Introduction to R

Mandy Vogel

University Leipzig

October 5, 2015

Overview

Table of Contents I

Objects

	Homogeneous	Heterogeneous
1d	Vector	List
2d	Matrix	Data frame
nd	Array	

Important function to inspect the structure of an object:

- `str()`

Vectors

- logical, integer, numeric (double), character
- `c()`, `length()`
- cœrcion while combining vectors of different types
- factors for coding categorical variables

Lists

- can contain everything
- can be arbitrarily complex
- created by `list()`, concatenated by `c()`

Data Frames

- common data table
- `names()` to get column names
- `dim()`, `nrow()`, `ncol()` to get the number of rows and columns

Indexing/Subscripting

- we need square brackets
- for a one-dimensional object (e.g. a vector) we need one index `vector[position]`, for a two-dimensional two `matrix[row,column]`, etc
- you can index use integers (position), logical values (conditions), or characters (names) for indexing
- when a subscript appears as a blank it is understood to mean all of
  - `[,4]` means all rows in column 4 of an object
  - `[2,]` means all columns in row 2 of an object.

Indexing - Exercises

First try to understand the following commands:

Input

```
> x <- c(2, 7, 0, 9, 10, 23, 11, 4, 7, 8, 6, 0)
> x[4]
> x[3:5]
> x[c(1, 5, 8)]
> x[x > 10]
> x[(1:6) * 2]
> x[x == 0] <- 1
> x
> ifelse(round(x/2) == x/2, "even", "odd")
```

Indexing - Exercises

Now try the following:

1. Display every third element in x
2. Display elements that are less than 10, but greater than 4
3. Modify the vector x, replacing by 10 all values that are greater than 10
4. Modify the vector x, multiplying by 2 all elements that are smaller than 5
5. Create a new vector y with elements 0,1,0,1, . . . (12 elements) and a vector z that equals x when y=0 and 3x when y=1. (You can do it using ifelse, but there are other possibilities)

Indexing - Exercises I

Now load the `pima` data set which is contained in the `faraway` package (probably you have to install the package; then load the package; afterwards the data can be loaded by typing `data(pima)`)

To find out a bit more about `pima` try

Input

```
> help(pima) ## or ?pima
> names(pima)
> head(pima)
```

Input

```
> pima[1, "glucose"]
```

This will list the value taken by the first subject for the bweight variable. Alternatively

Indexing - Exercises II

Input

```
> pima[1, 2]
```

will list the value taken by the first subject for the second variable (which is bweight). Similarly

Input

```
> pima[2, "diastolic"]
```

will list the value taken by the second subject for bweight, and so on. To list the data for the first 10 subject for the bweight variable, try

Input

```
> pima[1:10, "bmi"]
```

and to list all the data for this variable, try

Indexing - Exercises III

```
> pima[, "bmi"]
```

To list the data for the first subject on all variable except the second try

Input

```
> pima[1, -2]
```

Exercises - Indexing

1. Display the data on the variable `age` for row 7 in the `pima` data frame.
2. Display all data in row 7.
3. Display the first 10 rows of the data on the variable `triceps`.

Table of Contents I

Reading Data

The most convenient way of reading data into R is via the function called `read.table()`. It requires that the data is in "ASCII format", or a "flat file" as created with Windows' NotePad or any plain-text editor. The result of `read.table()` is a data frame.

It is expected that each line of the data file corresponds to a subject information, that the variables are separated by blanks or any other separator symbol (e.g., ",", ";"). The first line of the file can contain a header (`header=T`) giving the names of the variables, which is highly recommended!

read.table()

As an example we read in the data contained in the file `fishercats.txt`

Input/Output

```
> read.table("week1/data/fishercats.txt",
+               sep=" ",header=T)
  Sex Bwt Hwt
1   F 2.0 7.0
2   F 2.0 7.4
3   F 2.0 9.5
4   F 2.1 7.2
5   F 2.1 7.3
....
```

These data correspond to the heart and body weights of samples of male and female cats (R. A. Fisher, 1947).

read.table()

The first argument corresponds to the data file, the second to the fields separator and the third `header=T` specifies that the first line is a header with variable names. Important: the character variables will be automatically read as factors.

