TDS Facility

Introduction to R for flow cytometry data analysis Day 1

João Lourenço, Tania Wyss & Nadine Fournier

Translational Data Science – Facility

SIB Swiss Institute of Bioinformatics

With slides from Diana Marek, Thomas Junier, Wandrille Duchemin, Leonore Wigger

Outline & Schedule

Day 1 (afternoon)

03

Getting started with R syntax and objects
Exercises
(13:00 – 15:30)

15:30 -15:50 Coffee break

04

Importing, formatting and exporting data with R
Exercises
(15:50 – 16:50)

16:50 - 17:00 Feedback and end of day

03 Getting started with R syntax and objects

R Basic Data Types

- Variables we have seen so far can hold one value. This value can be of different types. Use mode() to display it.

The three most common data types:

- **Numeric**:
  - A number stored with decimal point. (Decimal point need not be displayed).
    Examples: 0, 5, 55.2, -11.111
    - in some contexts this data type is labeled "double“
    - integers, stored without decimal points, exist but are rarely used.

- **Character:**
  A text sequence. Must be enclosed in quotes " ". (Single quotes work, too).
  Examples: "1a++", "Hello World", "s", "99"

- **Logical:**
  TRUE or FALSE (This is binary. No other possible values).

R Syntax

Syntax refers to the spelling and "grammar" rules of a programming language.

A few important points :

- Case sensitive: R differentiates between small letters and capitals.
- Statements can be separated by a newline or by a semicolon ";" (for better readability, a newline after each statement is almost always preferable)
- Long statements can be written on multiple lines
- R has no strict rules about including or omitting blank spaces between elements, as long as the code is unambiguous. Make your spacing consistent and think of readability.

The # character stands for comments. Anything after a # on a line is ignored by R. Write comments into your code to explain what it does.

R Objects

An object is a storage space that takes (or contains) a value, a data structure or a section of code.

- All elements of an R statement can be thought of as objects.
  Variables are objects containing data.
  Functions are objects containing code.

Allowed Names for Objects

Object names can consist of letters, numbers, dots and underscores.

- Cannot start with a number.
- Cannot contain operators (including hyphen).
- Cannot start with underscore
- Best to start with letter

Valid examples:

x

mydata1

mydata.normalized

n_times

The Assignment Operator "<-" (or equivalent: "=")

We can use either the symbol "**<-**" or "**=**" to assign values to objects. Stick to one for consistency.

- Create an object:

```
> x <- 10     # Create object x, assign the 10 to it
  value # NB: This does the same as x = 10
```

- Change the value of an existing object:

```
> x <- 25    # x has the value 10; overwrite it
```

- Set one object to equal the value of another object:

```
> myNumber <- 15
> x <- myNumber  # Both x and myNumber now contain 15
```

- Modify the content of an object:

```
> x <- x + sqrt(16)  # add the square root of 16 to x
```

Using Functions (I)

- Functions are called with parentheses () after the function name

- Arguments are the input to functions, passed inside the ()

```
> ls()        # no argument - list objects in workspace
> sqrt(81)    # one argument - square root of the input
> rep(1,5)    # two arguments - repeat the number 1, 5 times
```

- Arguments have names (specified in the function definition). Function calls can be made with unnamed or named arguments or a mix of both. Use "=" for named arguments.

```
> rep(x=1, times=5)    # Named args. Equivalent to rep(1,5)
> rep(1, times=5)      # Mixed. Equivalent to rep(1,5)
```

Check R help (?function_name) to see which arguments are expected by a function.

Using Functions (II)

Many functions take more than one argument

- If unnamed, arguments must be listed in correct order (association by position).
- If named, arguments can be passed in arbitrary order (association by name).

```
> write.table(object, "outfile.txt", TRUE)

> write.table(object,  append=TRUE,  file="outfile.txt")
```

Unnamed arguments: must appear in their correct position
Named arguments: their position does not matter

Using Functions (III)

Some functions have arguments with default values.

Example: function **round()**

```
Usage (from R Help):   round(x, digits = 0)
```

Arguments with default values can be omitted in the function call; the default value is then used. Arguments without default values cannot be omitted.

```
> round(2.011)     #rounding to 0 digits after decimal point
[1] 2              # (default value)

>round(2.011, 2) #rounding to 2 digits after decimal point
[1] 2.01
```

Using Functions (IV)

Using and understanding the help/documentation is 50% of what makes a programmer!

