames use `tbl_df` from `dplyr` package

- offers better printing of large data frames
- reports number of [rows by columns]
- prints only 10 observations (option can be changed)
- prints only enough variables to fill your screen
- class becomes `tbl_df`, `tbl`, `data.frame`
- affects `print()`

```
data(Titanic)
detach("package:dplyr")
print(data.frame(Titanic))
plot(data.frame(Titanic))
library(dplyr)
print(tbl_df(Titanic))
plot(Titanic)
detach("package:dplyr")
```

## Matrix

- same as data frame, but mode must be the same, i.e. atomic objects (all numeric or all character)
- class is matrix
- actually one long vector stored with a dimension attribute (dim)

## Array

- Matrix that may have more than two dimensions
- Vectors are 1D Arrays, Matrices are 2D Arrays
- actually one long vector stored with a dimension attribute (dim)

# Data Structures Overview

## Lists

- object that can store any other type of objects, called components
- `mylist <- list(name1 = comp1, name2 = comp2, ...)`

```
mylist <- list(degree, gender, testscores)
mylist
mylist <- list(UniversityDegree=degree,
              sex=gender,
              "Test Score"=testscores)
mylist
names(mylist)
identical(mylist[[1]], mylist$UniversityDegree)
identical(mylist[[2]], mylist$sex)
identical(mylist[[3]], mylist$'Test Score')
```

- modeling functions often output their values as lists
- for indexing we need double square brackets or names with \$ sign

## Some useful commands

- `print`
- `head`
- `tail`
- `names`
- `rownames`
- `mode`
- `class`
- `attributes`
- `str`



## Sorting and merging

- The `sort` command sorts (numeric or character) vectors
- By default, the elements are sorted ascendingly, but one can also sort descendingly.
- Matrices are sorted as vectors
- Dataframes cannot be sorted by `sort`
- The function `order(x)` returns a vector of the position of the smallest, the second smallest,  $\dots$ , the largest elements of `x`
- Hence, `x[order(x)]` returns the sorted vector
- The `order` command is useful for sorting matrices and dataframes!

## Sorting and merging

- Two dataframes can be merged by common column names
- The command `merge(x,y,by=...)` merges two dataframes `x` and `y` by a common variable given in the `by`-option
- What happens if there are observations in `x` that are missing in `y` (or vice versa)?
- There are options to choose the way R deals with missings

## **Data import and export**

---

## General remarks

- R is all about working with data
- There are various ways to read data from different sources in many formats
- In R, datasets are usually represented as `data.frame` objects
- R has a large collection of “standard datasets”, see `data()`

## Data import and export - Manual data input

- Very small datasets can be typed in directly, e.g.  
`x <- data.frame(v1=c(2,6,1,1),v2=c(9,9,8,8))`
- To edit existing objects, use `data.entry`, e.g.  
`y <- data.entry(x)`
- However, editing data within R is *not* recommended
- Datasets should be stored outside R, preferably in separate directories
- The datasets should be easily accessible by data-managing programs (e.g. Excel, Stata, ASCII editors, ...)

## Data import and export - Saving and loading R objects

- All R objects can be saved by the command  
`save(obj1,obj2,...,file="c:/path/name.Rdata")`
- In principle, other file name extensions are possible, but not recommended
- All objects saved in a file can be loaded by the command  
`load("c:/path/name.Rdata")`
- The data format is R specific

## Data import and export - Reading and writing text files

- A convenient command to read simple text files is `read.csv("c:/path/filename.txt")`
- The command assumes the following data format:
  1. The first row contains the variables names, delimited by commas
  2. The following rows are the observations, the variables are again delimited by commas
  3. The decimal sign is a dot (not a comma)
- Use `read.csv2` if the variables are delimited by semi-colons and the decimal sign is a comma (i.e. German style)
- More options are available for the command `read.table`
- Exporting text files from R is usually not necessary. If it is, use `write.csv`, `write.csv2` or `write.table`