There is a variant for reading data from an url:

Input/Output

```
> winer <- read.table(
+ "http://socserv.socsci.mcmaster.ca/jfox/Courses/R/ICPSR/Wine
+ header=T)
```

`read.table()`

There are other variants of `read.table` function alike :

- `read.csv()` this function assumes that fields are separated by a comma instead of whites spaces
- `read.csv2()` this function assumes that the separate symbol is the semicolon, but use a comma as the decimal point (some programs, e.g., Microsoft Excel, generate this format when running in European systems)

`read.table()` - Arguments

- `header` - indicates if the first row contains column names
- `sep` - field separator
- `dec` - decimal point
- `na.strings` - strings are used to code missing values
- `skip` - number of lines to skip before beginning to read in the data

There are many more, check the help if needed

Reading data from the clipboard

With the function `read.delim()` or also `read.table()` it is possible to read data directly from the clipboard.
For example mark and copy some columns from an Excel spreadsheet and transfer this content to an R by

Input/Output

```
> mydata <- read.delim("clipboard",na.strings=".")
> str(mydata) # structure of the data
```

Reading Data from Other Programs

You can always use the export function from other (statistical) software to export data from other statistical systems to a tab or comma-delimited file and use the `read.table()`. However, R has some direct methods.

The `foreign` package is one of the "recommended" packages in R. It contains routines to read files from SPSS (`.sav` format), SAS (export libraries), EpiInfo (.rec), Stata, Minitab, and some S-PLUS version 3 dump files. For example

Input/Output

```
> library(foreign)
> mydata <- read.spss("test.sav", to.data.frame=T)
```

read the `test.sav` SPSS data set and convert it to a data.frame.

Reading Data from Excel Files

Input/Output

```
> library(XLConnect)
> setwd("/media/TRANSCEND/mpicbs/data/")
> my.wb <- loadWorkbook("Duncan.xls")
> sheets <- getSheets(my.wb)
> content <- readWorksheet(my.wb, sheet=1)
> head(content)
        Col0 type income education prestige
1 accountant prof     62        86       82
2      pilot prof     72        76       83
3  architect prof     75        92       90
4     author prof     55        90       76
5    chemist prof     64        86       90
6   minister prof     21        84       87
>
```

Reading Data from Excel Files

- whereas `XLConnect` is the most sophisticated R package to read and write Excel files it depends on java and is therefore a bit clumsy
- the relatively new package `readxl` dœs not depend on java nor on perl
- up to now two commands: `excelsheets()`, `read_excel()`

Input/Output

```
> require(readxl)
Lade nötiges Paket: readxl
> x <- read_excel("week1/data/Duncan.xls")
> head(x)
          NA type income education prestige
1 accountant prof     62        86       82
2      pilot prof     72        76       83
3  architect prof     75        92       90
4     author prof     55        90       76
```

Reading Data from Excel Files

If someone is really fond of Excel, RExcel (http://rcom.univie.ac.at/download.html) is really worth the effort. There is also a function reading MSAccess files (`mdb.get()` from the Hmisc package)

Something on Connections

The function `read.table()` opens a connection to a file, read the file, and close the connection. However, for data stored in databases, there exists a number of interface packages on CRAN.

The RODBC package can set up ODBC connections to data stored by common applications including Excel and Access (for Excel and Access RODBC dœsn't work on Unix but it is great for data base connections). There are also more general ways to build connections to data bases.

For up-to-date information on these matters, consult the "R Data Import/Export" manual that comes with the system.