- Look up the help page, try the examples, experiment

```
?paste
?"^"
```

Also, internet is your friend:

Google "R paste function"
"R how to ..."

https://stackoverflow.com/questions/tagged/r

Let's practice – 3

For all exercises, feel free to use

- cheat sheets on internet or provided on the online document

- R help (? at command prompt)

Open a new script file and save it as ex3.R

1) Assign the values 6.7 and 56.3 to variables **a** and **b**, respectively.

2) Calculate (2*a)/b + (a*b) and assign the result to variable **x**.
Display the content of **x**.

3) Find out how to compute the square root of variables.
Compute the square roots of **a** and **b** and of the ratio **a/b**.

4) a) Calculate the logarithm to the base 2 of **x** (i.e., $\log_2 x$).
b) Calculate the natural logarithm of **x** (i.e., $\log_e x$).

Common Object Classes

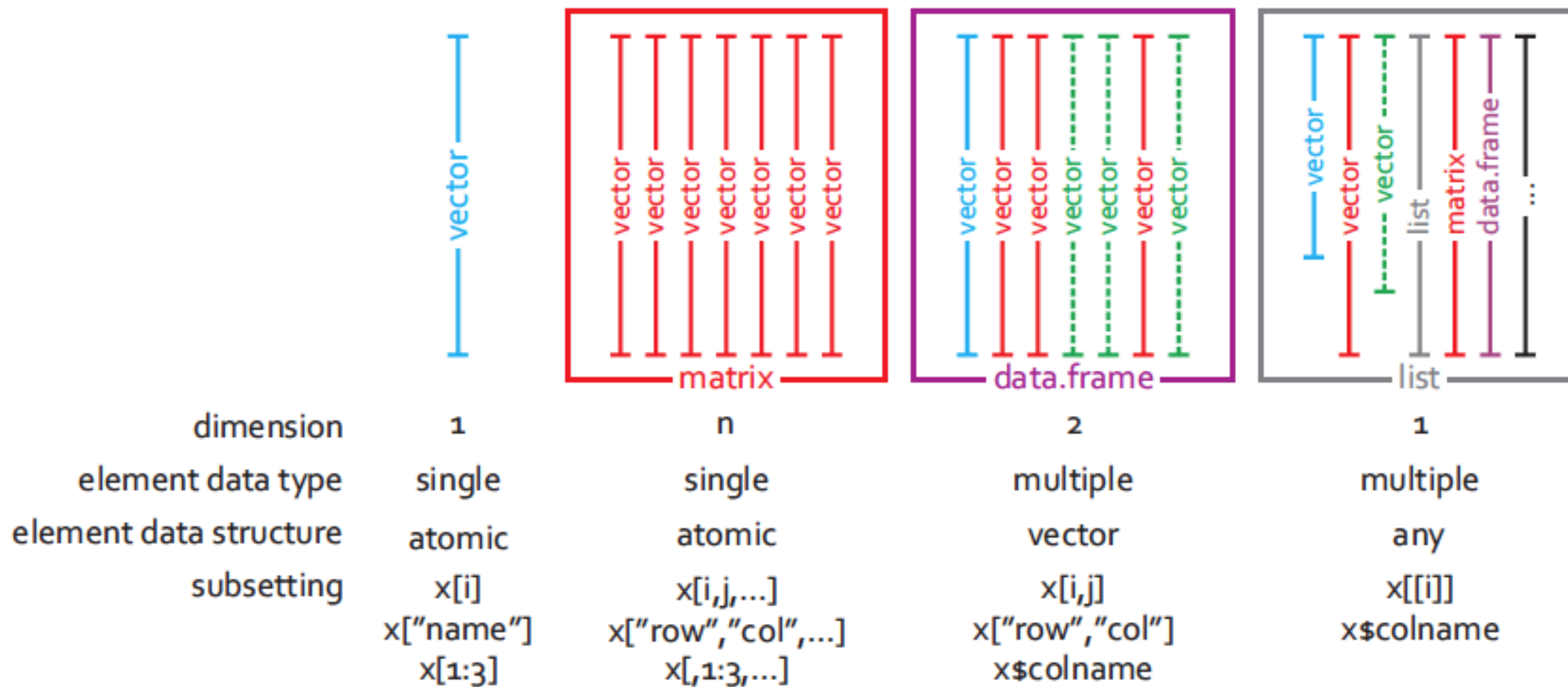
- **vector** – a series of data, all of the same type
- **matrix** – multiple columns of same length, all must have the same type of data
- **data frame** – multiple columns of same length, can be mix of data types
- **list** – a collection of objects; can be of different classes and different sizes