## Exercise 7



## Data import and export - Other data formats

- there are many packages that provide easy access to datasets in other data formats
- flat files
  - `readr`: fast, easy to use, consistent
    - `read_delim` instead of `read.table`
    - `read_csv` instead of `read.csv`
    - `read_tsv` instead of `read.delim`
  - `data.table` for huge data sets
    - `fread` just works and ridiculously fast (infers column types and separators)
- Excel
  - `readxl` is fast
    - `read_excel("data.xlsx", sheet = "my_sheet")`
  - `XLConnect` to have much more control and bridge Excel into R
  - several other packages, e.g. `gdata` uses Perl, `xlsx` uses Java...
- Databases
  - `dbConnect` from DBI package

XLConnect is for big companies... installing is a bit hard, due to java,

# Data import and export - Other Statistical Software Packages

haven

- consistent, easy and fast (uses C library)
- SAS (`read_sas`), STATA (`read_stata` or `read_dta`), and SPSS (`read_por` or `read_sav`)

foreign

- less consistent, very comprehensive (saves everything into attributes),
- for formats dbf, Stata, SPSS, SAS, and a few more (but not Excel)
- `read.dta` takes also care of STATA's different missing values

haven: Stata 13 and 14

foreign: STATA 5 to 12, , might have to use saveold

Hmisc: `stata.get` for STATA with saveold

## **Exercises 8, 9, 10 and 11**

# **Selection and Transformations of Variables**

---

# Selecting Variables

Most programming packages:

- Select variables by name
- Select observations by logical condition

R can do that as well, but has many more ways to select and transform variables (we can even reverse this order)

## Indexing vectors

- R has a rich indexing syntax
- The basic ideas are the same for vectors, matrices and other objects
- Indexing is used to read or manipulate specified elements of the objects
- Indexes are always given in square brackets: `[]`  
(or sometimes `[][]`)
- Indexes can be either numerical or logical
- We will start with vectors and then look at matrices and dataframes
- The symbols `i` and `j` denote integer variables (not vectors)



# Indexing Vectors

## Numerical indexing

`x[1]` first element

`x[2]` second element

`x[i]`  $i$ -th element

`x[-i]` all elements, without position  $i$

`x[a:b]` all elements from position  $a$  to position  $b$

`x[k]`  $k$  numerical vector: all elements at positions given in  $k$

## Logical indexing

`x[a]` a logical vector: all elements where  $a$  is true  
( $a$  must have the same length as  $x$ )

# Indexing Vectors

## Indexing vectors

```
x <- c(2,3,4,5,1,1,9,3,2)
x[2]
x[4:7]
x[20]
x[-9]
x[-3]
x[c(1,5,1,9,9)]
a <- (x<4)
x[a]
x[x<4]
```

## Exercise 12

# Indexing Matrices

## Numerical indexing

`x[i,j]` element in row  $i$ , column  $j$

`x[:,j]` column  $j$  (as a vector)

`x[i,]` row  $i$  (as a vector)

`x[:,~j]` without column  $j$

`x[~i,]` without row  $i$

`x[a:b,j]` elements  $a$  to  $b$  in column  $j$

`x[k,m]`  $k, m$  numerical vectors: all elements at positions given in  $k$  and  $m$

# Indexing Matrices

Logical indexing Let  $a$  denote a logical matrix of the same dimension as  $x$ ;  
let  $k$  and  $m$  denote logical vectors of suitable length

$x[a]$  All elements of  $x$  at positions where  $a$  is true,  
as a *vector*!

$x[:,m]$  All columns of  $x$  where  $m$  is true

$x[k,]$  All rows of  $x$  where  $k$  is true

Of course, one may use numerical indexing for one dimension  
and logical indexing for the other dimension

$x[k,1:2]$  All elements of columns 1 and 2 where  $k$  is true

$x[3,m]$  All elements of row 3 where  $m$  is true

## Examples

```
x <- matrix(1:16,4,4)
x[3,3]
x[,4]
x[2,]
x[,-1]
x[-3,]
x[2:4,4]
x[c(1,4,2,2,2),1:2]
```

# Indexing Matrices

## Examples

```
x <- matrix((-7:8)^2,4,4)
a <- (x<10)
x[a]
x[,c(TRUE,FALSE,TRUE,FALSE)]
x[x[,1]<30,3:4]
x[x[,2]==1 | x[,3]==1,]
x[2:4,4]
x[c(1,4,2,2,2),1:2]
```