Read Data - Exercises

1. read the file `T0_22029_39_20130503_946-empatom.csv` using `read.table()` with the suitable arguments (omit the first three lines, use the third line as names for the columns); do not forget to assign the data to a variable
2. how many rows
3. how many columns
4. how many answers were right and how many wrong (`accuracy = 1`; `accuracy = 0` resp)

dir()

- `dir()` without additional argument shows all files/directories in the working directory

Input/Output

```
> dir()
 [1] "auto"                "contentRintro.aux"   "contentRintro.
 [4] "contentRintro.tex~"  "data"                "#meins.r#"
 [7] "#rstuff2.r#"         "session1.tex"        "session2.aux"
[10] "session2.log"        "session2.nav"        "session2.out"
[13] "session2.pdf"        "session2.snm"        "session2.tex"
[16] "session2.toc"        "session2.vrb"
```

dir()

- given a path `dir()` will show the content of resp folder

Input/Output

```
> dir("../logfiles")
[1] "T0_02411_39_20140724_1328-empatom.csv"
[2] "T0_02544_39_20140808_811-empatom.csv"
[3] "T0_03858_39_20140626_1504-empatom.csv"
[4] "T0_04517_39_20130723_1344-empatom.csv"
[5] "T0_09458_39_20130522_1133-empatom.csv"
[6] "T0_09754_39_20140717_1509-empatom.csv"
[7] "T0_10120_39_20130423_1112-empatom.csv"
[8] "T0_10698_39_20130425_1114-empatom.csv"
...
```

dir()

Input/Output

- setting `recursive` to `TRUE` R will recurse into directories recursively through

```
> dir("..",recursive = T)
   [1] "data/empatom_trialdata.csv"
   [2] "empatom-1.R"
   [3] "flights.jpg"
   [4] "logfiles/T0_02411_39_20140724_1328-empatom.csv"
   [5] "logfiles/T0_02544_39_20140808_811-empatom.csv"
   [6] "logfiles/T0_03858_39_20140626_1504-empatom.csv"
   [7] "logfiles/T0_04517_39_20130723_1344-empatom.csv"
   [8] "logfiles/T0_09458_39_20130522_1133-empatom.csv"
   [9] "logfiles/T0_09754_39_20140717_1509-empatom.csv"
...
```

dir()

- setting `full.names` to `TRUE` R will give the full path

Input/Output

```
> dir("../logfiles", full.names = T)
  [1] "../logfiles/T0_02411_39_20140724_1328-empatom.csv"
  [2] "../logfiles/T0_02544_39_20140808_811-empatom.csv"
  [3] "../logfiles/T0_03858_39_20140626_1504-empatom.csv"
  [4] "../logfiles/T0_04517_39_20130723_1344-empatom.csv"
  [5] "../logfiles/T0_09458_39_20130522_1133-empatom.csv"
  [6] "../logfiles/T0_09754_39_20140717_1509-empatom.csv"
  [7] "../logfiles/T0_10120_39_20130423_1112-empatom.csv"
  [8] "../logfiles/T0_10698_39_20130425_1114-empatom.csv"
...
```

dir()

- with `pattern` we can specify which files to show (regexpr), e.g. all r files or all file names containing the string 25502
- you can combine more the one pattern using the pipe symbol (or)

Input/Output

```
> dir("../logfiles", pattern = "25502")
[1] "T0_25502_39_20140527_824-empatom.csv"
[2] "T1_25502_39_20140826_902-empatom.csv"
[3] "T2_25502_39_20141117_1053-empatom.csv"
[4] "T3_25502_39_20150217_844-empatom.csv"
```

`dir()` Exercise

- create a variable containing the names of all text files in the data directory where the names contain
  - 2013 as year
  - T2 as time point
  - 25502 or 11944 as id

Table of Contents I

`apply()`

- now we know how to read in a data file and
- how to get vector containing file names
- we need a function which takes the file names and the `read.table()` function and read in all the files at once

Implicit Loops

A common application of loops is to apply a function to each element of a set of values and collect the results in a single structure.
In R this is done by the functions (but there is of course also `for()`):

- `lapply()`
- `sapply()`
- `apply()`
- `tapply()`

lapply()

Input/Output

```
> lapply(mtcars,mean)
$mpg
[1] 20.09062

$cyl
[1] 6.1875

$disp
[1] 230.7219

$hp
[1] 146.6875
...
```

lapply()

Input/Output

```
> sapply(mtcars,mean)
       mpg        cyl       disp         hp       drat
 20.090625   6.187500 230.721875 146.687500   3.596563   3.217
        vs         am       gear       carb
  0.437500   0.406250   3.687500   2.812500
```

apply()