Data objects

- **function** – a command to perform a specific task

Function objects

Graphical View on Data Object Classes



From M. Stadler

Creating Objects: Vectors

Vector: A series of data, all of the same type

- Create a vector using c()

```
height_in_cm <- c(180, 167, 199) # c() stands for concatenate
```

- Create a vector using c() where each element has a name

```
height_in_cm <- c(Mia=180, Paul=167, Ed=199)
```

- Access elements of a vector using [ ]

```
height_in_cm[1]         # get the first element
height_in_cm[c(1,3)]   # get the 1st and 3rd element
height_in_cm["Paul"]   # get the element named "Paul"
```

> Scalars in R (the simple variables we have seen so far) can be thought of as vectors of length 1.

Creating Objects: More Ways to Generate Vectors

- Vectors of defined length with default value

```
> numeric(4); character(4); logical(4)
```

- **:** (colon operator)

```
> a <- 1:10
```

- **seq()** for sequences with any step size

```
> s <- seq(4,10,2) #start at 4, end at 10, step by 2
```

- **rep()** for vectors with repeating elements

```
> genotypes <- c(rep("WT",3), rep("KO",3))
```

- **Trick** : Use **[]** to extract repeated elements

```
> tplayer <- c("Federer", "Nadal")
> tplayer[c(1,1,1,2,2,1)] #3x Federer, 2x Nadal, 1x Federer
```

Vector Manipulation (I)

Applying operators to vector results in element-wise operations

```
 > a
  [1]  1  2  3  4

> a * 2          # multiply each element of [1]  a by 2
  2    4  6  8

> a + c(12,10,12,10) # add the elements in 2 vectors
  [1]  13  12  15  14
```

Vector Manipulation (II)

Many functions take a vector as argument.

Some perform an element-wise operation. Example:

```
> log2(a) # compute the logarithm in base 2 of each element
    [1] 0.000000 1.000000 1.584963 2.000000
```

Some return a single value. Example:

```
> mean(a) # compute mean of the elements
  [1] 2.5
```

Coercion

- All elements of a vector must be of the same type
- If combining different types, they will be coerced to the most flexible type
  - least to most flexible are: logical < numeric < character

Example:

```
> vec <- c(12, "twelve", TRUE) #combine 3 data types
> vec                           #all are coerced to character

[1] "12" "twelve" "TRUE"



> class(vec)

[1] "character"
```

Coercion (II)

- We can coerce an existing vector to another type using the functions **as.logical(), as.numeric(), as.character()**.

- Example: Coerce a logical vector to numeric

  Values are converted to 1 (for TRUE) and 0 (for FALSE)

  - we can use **as.numeric()** for explicit coercion
  - we can use mathematical functions on logical vectors, coercion to numeric happens automatically

```
> x <- c(FALSE, FALSE, TRUE)
> as.numeric(x)
> sum(x)       # number that are true
> mean(x)      # proportion that are true
```

Factors

- A **factor** is a vector containing values from a limited set; used for storing **categorical data**.

- Example: Genotype of mouse individuals

```
> genotype <- factor(c("WT", "WT", "Mut2", "Mut1", "Mut2"))
```

The available values in a factor are called levels. Extract them:

```
> levels(genotype)
[1] "Mut1" "Mut2" "WT"
```

- Convert the factor back to a character vector:

```
> geno <- as.character(genotype)
```

Factors with Custom Sorted Levels

- By default, factor levels are sorted alphabetically.
- We can specify a different sorting with the argument levels.

- Example: Genotype of mouse individuals

```
> genotype <- factor(c("WT", "WT", "Mut2", "Mut1", "Mut2"),
                     levels = c("WT", "Mut1", "Mut2"))
```

```
> levels(genotype)
[1] "WT" "Mut1" "Mut2"
```

Levels are sorted the way we wanted

Let’s practice - 4

1) Create two vectors, **vector_a** and **vector_b**, containing the values from −5 to 5 and from 10 down to 0, respectively.

2) Calculate the (element-wise) sum, difference and product between the elements of **vector_a** and **vector_b**.

3) a) Calculate the sum of elements in **vector_a**.

   b) Calculate the overall sum of elements in both **vector_a** and **vector_b**.

4) a) Identify the smallest and the largest value in **vector_a**

   b) among both **vector_a** and **vector_b**.