## Exercise 13



# Indexing Data Frames

- Dataframes have the same index methods as matrices
- Logical conditions can include strings (character variables)
- There are three additional ways to extract data frame columns:
  1. `x$varname`
  2. `x[[i]]`  
where `i` can also be a numerical vector
  3. `x["varname"]`  
or `x[c("varname1", "varname2", ...)]`
- Dataframe variables can be addressed directly by their name when you attach the dataframe, e.g. `attach(x)`
- `all`, `any` and `which` are also useful here

you can use the attach function, copies the variable into a temp space that R searches after your workspace they're available as vectors and factors Eventually detach(mydata) is optional to free up memory space, as this is only a copy! with(mydata,summary(q1)) now R looks in data frame BEFORE workspace but you need to do this for all commands (different to attach)

# Indexing Data Frames

## Common Error

```
mean(mydata["testscore"]) #will give you NA  
mean(mydata[, "testscore"]) #Don't forget the comma!
```

## Exercise 14

## select from dplyr package

- the select function makes life much easier as it selects all kinds of variables and always returns a data frame

```
library(dplyr)
select(mydata100, degree, gender) # for as many as
  I like
select(mydata100, gender:q4) # take all variables
  that are in between and itself too
select(mydata100, contains("q"))
select(mydata100, starts_with("q"))
select(mydata100, ends_with("shop"))
select(mydata100, num_range("q", 1:4))
```

- you can also use regular expressions
- be careful, most stat functions work on vectors, not on data frames

## Selecting observations

Put logic in the row position (before the comma)

```
summary(mydata100[mydata100$gender == "f", ]) #don't  
forget the comma!
```

you could actually index on different objects

## filter function from dplyr package

Use the filter() function

```
library("dplyr")  
summary(filter(mydata100, gender == "f"))
```

## Selecting both variables and observations

Traditional way:

```
myVars <- c("gender", "q1", "q2", "q3", "q4")
myObs <- which(mydata100$gender == "f")
mysubset <- mydata100[myObs, myVars]
summary(mysubset)
```

Modern way:

```
library("dplyr")
mysubset <- select(mydata100, gender, q1:q4)
mysubset <- filter(mysubset, gender == "f")
summary(mysubset)
```

- first call select or filter, whichever gives you the smallest subset
- advanced feature "pipes" %>%: feeds results from one to another



# Transformations

Very tedious:

```
mydata2[, "diff"] <- mydata[, "q4"] - mydata[, "q1"]
mydata2[, "ratio"] <- mydata[, "q4"] / mydata[, "q1"]
mydata2[, "q4log"] <- log(mydata[, "q4"])
mydata2[, "z4"] <- as.numeric(scale(mydata[, "q4"]))
mydata2[, "meanQ"] <- (mydata[, "q1"] + mydata[, "q2"]
  + mydata[, "q3"] + mydata[, "q4"])/4
```

Much cooler: mutate function from dplyr

```
mydata2 <- mutate(mydata,
  diff = q4 - q1,
  ratio = q4 / q1,
  q4log = log(q4),
  z4 = scale(q4),
  meanQ = (q1+q2+q3+q4)/4
)
mydata2
```

Making a copy to not overwrite dataset, don't forget commas!

## Exercise 15

# Graphics

---

## Some Remarks on Graphics

### Traditional or Base Graphics

- `plot()` offers many methods
- extremely flexible and extensible, but not easy to use with groups
- Uses “traditional graphics system”

```
load("mydata100.RData")
mydata100 <- na.omit(mydata100)
attach(mydata100)
head(mydata100)
plot(workshop)
plot(workshop,gender)
plot(gender,workshop)
plot(workshop,posttest)
plot(posttest,workshop)
plot(posttest)
plot(pretest,posttest)
hist(posttest)
rug(posttest)
```

plots of factors, these are standardized rather than counts

`plot(posttest,workshop)` is a strip plot

# Many options

Nicer plots

```
plot(pretest, posttest
     pch = 19, # plot character
     cex = 2,  # character expansion
     main = "My Main Title",
     xlab = "My X Axis Label",
     ylab = "My Y Axis Label")
grid() #add grid to plot
par() #graphics parameters
```

## Subplots

```
par(mfrow = c(2, 1)) # 2 rows, 1 column
plot(workshop[gender == "Female"], main = "The Females
")
plot(workshop[gender == "Male"], main = "The Males")
#scatter plot with regression line
par(mfrow = c(1, 1))
plot(pretest, posttest)
abline(c(18.78, 0.845))
myModel <- lm(posttest ~ pretest, data=mydata100)
abline(coefficients(myModel))
```

### Problems

- axis are not standardized
- a lot of white unnecessary space
- titles of each plot are in the vertical position taking valuable space
- one could fix all this with several options...