- `apply()` this function can be applied to an array. Its argument is the array, the second the dimension/s where we want to apply a function and the third is the function. For example

Input/Output

```
> x <- 1:12
> dim(x)<-c(2,2,3)
> apply(x,3,quantile) #calculate the quantiles
     [,1] [,2]  [,3]  #for each 2x2 matrix
0%   1.00 5.00  9.00
25%  1.75 5.75  9.75
50%  2.50 6.50 10.50
75%  3.25 7.25 11.25
100% 4.00 8.00 12.00
```

tapply()

- The function `tapply()` allows you to create tables (hence the "t") of the value of a function on subgroups defined by its second argument, which can be a factor or a list of factors. For example in the `quine` data frame, we can summarize `Days` classify by `Eth` and `Lrn` as follows:

Input/Output

```
> tapply(mtcars$mpg,mtcars$cyl,mean)
       4        6        8
26.66364 19.74286 15.10000
> tapply(mtcars$mpg,list(mtcars$cyl,mtcars$vs),mean)
         0      1
4 26.00000 26.730
6 20.56667 19.125
8 15.10000     NA
```

Apply Family - Exercises

1. from the data frame read in above (from the subject 22029, T0) calculate the mean response time `response_time` for each of the categories of `accuracy` using `tapply()`
2. find also the min, max, and median
3. use `lapply()` on the data frame in combination with `class()` to get the column types of the data frame

lapply() and read.table()

1. now use `dir()` to get all files from id 22111, set the `full.names` argument to true; store them in a vector called `files`
2. then use `lapply()` in combination with `read.table()`

Input

```
> files <- dir("../logfiles", pattern = "22111", full
> data.list <- lapply(files,read.table,header = T, sk
```

3. repeat but now reading in addition the files from id 25507, 25508, and 25509
4. what is the length of the resulting list? how to interpret the length?

Reduce()

- now we have separate data frame
- what we want to have is one data frame
- so the data frame have the same columns (how to check?)
- what is the right command to combine? (we had seen last week `rbind()`, `cbind()` and `merge()`)

`Reduce()`

- is a higher order function (functional)
- `Reduce()` uses a binary function (like `rbind()` or `merge()`) to combine successively the elements of a given list
- it can be used if you have not only two but many data frames

`Reduce()`

- `Reduce()` takes a binary function as first argument and a list containing the objects to be combined as the second argument so

  ```
  > res <- Reduce(rbind,data.list)
  ```

  should combine the 16 data frames into one

Reduce()

- additional arguments to the function could not be given to `Reduce()`, you have to modify to function itself instead
- as example we can change the column names (not recommended in real analyses)

```
> res <- Reduce(function(x,y){
+     names(y) <- names(x)
+     rbind(x,y)
+ }, data.list)
> res <- Reduce(function(x,y){
+     names(x) <- names(y) <- paste0("col",1:ncol(x))
+     rbind(x,y)
+ }, data.list)
```

`readLines()`

- now we have almost everything we need but some important information is contained in the first two lines
- `readLines()` can be used to read in a file line by line (without any conversion)
- the result is a line by line version of the file like a text
- this text can be passed through to `read.table()`
- so we can read line one and two as a first table and line three to the end as second

readLines()

Input

```
> filename <- files[1]
> xx <- readLines(filename)
> head(xx)
[1] "subject:\ttimepoint:\tdate:\tlist:\tno_trials:\tname_duration:\tquestio

[2] "22111_39\tT0\t05/30/2013 10:12:57\tT0_22111\t48\t2000\t14000\t1000\t400

[3] "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t

[4] "measurement\tfirst_pulse\ttrial\tjitter_1\tjitter_2\tjitter_3\tcategory
[5] "T0_22111\t9982\t1\t1400\t3000\t1200\t1\t0\t0\tKlara\tKlara1.avi\t243\t3
```

readLines()

Input

```
> d1 <- read.table(text = xx[4:length(xx)],
+                    sep = "\t",header=T)
> d2 <- read.table(text = xx[1:2],sep = "\t",header=T)
```