5) Compute the overall mean of the values among both **vector_a** and **vector_b**.

Hint: Each task in exercises 1-5 can be performed in a single statement per vector (the minimum and maximum count as 2 tasks)

Operators (Most Commonly Used Ones)

- **Arithmetic**

+, -, *, /, ^

- **Comparison**

>, <, <=, >=, == (equal to), != (not equal to)

- **Logical**

! (negation), & (AND), | (OR)

- **Other**

%in% (in operator)

Comparisons, logical operators and %in% always return logical values! (TRUE, FALSE)

Operators returning logical values: examples

```
> c(1,3,2) == 2
[1] FALSE FALSE TRUE

> !(c(1,3,2) < 2)
[1] FALSE  TRUE  TRUE

> table(!(c(1,3,2) < 2))
#FALSE
TRUE #2 1

> c("Fred", "Marc", "Dan", "Ali") %in%
  c("Dan", "Geoff", "Ali")
[1] FALSE FALSE TRUE  TRUE
```

Missing Values

- R distinguishes between
  - NA (not available)
  - NaN (not a number, e.g. result of 0/0)

- Use the functions **is.na()** and **is.nan()** to detect them.

Missing Values: Examples (I) NA

Missing values are usually represented by NA:

```
> y <- c(1,2,3,4,5,NA,NA)
```

NA's interfer with many functions:

```
> mean(y)
[1] NA
```

Arguments often exist to remove NA's before calculation

```
> mean(y, na.rm=TRUE)
[1] 3
```

Alternatively, use **na.omit()** to remove NAs from the data

```
> y_cleaned <- na.omit(y)
> mean(y_cleaned)
[1] 3
```

Missing Values: Examples (II)

```
> x <- c(1, NA, 0/0) ; x # a vector to play with
[1] 1 NA NaN

> is.na(x) #detects NAs and NaNs from x
[1] FALSE TRUE TRUE

> is.nan(x) # detects only NaNs from x
[1] FALSE FALSE TRUE

> x > 2  # what if we try to compare NA and NaN to a number?
[1] FALSE NA NA

> x[!is.na(x)]  # removes NAs and NaNs from x
[1] 1
```

Creating Objects: Data Frames

data frame: multiple columns of same length, can be mix of data types

```
> name <- c("Joyce", "Chaucer", "Homer")
> status <- c("dead", "deader", "deadest")
> reader_rating <- c(55, 22, 100)
```

Create a data frame using the function **data.frame()**

```
>poets <- data.frame(name, status, reader_rating)
>poets
```

```
     name   status reader_rating
1   Joyce     dead            55
2 Chaucer   deader            22
3   Homer  deadest           100
```

Creating Objects: Lists

List: a collection of objects; can be of different classes and different sizes

Create a few objects:

```
> vec <- c(0.4, 0.9, 0.6)
> mat <- cbind(c(1,1), c(2,1))
> df <- data.frame(name=c("Ed", "Lisa"), age=c(61, 71))
```

Unnamed list - collect these objects in a list, using the function **list()**:

```
> l <- list(vec, mat, df)
```

Named list - collect these objects in a list with named elements:

```
> l_with_names <- list(myvec=vec, mymatrix=mat, mydata=df)
```

Detecting Data Types and Object Classes

The function **class()** is useful when we are not sure what kind of object we are dealing with.

- for vectors, returns the basic data type of its elements ("numeric", "character", "logical", ...)

    similar to mode() but slightly more fine-grained
    - recognizes "integer" as different from "numeric"
    - recognizes factors (categorical variables)

- for all other objects covered on previous slide, returns their class ("matrix", "data.frame", "list", "function", ...)

Accessing Data Elements

matrix:

```
>m[2, 2]      # gets the element on row 2 in column 2
>m[1:3, ]     # gets rows 1,2,3
>m[, c(1,4)] # gets columns 1 and 4
```

data frame:

```
>poets[2, 2]       # gets the element on row 2 in column 2
>poets[, c(1,3)]   # gets columns 1 and 3
>poets[, c("name", "reader_rating")] # gets columns "name"
                                    #  and "reader_rating"
>poets$name        # gets column "name"
```

list:

```
>l[[1]]                         # gets the first object
>l_with_names[["myvec"]]# gets the object named "myvec"
>l_with_names$myvec      # gets the object named "myvec", too
```

Accessing Names of Data Elements

matrix and data frame:

```
>rownames(poets)    # gets the row names
>colnames(poets)    # gets the column names

>rownames(poets) <- c("J", "C", "H") # overwrites row names
```

list:

```
>names(l_with_names) # gets the list elements' names
>names(l_with_names) <- c("A", "B", "C") # overwrites names
```

Let's practice – 5

Open a new script and save it as "Ex5.R". Comment it.