# ggplot - Basic idea

ggplot

- follows Wilkinson's Grammar of Graphics
- works with underlying graphics concepts, not pre-defined graph types
- enables to create any data graphic that you can think of
- uses Grid Graphics System instead of traditional system

Grammar of Graphics

- Aesthetics: how will variables appear? On axis? Shape, color, size...
- Geoms: set geometric objects, e.g. points, bars, lines, boxes
- Statistics: bins, smoother, fits,...
- Scales: axes (regular, log), legends
- Coordinate system: cartesian or polar
- Facets: plot by group(s)

how to create any kind of graphs

## ggplot - examples

```
library("ggplot2")  
ggplot(mydata100, aes(workshop)) +  
  geom_bar()  
  
ggplot(mydata100, aes(workshop), fill = gender) +  
  geom_bar(position="stack")  
  
ggplot(mydata100, aes(workshop), fill = gender) +  
  geom_bar(position="stack") +  
  scale_fill_grey()  
  
ggplot(mydata100, aes(workshop), fill = gender) +  
  geom_bar(position="dodge")  
  
ggplot(mydata100, aes(workshop), fill = gender) +  
  geom_bar() +  
  facet_grid(gender ~ .)
```

aes: asthetic, with very nice defaults

As an argument to this function, you specify a formula, where the factor before the ~ specifies the number of rows, while the factor after the ~ specifies the number of columns.

## ggplot - examples

```
ggplot(mydata100, aes(workshop, posttest)) +  
  geom_boxplot() +  
  geom_point()  
ggplot(mydata100, aes(workshop, posttest)) +  
  geom_boxplot() +  
  geom_point() +  
  facet_grid(. ~ gender)  
ggplot(mydata100, aes(pretest, posttest)) +  
  geom_points()  
ggplot(mydata100, aes(pretest, posttest, shape = gender)) +  
  geom_points(size = 5)  
ggplot(mydata100, aes(pretest, posttest, shape = gender,  
  linetype = gender)) +  
  geom_points(size = 5) +  
  geom_smooth(method = "lm")  
ggplot(mydata100, aes(pretest, posttest, shape = gender,  
  linetype = gender)) +  
  geom_point() +  
  geom_smooth(method = "lm") +  
  facet_grid(workshop ~ gender)
```

do analysis how equal are slopes, you get a graphic on how unequal the slopes are, you can check whether the ranges are similar

- layer will come on top
- there are no rows but gender in the columns

Same, more fully specified

```
ggplot() +  
  geom_point(data=mydata100, aes(pretest, posttest,  
    shape = gender)) +  
  geom_smooth(data=mydata100, aes(pretest, posttest),  
    method = "lm") +  
  facet_grid(workshop~gender)
```

- different layers can even point to different data sets!

so one data set could be raw data, another data set could be summary or confidence interval data



### Nicer ggplots

```
ggplot(mydata100, aes(pretest, posttest)) +  
  geom_point() +  
  labs(title = "Plot of Test Scores",  
        x = "Before Workshop",  
        y = "After Workshop") +  
  theme(plot.title = element_text(size=rel(2.5)))  
  
#Colors  
library(RColorBrewer)  
display.brewer.all(n=4) # how many column patterns  
ggplot(mydata100, aes(workshop, fill=gender)) +  
  geom_bar(position = "stack") +  
  scale_fill_brewer(palette = "Set1")
```

## Exercise 16

## **Data description**

---

## Frequency tables

- Let  $x = (x_1, \dots, x_n)'$  be a vector of (numerical or character) observations
- The command `table(x)` returns an object of the class “table”, representing the frequency distribution of  $x$
- The top row shows the values that occur (as a character vector)
- The bottom row shows the absolute frequencies
- The distribution can be plotted by `plot(table(x))` or `barplot(table(x))`
- try also `prop.table(table(x))` for relative frequencies or `cumsum(prop.table(table(x)))`