Table of Contents I

Reading the Data

Input/Output

```
> tmp <- lapply(files,function(filename){
+        xx <- readLines(filename)
+        d1 <- read.table(text = xx[4:length(xx)],fill = T,hea
+        d2 <- read.table(text = xx[1:2],fill = T,header=T)
+        d1$subject <- rownames(d2)
+        d1$timepoint <- d2$subject
+        d1$date <- d2$timepoint
+        d1$time <- d2$date
+        d1$no.trials <- d2$no_trials
+        return(d1)
+    })
>
> system.time(result <- Reduce(rbind,tmp))
Fehler in match.names(clabs, names(xi)) :
  Namen passen nicht zu den vorhergehenden Namen
Timing stopped at: 0.01 0 0.01
```

Reading the Data

Input/Output

```
> tmp <- lapply(files,function(filename){
+         xx <- readLines(filename)
+         d1 <- read.table(text = xx[4:length(xx)],fill = T,hea
+         names(d1)[3] <- "trial"
+         d2 <- read.table(text = xx[1:2],fill = T,header=T)
+         d1$subject <- rownames(d2)
+         d1$timepoint <- d2$subject
+         d1$date <- d2$timepoint
+         d1$time <- d2$date
+         d1$no.trials <- d2$no_trials
+         return(d1)
+     })
>
> system.time(result <- Reduce(rbind,tmp))
       User      System verstrichen
     57.812       0.017      57.765
```

Table of Contents I

Introduction

The dplyr package makes each of these steps as fast and easy as possible by:

- Elucidating the most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem.
- Providing simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code.
- Using efficient data storage backends, so that you spend as little time waiting for the computer as possible.

filter()

- Base R approach to filtering forces you to repeat the data frame's name
- dplyr approach is simpler to write and read
- Command structure (for all dplyr verbs):
  - first argument is a data frame
  - return value is a data frame
  - nothing is modified in place
- Note: dplyr generally dœs not preserve row names

filter() example

```
> require(dplyr)
> prob22029 <- filter(result, subject == "22029_39")
> table(prob22029$subject)

22029_39
     192
> prob22029 <- filter(result, subject == "22029_39",
+                     accuracy == 1)
> table(prob22029$subject,prob22029$accuracy)

            1
  22029_39 116
```

`select()`

- `select()` selects colums
- it can be used in combination with `filter()`
- combining can be done by the infix $\% > \%$

select() example

```
> subframe <- select(result, measurement, first_pulse, subject)
> nrow(subframe)
[1] 47610
> subframe <- filter(result, response_time < 5000) %>%
+     select(measurement, first_pulse, subject)
> nrow(subframe)
[1] 3183
```

arrange()

- arrange can be used to change the order of rows

```
> head(result[,c("name","measurement","timepoint","response_time"]
       name measurement timepoint response_time
1     Silke    T0_02411        T0          6033
2      Lina    T0_02411        T0          4561
3      Paul    T0_02411        T0          9373
4 Alexandra    T0_02411        T0          6290
5     Peter    T0_02411        T0          7339
6      Mona    T0_02411        T0          6329
```

arrange()

```
> arr.frame <- arrange(result, response_time)
> head(arr.frame[,c("name","measurement","timepoint","response_ti
       name measurement timepoint response_time
1 Stephanie    T0_02544        T0             0
2    Tobias    T0_02544        T0             0
3     Achim    T0_03858        T0             0
4    Sabine    T0_04517        T0             0
5   Richard    T0_09458        T0             0
6      Lina    T0_09458        T0             0
```

arrange()

```
> arr.frame <- arrange(result, response_time) %>%
+     filter(response_time > 0)
> head(arr.frame[,c("name","measurement","timepoint","response_ti
     name measurement timepoint response_time
1 Kathrin    T1_22074        T1             1
2 Carolin    T2_25505        T2            32
3   Klaus    T3_22035        T3            37
4    Lara    T3_25406        T3            46
5   David    T1_22047        T1           108
6     Tim    T1_25396        T1           123
```

mutate()