1) In your script, write the command to load the package "MASS".
2) Write the following command to load the bacteria data set from the package MASS:

```
data(bacteria) # loads the bacteria data set (from MASS)
```

Execute the command. Check: You should have a variable named "bacteria" in your Environment.

3) What are the names of the columns of the **bacteria** data.frame ?
4) Use **[]** to select rows 100 to 119 of the column "ap" .
5) Use $ to get the column "week" and check how many missing values it has.

Optional : 6) Count how many rows correspond to a "placebo" treatment ("trt" column) using the comparison operator "==".

In a Nutshell

- Everything in R is an object.
- Using R is all about creating and manipulating data objects using functions (which are themselves objects).
  - Objects can be assigned to a name
  - Objects have a class (data frame, matrix, list etc)
  - Data values inside objects have different data storage modes (numeric, character, logical)
- We covered many ways to generate data (create objects).
- Now, let's import some data !

04 Importing, Formatting and Exporting data with R

Prepare Your Data Outside of R

Before using R and importing the dataset you collected from an experiment, you need to know how to format it properly, so R can read it.

A spreadsheet program such as Excel or OpenOffice can be used for data entry and simple manipulation.

Three precepts of tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

http://www.ucd.ie/ecomodel/pdf/TidyData.pdf

Example of Well-Formatted Dataset

	A	B	C	D	E	F
1	sample_id	file_name	patient id	collection_date	age	gender
2	LC01	Tube_001_CD45+ cells.fcs	P0228	22.01.20	71	male
3	LC03	Tube_003_CD45+ cells.fcs	P0113	04.02.20	73	female
4	LC04	Tube_004_CD45+ cells.fcs	P0248	19.02.20	60	male
5	LC06	Tube_006_CD45+ cells.fcs	P0255	26.02.20	75	male
6	LC07	Tube_007_CD45+ cells.fcs	P0256	27.02.20	68	female
7	LC08	Tube_008_CD45+ cells.fcs	P0071	02.03.20	73	male
8	LC09	Tube_009_CD45+ cells.fcs	P0258	05.03.20	68	male
9	LC10	Tube_010_CD45+ cells.fcs	P0261	10.03.20	56	male
10	LC12	Tube_012_CD45+ cells.fcs	P0279	28.04.20	84	male
11	LC13	Tube_013_CD45+ cells.fcs	P0280	28.04.20	69	male

- A header line with variable names
- 5 variables, one in each column
- One observation per row