# Frequency tables

## Examples

```
myWG <- table(workshop,gender)
myWG
summary(myWG)
chisq.test(myWG)
```

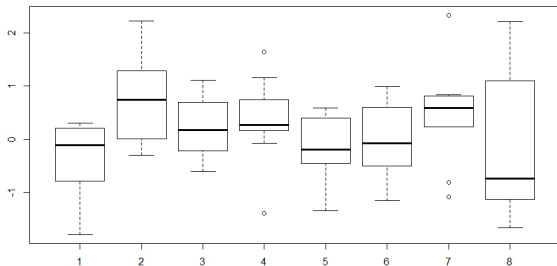
we can not reject the null that workshop and gender are independent

# Quantiles

- Quantiles can be computed by `quantile(x,prob=...)`
- The argument `prob` can be a scalar or a vector of probabilities
- If `prob` is a vector the `quantile` function returns a vector
- Note that there are many definitions of quantiles (see the option `type` of `quantile`)
- For large datasets, the differences are negligible

# Boxplots

- If the argument of `boxplot` is a vector, one boxplot is generated
- If the argument is a matrix (or data frame), one boxplot for each column is generated





## Mean, variance, standard deviation

- `mean` Calculates the mean of a vector, the mean of all elements of a matrix, each column mean of a dataframe
- `sd` Calculates the standard deviation of a mean, or the standard deviation of each column of a matrix or dataframe
- `var` Calculates the variance of a vector, or the covariance matrix of a matrix or dataframe
- `na.rm` All three functions have the option `na.rm` (remove missings) which can be `TRUE` or `FALSE` (default)

# Histograms

- The built-in command `hist` generates a plot of the histogram
- An improved command in the `library(MASS)` is `truehist`
- See the help file of `truehist` for the options
- Important options are: `xlab`, `ylab`, `xlim`, `ylim`, `main`
- One can easily add lines and curves to the plot  
(with `abline` or `lines`)

# Histograms

## Examples

```
library(MASS)
x <- rnorm(2000) # random data
truehist(x)
abline(v=0)
g <- seq(-3,3,length=500)
lines(g,dnorm(g))
```

## Exercise 17

# Covariance

- If there are two vectors  $x$  and  $y$  of the same length  $n$ , then `cov(x,y)` or `var(x,y)` compute the covariance
- If  $x$  is a  $(n, m)$ -matrix or dataframe, then `cov(x)` or `var(x)` compute the covariance matrix of its columns
- If missing values exist, one can specify which observations should be included (option `use`)

# Correlation

- If there are two vectors  $x$  and  $y$  of the same length  $n$ , then `cor(x,y)` computes the correlation coefficient (Bravais-Pearson)
- If  $x$  is a  $(n, m)$ -matrix or dataframe, then `cor(x)` computes the correlation matrix of its columns
- If missing values exist, one can specify which observations should be included (option `use`)
- Use the option `method` to compute Spearman's or Kendall's correlation coefficients

## Testing significance

```
cor(select(mydata100, q1:q4),  
method="pearson",  
use    ="pairwise")  
  
cor.test(mydata$q1, mydata$q4, use="pairwise")
```

first p-values, second p-values are adjusted corrected for the number of tests done.



## Exercise 18

## **Cleaning Data**

---

# Missing Values

- Missing values are neither negative nor positive infinity like in STATA
- Inf is infinity, also a kind of missing value, and you CAN do size comparison to it
- Finding missing values:
  - Not `x == NA` but `is.na(x)`
  - Counting missing values:

```
x <- c(NA,2,NA,2,1)
length(x)           #number of all variables
sum(is.na(x))       #number of missing values
sum(!is.na(x))      #number of valid values
```

- Hint: have a look at `n.valid()` from the `prettyR` package or write your own function:

```
n.missing <- function(x){
  sum(is.na(x))
}
```