- `mutate()` can be used to transform or add columns
- you can add/change more than one column at once

mutate() example

```
> subframe <- filter(result,subject == "22074_39") %>%
+     mutate(video2 = str_replace(video, "\\.avi", ""),
+            video3 = str_replace(video2, "[0-9]", ""),
+            concern_time = concern_time_ended - concern_time_started )
>
```

mutate() example

```
> table(subframe$subject)

22074_39
     192
> summary(subframe$concern_time)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2090000 -1542000 -1045000 -1044000  -535100   -32480
```

mutate() example

```
> table(subframe$video3)

     Achim Alexandra   Anahita      Anna     Anton      Aziz    Birgit   Carolin
         4         4         4         4         4         4         4         4
 Charlotte Christoph    Daniel     David     Felix      Gabi      Gerd Hannelore
         4         4         4         4         4         4         4         4
    Hannes    Holger    Isabel     Ivko   Juliane   Kathrin     Klara      Lara
         4         4         4         4         4         4         4         4
     Laura      Lena       Leo      Lina     Marie    Marius    Markus  Matthias
         4         4         4         4         4         4         4         4
   Michael      Mona    Nabeel  Natalie      Olga      Paul     Peter   Richard
         4         4         4         4         4         4         4         4
    Sabine   Sabrina     Silke Stephanie    Thomas       Tim    Tobias       Uwe
         4         4         4         4         4         4         4         4
```

summarise()

- summarise() makes summary statistics

```
> sumframe <- summarise(result,
+                       right.perc = sum(accuracy == 1)/
+                       mean.resp.time = mean(response_t
> head(sumframe)
  right.perc mean.resp.time
1  0.6379752       8275.904
```

summarise() example

- `summarise()` gets really interesting in combination with `group_by()` also included in the `dplyr` package

```
> sumframe <- group_by(result, subject) %>%
+     summarise(right.perc = sum(accuracy == 1)/n(),
+               mean.resp.time = mean(response_time, na.rm = T))
> head(sumframe)
Source: local data frame [6 x 3]

   subject right.perc mean.resp.time
1 00436_39  0.6388889       7974.889
2 02411_39  0.7500000       7048.104
3 02544_39  0.6354167       9079.635
4 03858_39  0.7552083       9031.745
5 04517_39  0.7916667       8727.469
6 09458_39  0.7083333       7214.573
```

summarise() example

```
> sumframe <- group_by(result, subject, timepoint) %>%
+     summarise(right.perc = sum(accuracy == 1)/n(),
+               mean.resp.time = mean(response_time, na.rm = T))
>
> head(sumframe)
Source: local data frame [6 x 4]
Groups: subject

   subject timepoint right.perc mean.resp.time
1 00436_39        T1  0.5208333       7061.958
2 00436_39        T2  0.7708333       8296.083
3 00436_39        T3  0.6250000       8566.625
4 02411_39        T0  0.7291667       7209.417
5 02411_39        T1  0.7708333       6886.792
6 02544_39        T0  0.7083333      10043.083
```

summarise() example

```
> sumframe <- group_by(result, subject, timepoint) %>%
+     summarise(right.perc = sum(accuracy == 1)/n(),
+               mean.resp.time = mean(response_time, na.rm = T)) %
+     arrange(right.perc,desc(mean.resp.time))
> head(sumframe)
Source: local data frame [6 x 4]
Groups: subject

   subject timepoint right.perc mean.resp.time
1 00436_39        T1  0.5208333       7061.958
2 00436_39        T3  0.6250000       8566.625
3 00436_39        T2  0.7708333       8296.083
4 02411_39        T0  0.7291667       7209.417
5 02411_39        T1  0.7708333       6886.792
6 02544_39        T1  0.5625000       8116.188
```

summarise() example

```
> sumframe <- group_by(result, subject, timepoint) %>%
+     summarise(right.perc = sum(accuracy == 1)/n(),
+                  mean.resp.time = mean(response_time, na.rm = T)) %
+     ungroup() %>%
+     arrange(right.perc,desc(mean.resp.time))
> head(sumframe)
Source: local data frame [6 x 4]

   subject timepoint right.perc mean.resp.time
1 22127_39        T1  0.2916667       7052.812
2 22076_39        T0  0.3125000       8594.125
3 25507_39        T0  0.3125000       8396.375
4 22086_39        T0  0.3125000       8078.333
5 22028_39        T3  0.3125000       7882.042
6 22062_39        T0  0.3125000       7457.458
```