Formatting Recommendations – Checklist

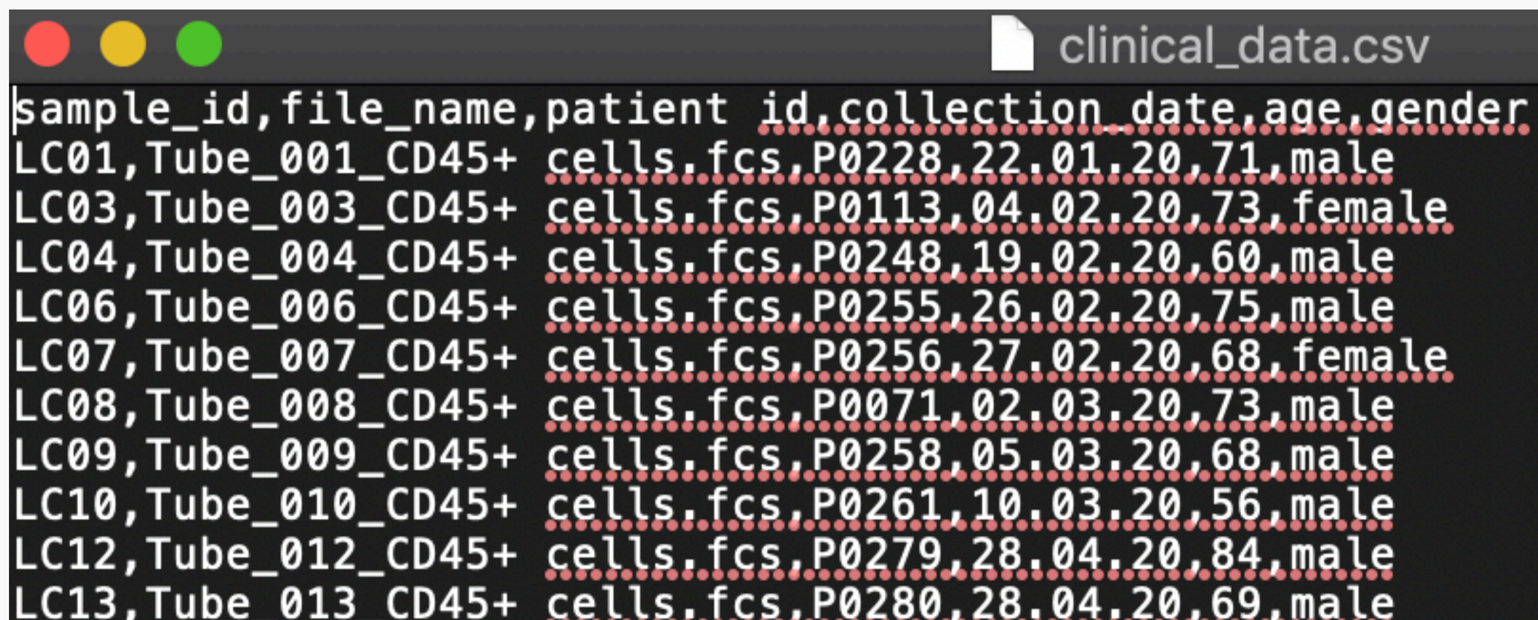
- If you work with spreadsheets, the first row is usually reserved for the header.
- The first column may or may not be an ID column.
- Remove blank spaces from column names and in fields. If you want to concatenate words, insert a "_" between words.
- Avoid column names containing symbols other than "_".
- Short names are preferred over longer names.
- Delete any comments or other content in the spreadsheet that are not part of the data table but are above, below or beside the data table.
- Make sure that any missing values in your data set are indicated with NA. (Check spelling! N.A. or n.a. does not work.)

Other Recommendations

- If you’re using a spreadsheet, keep a copy of the original data as it was provided to you. Prepare a new, "cleaned" version for your data analysis.
- Do not include columns that you do not need for your analysis.
- Have data backups!

Saving Your Data

- Export the spreadsheet to your computer in a text file format:
  - csv (comma separated values) format, with file extension .csv OR
  - tsv (tab separated values) format, with file extension .txt or .tsv



clinical_data.csv

```
sample_id,file_name,patient id,collection_date,age,gender
LC01,Tube_001_CD45+ cells.fcs,P0228,22.01.20,71,male
LC03,Tube_003_CD45+ cells.fcs,P0113,04.02.20,73,female
LC04,Tube_004_CD45+ cells.fcs,P0248,19.02.20,60,male
LC06,Tube_006_CD45+ cells.fcs,P0255,26.02.20,75,male
LC07,Tube_007_CD45+ cells.fcs,P0256,27.02.20,68,female
LC08,Tube_008_CD45+ cells.fcs,P0071,02.03.20,73,male
LC09,Tube_009_CD45+ cells.fcs,P0258,05.03.20,68,male
LC10,Tube_010_CD45+ cells.fcs,P0261,10.03.20,56,male
LC12,Tube_012_CD45+ cells.fcs,P0279,28.04.20,84,male
LC13,Tube_013_CD45+ cells.fcs,P0280,28.04.20,69,male
```

Keep your data safe:

Have a back up!

Importing Data

- Most widely used R base functions for data import: **read.table() , read.csv()** and **read.delim()**
    - reads a formatted text file
    - imports it as a data frame
    - **many** options, to accommodate most text files (e.g., csv, tsv).

- To read an entire data frame from a file, it should have:

1. a header line containing the names of all variables

   -> (not obligatory but preferable)

2. one line per row, with values for each variable

   -> (missing values should be indicated using NA)

3. Items must be separated by the same separator symbol

   -> (most common: , ; \t)

Importing Data – Two Questions

- Where is the file I want to import?
  - Look for your file in the file system.
  - Note its path: the succession of folders to access it

- Where is my working directory?
  - use **getwd()** **(recommendation: should be the project directory)**

File Paths in R

File paths can be specified as a string with **'/'** as separator:

```
"C:/Users/Leo/courses/data/clinical_data.csv"
```

Or with a little help from the function **file.path():**

```
file.path("C:", "Users", "Leo", "courses", "data",
"clinical_data.csv")
```

WARNING:

On Windows : replace '\' by '/'

File Paths in R - Relative

R understands "." and ".." for relative file paths

. is the current directory (=working directory)

.. is the parent directory

```
"./course_datasets/clinical_data.csv"  # file "clinical_data.csv" in subfolder "course_dataset" of current directory
"../../clinical_data.csv"   # file "clinical_data.csv", 2 levels up from current directory
```

Also works with file.path():

```
file.path("..", "..", "clinical_data.csv")
```

Importing Data – File Paths

read.csv() needs to know where the file is located.