R's logic is to have many tiny pieces,

# Missing Values

## Setting values to missing

- R reads numeric blanks as missing
- Remember: when creating factors, non-specified levels will become missing values
- When reading text files you can specify NA by option  
`na.string = c(".", "99", "999")`
- Better: use conditional transformations:  
`age[age == 999] <- NA`

na.strings affects all columns of your data set, and the missing values might be different for different values, so better use conditional transformations

## Action on Missing Values

- Summary functions return NA unless `na.rm=TRUE`
- Modeling functions (that accept formula) automatically exclude NAs
- Replacing/Imputing missing values
  - VIM (Visualization and Imputation of Missing Values): useful to find patterns in missing values and visualize them in color maps (`colormapMiss()`), bar charts (`barMiss()`) and histograms (`histMiss()`)
  - mice (Multivariate Imputation by Chained Equations):  
`md.pattern()` function also searches for patterns of missing values

## Exercise 19



## User-defined functions

---

# User-defined functions

- One can define new functions in R
- Functions are objects of class `function`
- Each function has a name, one or more inputs (arguments) and one output (return)
- Inputs can be any objects (usually vectors)
- The function can return only one object (which can be a list)
- Variables defined within a function are only local

similar to macros

# User-defined functions

## Syntax

```
fn <- function(x,y){  
  block of commands to compute output out  
  return(out)  
}
```

## Example

```
utility <- function(cons,gam){  
   $U <- (cons^{(1-gam)} - 1) / (1-gam)$   
  return(U)  
}
```

# User-defined functions

## Example

```
mystats <- function(x) {  
  mymean <- mean(x, na.rm=TRUE)  
  mysd    <- sd(x, na.rm=TRUE)  
  c(mean=mymean, sd=mysd) #only last thing is  
                           remembered  
}  
mystats(posttest)  
mymean #not found
```

- functions return only a single object, the last one created, but can contain many results
- applying functions by group

```
by(posttest, workshop, mean)  
by(posttest, workshop, mystats)
```

## Exercise 20

# Programming

---

# Loops

- If the same commands should be executed for different values of some variable, loops are useful
- There are three kinds of loops: `for`, `while`, `repeat`
- By far the most important loop is the `for`-loop
- General syntax:

```
for([var] in vector) {  
    [commands]  
}
```

- The commands are executed for each value of vector



# Loops

## Example

```
z <- rep(NA,10)
for(i in 1:10) {
  z[i] <- i^2
}
print(z)
```

# Loops

- Syntax of the while-loop:

```
while([condition]) {[commands]}
```

- Syntax of the repeat-loop:

```
repeat {[commands]}
```

- The repeat-loop does never stop but can be left using the command `break`

## Syntax of the if-command

```
if([condition]) {  
    [commands]  
}
```

- The condition must not be a vector (else only its first element is used)
- If there is just a single command, the brackets can be omitted
- The opening curly bracket must appear in the same line as the if-command
- It is possible to add else {[commands]}

# Random numbers

- A large number of standard distributions is implemented in R
- There is a common syntax for cdfs, density functions, quantile functions, and random number generation:

`pNAME(x,pars)` cumulative distribution function at  $x$

`dNAME(x,pars)` density (or probability) function at  $x$

`qNAME(p,pars)` quantile function at  $p$

`rNAME(n,pars)` generate  $n$  random draws

- Here NAME is the abbreviated name of the distribution and pars are its parameters

# Random numbers

Some continuous distribution names:

`norm` normal

`unif` uniform

`lnorm` log-normal

`exp` exponential

`t`  $t$ -distribution

`chisq`  $\chi^2$ -distribution

`F`  $F$ -distribution

# Random numbers

Some discrete distribution names:

`binom` binomial

`pois` Poisson

`geom` geometric

`hyper` hyper-geometric

`nbinom` negative binomial

`multinom` multinomial

# Random numbers

- Define a vector  $x$  on an appropriate grid  $[a, b]$
- Plots of cdf and density functions:

```
plot(x,pNAME(x, pars))
```

```
plot(x,dNAME(x, pars))
```

- Define a grid vector  $p$  on  $[0, 1]$ ; plot of quantile function:

```
plot(x,qNAME(p, pars))
```

# Simulations

Example: Simulate the distribution of the moment estimator of the exponential distribution

```
R <- 10000
Z <- rep(NA,R)
for(r in 1:R) {
  x <- rexp(n=10,rate=0.5)
  Z[r] <- 1/mean(x)
}
truehist(Z)
abline(v=2,col="red")
```