dplyr Exercises

1. use `select()` and `filter()` in combination (%>%) to select all rows belonging to the T0 or the T3 test, keep subject, timepoint, accuracy, response_button, response_time, affect_time_ended and concern_pos_started column. Create a new data frame with a appropriate name.
2. add two new variables containing the counts of each of the possible values of accuracy. Use `mutate()` and `sum(accuracy==1)`.
3. use `group_by()` and `summarise()` to extract the minimum and maximum response_time per person from the original data frame
4. repeat the last exercise, but now group per person and time point

Table of Contents I

Structure of a ggplot Object

begin with an empty object to see the structure:

```
> po <- ggplot()
> summary(po)
data: [x]
faceting: facet_null()
```

Structure of a ggplot Object

```
> str(po)
  List of 9
  $ data       : list()
  ..- attr(*, "class")= chr "waiver"
  $ layers     : list()
  $ scales     :Reference class 'Scales' [package "ggplot2"] with 1 fields
  ..$ scales: NULL
  ..and 21 methods, of which 9 are possibly relevant:
  ..  add, clone, find, get_scales, has_scale, initialize, input, n,
  ..  non_position_scales
  $ mapping    : list()
  $ theme      : list()
  $ coordinates:List of 1
  ..$ limits:List of 2
  .. ..$ x: NULL
  .. ..$ y: NULL
  ..- attr(*, "class")= chr [1:2] "cartesian" "coord"
  $ facet      :List of 1
  ..$ shrink: logi TRUE
  ..- attr(*, "class")= chr [1:2] "null" "facet"
  $ plot_env   :<environment: R_GlobalEnv>
  $ labels     : list()
  - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

Structure of a ggplot Object

Now we fill this structure - first the three main steps:

- the first argument to ggplot is `data`
- then specify what graphics shapes you are going to use to view the data (e.g. `geom_line()` or `geom_point()`).
- specify what features (or `aesthetics`) will be used (e.g. what variables will determine `x`- and `y`-locations) with the `aes()` function
- if these æsthetics are intented to be used in all layers it is more convenient to specify them in the `ggplot` object

Feed the Object

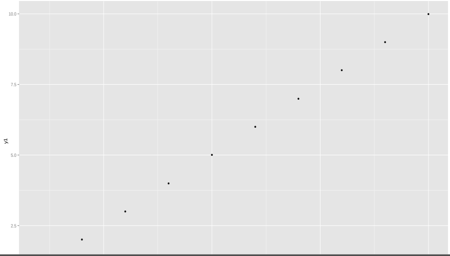
- first we create a little sample data frame

```
> x1 <- 1:10; y1 <- 1:10; z1 <- 10:1
> l1 <- LETTERS[1:10]
> a <- 10; b <- (0:-9)/10:1
> ex <- data.frame(x=x1,y=y1,z=z1,l=l1,a=a,b=b)
> ex
    x  y  z l  a          b
1   1  1 10 A 10  0.0000000
2   2  2  9 B 10 -0.1111111
3   3  3  8 C 10 -0.2500000
4   4  4  7 D 10 -0.4285714
5   5  5  6 E 10 -0.6666667
6   6  6  5 F 10 -1.0000000
7   7  7  4 G 10 -1.5000000
8   8  8  3 H 10 -2.3333333
9   9  9  2 I 10 -4.0000000
10 10 10  1 J 10 -9.0000000
```

Feed the Object

- then create a ggplot object containing the data and some standard æsthetics (here we define the x and the y positions)
- add one or more geoms, we begin with geom_point