- **Data file is in the working directory:** file name suffices.

  ```
  read.csv("clinical_data.csv“)
  ```

- **Data file is in a sub-folder of working directory:** It's easy to use a relative path. (Great option for projects shared with others).

- read.csv("course_datasets/clinical_data.csv") or read.csv(file.path(”course_datasets", "clinical_data.csv"))

- **Data file is somewhere else or you are not working inside a project:** it's safest to use an absolute path (but can be more painful to specify!).

  ```
  read.csv("C:/Users/Leo/courses/data/clinical_data.csv”)
  ```

Importing Data

Important optional arguments of **read.table(), read.csv(), read.delim()**

- **header** (TRUE/FALSE): specifies whether the first line contains column names
  Default in read.table() is FALSE.
  Default in read.csv() and read.delim() is TRUE.

- **sep:** specifies the field separator character (e.g. "," or tab "\t").
  Default in read.table() is any white space characters (space, tab, newline and carriage return).
  Default in read.csv() is comma. Default in read.delim() is tab.

- **colClasses**: manually setting each variable data type

When in doubt, use **help(read.table)**

The file can be imported as a data frame using the functions **read.table() or read.csv()**

Use read.table()

```
# we need to supply certain arguments
Clinical_data <- read.table("course_datasets/clinical_data.csv",
                            sep=",”,
                            header=TRUE)
```

Use read.csv()

```
# arguments can be  omitted since defaults are adapted to reading .csv
Clinical_data <- read.csv("course_datasets/clinical_data.csv")
```

Checking the Imported Data

- It is very important to check that data you asked R to import is the data you wanted.
- **head()** returns the first 6 lines of the data frame
- **dim()** returns the dimension of the data frame
- **nrow(), ncol()** returns the number of row and columns
- **colnames()** and **rownames()** functions return the column and row names of the data frame
- **str()** returns the structure of the data frame
- **summary()** is a generic function that can be applied to many types of objects. For data frames, it returns:
  - Numeric columns: min, max, median, mean, 1st and 3rd quantiles.
  - Factors columns: counts of each factor level

```
>head(clinical_data) # shows first 6 rows
(tail(clinical_data) - shows last 6 rows)

  sample_id collection_date age gender stage
1      LC01        22.01.20  71   male   III
2      LC03        04.02.20  73 female    IV
3      LC04        19.02.20  60   male    II
4      LC06        26.02.20  75   male   III
5      LC07        27.02.20  68 female    II
6      LC08        02.03.20  73   male     I

>dim(clinical_data)
[1] 30  5

>nrow(clinical_data); ncol(clinical_data)
[1] 30
[1] 5
```

```
> colnames(clinical_data)  # column names
```

```
[1] "sample_id"       "collection_date" "age"
[4] "gender"          "stage"
```

```
> str(clinical_data) # structure of the data frame
```

```
'data.frame':   30 obs. of  5 variables:
 $ sample_id      : chr  "LC01" "LC03" "LC04" "LC06" ...
 $ collection_date: chr  "22.01.20" "04.02.20" "19.02.20" "26.02.20"
 ...
 $ age            : int  71 73 60 75 68 73 68 56 84 69 ...
 $ gender         : chr  "male" "female" "male" "male" ...
 $ stage          : chr  "III" "IV" "II" "III" ...
```

R made its best guess for data types.
- Are they what we need?
- Do we wish to convert any variables to factors?