## **Exercises 21, 22, 23**

# Linear regressions

---

# Multiple linear regression

- The general syntax of regression models is rather idiosyncratic:

```
a <- lm(formula)
```

- Basic “formula” syntax

$$y \sim x_1 + x_2 + \dots + x_K$$

- Endogenous variable is on the left of  $\sim$ ; exogenous variables are on the right of  $\sim$ , separated by  $+$
- In R modeling functions: accept formulas, create model objects, generic functions show more, extractor functions show more

# Multiple linear regression

## Example

```
library(foreign)
x <- read.dta("wave2009.dta")
attach(x)
regr1 <- lm(satisfaction ~ age + netincome + children)
regr1
plot(regr1)
summary(regr1)
names(regr1)
print(unclass(regr1))
```

# Multiple linear regression

- The `lm`-object is a list containing:
  1. The estimated coefficients  $\hat{\beta}$
  2. The residuals  $\hat{u}_t$
  3. The fitted values  $\hat{y}_t$
  4. Some other things
- If `a` is an `lm`-object one can access its elements using `coefficients(a)`, `residuals(a)`, `fitted.values(a)`
- Alternatively, one can use the `$`-operator: `a$coefficients`, `a$residuals`, `a$fitted.values`

# Multiple linear regression

## Extensions (I):

- An intercept is added automatically but can be removed:  
`lm(y~x1+x2-1)`
- If the variables are organized in an unattached dataframe `x`, one can use the syntax: `lm(formula,data=x)`
- The formula may contain mathematical functions, e.g.  
`lm(log(y)~log(x1))`
- **Attention:** Squares, sums and differences are not allowed!
- Use the function `I()` for squares, sums and differences

# Multiple linear regression

## Extensions (II):

- Syntax for interaction terms

```
a <- lm(y ~ x1 + x2 + x1:x2)
```

- Abbreviations:
  - `a <- lm(y ~ x1*x2)`
  - `a <- lm(y ~ (x1+x2)^2)` for all interactions up to 2
- Weights can be added using the option `weights`
- One can select a subset of observations using the option `subset`

# Multiple linear regression

## Extensions (III):

- The `lm`-object can be used to add a regression line to a plot:

```
regr <- lm(y~x)
plot(x,y)
abline(regr)
```

- The `lm`-object can be used for forecasting:

```
regr <- lm(y~x1+x2)
xn <- data.frame(x1=c(...),x2=c(...))
predict(regr,newdata=xn,se.fit=TRUE)
```



# Multiple linear regression

Extensions (IV):

- Heteroskedasticity consistent standard errors are not reported by default
- The package `sandwich` supplies functions for robust standard errors
- The syntax for robust standard errors is

```
coeftest(regr,vcov=vcovHC)
```

## Multiple linear regression

Putting it all together:

```
regr2 <- lm(satisfaction~age+netincome,data=x)

regr3 <- lm(satisfaction~age+I(age^2))

regr4 <- lm(satisfaction~log(netincome))

regr5 <- lm(satisfaction~gender*marital)

z <- gender=="Female"

regr6 <- lm(satisfaction~log(netincome),subset=z)

coeftest(regr6,vcovHC)
```

# Polynomial regression

Polynomial regression

```
y ~ x + I(x^2) + I(x^3)
```

```
y ~ poly(x,3)
```

I means isolates this call

## Exercise 24

## High Quality Output

---

# High Quality Output

- Paste into word processor
- Use packages, e.g. xtable and texreg
- rtf or R2DOCX to write complex Word docs, but hard to set up
- Reproducible research: knitr and rnotebook

## Example

```
myM1 <- lm(q4 ~ q1 + q2 + q3, data=mydata100)
myM2 <- lm(q4 ~ q1, data=mydata100)

library("xtable")
print(xtable(myM1), type="html", file="myM1-xtable.doc")

library("texreg")
htmlreg(myM1, single.row=TRUE, file="myM1-htmlreg.doc")
htmlreg(list(myM1, myM2), file="myM1-myM2-htmlreg.doc")

texreg(list(myM1, myM2))
```