```
> po <- ggplot(ex,aes(x=x1,y=y1))
> summary(po)
data: x1, y1, z1, l, a, b [10x6]
mapping:  x = x1, y = y1
faceting: facet_null()
> p1 <- po + geom_point()
```



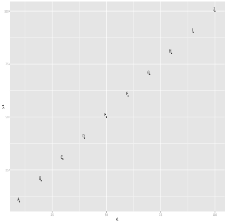
Layers

- `ggplot()` creates an object - every "+" adds something to this object (change the object)
- the default method of ggplot() is `print()`, which creates the plot
- it is better to store the object - so you can change it (e.g. you can change the data frame)

Layers

- so we add another layer, which adds a label to the points (use geom_text)

- `aes(label=l)` maps the `l` variable to the `label` æsthetic, and `hjust` and `vjust` define where our labels are placed



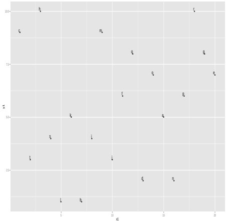
Layers

- imagine you have worked a little time on a plot - and then you detect a mistake in your data, so the real data frame looks different
- so you can replace the old, wrong data by the new data (using %+%

```
> ## the new data
> ex2 <- data.frame(x1=sample(1:20),
+                   y1=sample(1:10),
+                   l=letters[1:20])
> head(ex2,10)
   x1 y1 l
1   3  6 a
2   6  2 b
3  14  1 c
4  19 10 d
5  12  4 e
6  15  8 f
7  20  5 g
8  17  7 h
```

Layers

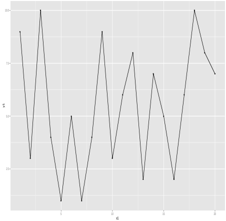
```
> p2 %+% ex2
```



Layers

- by using the line geom you can join the points (we use the new data)

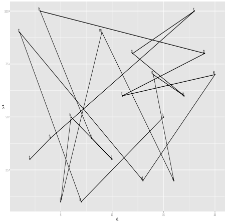
```
> pn <- p %+% ex2 ## replace data in p
> pn + geom_line()
```



Layers

- you can also join the points in the order of the data frame by using the path geom instead

```
> my.text <- geom_text(aes(label=l),
+                                  hjust=1.1,
+                                  vjust=-0.2)
> pn + geom_path() + my.text
```

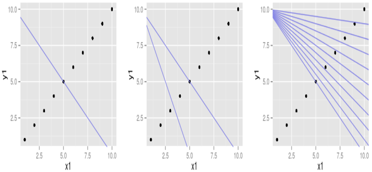


Layers

Adding extra lines:

- there are three geoms: `abline`, `vline`, `hline`
- `abline` adds one or more lines with specified slope and intercept to the plot

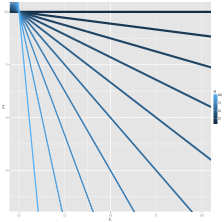
```
> ## one line
> p + geom_abline(intercept=10,slope=-1,
+                              colour=rgb(.5,.5,.9))
> ## two lines
> p + geom_abline(intercept=c(10,9),slope=c(-1,-2),
+                                  colour=rgb(.5,.5,.9))
> more lines
> p + geom_abline(intercept=10:1,slope=-(10:1)/10,
                                      colour=rgb(.5,.5,.9))
```



Layers

- adding lines referring to the data frame

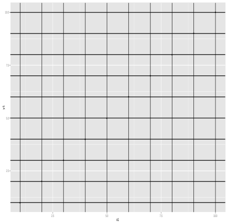
```
> p1 +
+   geom_abline(aes(slope=b,intercept=a,colour=x1)) +
+   scale_x_continuous(limits=c(0,10))
```



Layers

- the same works for the hline and the vline geom which add horizonal and vertical line(s)
- argument: yintercept, xintercept respectively
- setting and mapping are possible

```
> p1 + geom_hline(yintercept=1:10)
> p1 + geom_hline(yintercept=1:10) +
+      geom_vline(xintercept=1:10)
```



Other Common Layers

- some other layers for 1 continuous variable:
  - `geom_boxplot()`
  - `geom_histogram()`
  - `geom_density()`
- some other layers for 1 discrete variable:
  - `geom_bar()`
- some other layers for 2 or more continuous variables:
  - `geom_smooth()`
  - `geom_density2d()`
  - `geom_contour()`
  - `geom_quantile()`