Setting factor variables

Convert categorical variables to factors as needed.

```
> clinical_data$gender <- factor(clinical_data$gender)
> clinical_data$stage <- factor(clinical_data$stage,
  levels = c("I","II","III","IV"))
```

```
> str(clinical_data) # structure of the data frame
```

```
'data.frame':	30 obs. of  5 variables:
 $ sample_id      : chr  "LC01" "LC03" "LC04" "LC06" ...
 $ collection_date: chr  "22.01.20" "04.02.20" "19.02.20" "26.02.20" ...
 $ age            : int  71 73 60 75 68 73 68 56 84 69 ...
 $ gender         : Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2 2 2 2 ...
 $ stage          : Factor w/ 4 levels "I","II","III",..: 3 4 2 3 2 1 4 3 2 2 ...
```

```
> summary(clinical_data)
```

```
  sample_id          collection_date          age            gender      stage  
 Length:30          Length:30          Min.   :56.00   female: 9   I   : 6  
 Class :character   Class :character   1st Qu.:62.00   male  :21   II  :11  
 Mode  :character   Mode  :character   Median :68.00               III : 6  
                                       Mean   :67.67               IV  : 5  
                                       3rd Qu.:72.75               NA's: 2  
                                       Max.   :84.00                        
```

Accessing Parts of the Data

```
> clinical_data[2,] # 2nd row
```

```
  sample_id collection_date age gender stage
2      LC03        04.02.20  73 female    IV
```

```
> clinical_data[, "age"]   # column named «age»
```

```
 [1] 71 73 60 75 68 73 68 56 84 69 70 70 67 57 62 72 61 74 57 65 68 62 75 65
[25] 61 71 62 76 77 61
```

```r
> clinical_data$stage # vector of stages, equivalent to
  clinical_data[, 5]
```

```
 [1] III IV  II  III II  I   IV  III II  II  IV  II  II  II  II  III II  I  
[19] II  IV  III II  III IV  III I   I   I   III I  
Levels: I II III IV
```

```r
> clinical_data$stage[30] # stages of the last row
```

```
[1] I
Levels: I II III IV
```

Subsetting the Data

- subset() is a powerful function which allows you to subset your data by specific columns and values in those columns. Logical operators can be used within the subset.

```
> subset(clinical_data, stage=="II") # keeps only the
  samples where stage is "II"
   sample_id collection_date age gender stage
3       LC04        19.02.20  60   male    II
5       LC07        27.02.20  68 female    II
9       LC12        28.04.20  84   male    II
10      LC13        28.04.20  69   male    II
12      LC16        03.06.20  70 female    II
13      LC17        23.07.20  67   male    II
14      LC18        23.07.20  57   male    II
15      LC19        29.07.20  62   male    II
17      LC22        08.09.20  61 female    II
19      LC26        19.11.20  57   male    II
22      LC29        10.12.20  62   male    II
```

```
> subset(clinical_data, stage=="II" & gender=="female")
  # keeps samples from female patients in stage II
```

```
   sample_id collection_date age gender stage
5       LC07        27.02.20  68 female    II
12      LC16        03.06.20  70 female    II
17      LC22        08.09.20  61 female    II
```

```
> subset(clinical_data, (stage=="I" | stage=="II") &
  gender=="female") # keeps samples from female
  patients in stages I or II
```

```
   sample_id collection_date age gender stage
5       LC07        27.02.20  68 female    II
12      LC16        03.06.20  70 female    II
17      LC22        08.09.20  61 female    II
28      LC36        09.07.20  76 female     I
```

Customising Summaries of Data

tapply() generates custom summaries of your data using :

- X: a column you want to aggregate (of any data type)
- INDEX: a factor column, or list of factor columns, for grouping
- FUN: a function to be applied to X (mean, sd, min, max, length, median, range, quantiles…), separately for each grouping indicated by INDEX

```
> tapply(X=clinical_data$age,
  INDEX=clinical_data$stage, FUN=min)

 I  II III  IV
61  57  56  65
```

In each stage, find the age from the youngest patient (min)

Data Reshaping : Adding Rows and Columns

- Rows and columns of data can be added using the functions **rbind()** and **cbind()**, respectively.

- Add a row to the clinical data:

```
> clinical_updated <- rbind(clinical_data,
  data.frame(sample_id = "LC02", collection_date =
  "18.02.21", age=71, gender= "female", stage="I"))
```

- Add a column to the clinical data:

```
> treated <- rep( c("yes","no"), nrow(clinical_data)/2)
> clinical_mod <- cbind(clinical_data, treated)
```

Always check that your new dataset is what you expect, the same way you did after you imported the original one

Data Reshaping : Removing a Column

- Remove the new column of indexes, using exclusion (-) or column extraction

```
> clinical_orig <- clinical_mod[,-6] # remove the 6th
  column
> head(clinical_orig)  # check resulting data
```

or

```
> clinical_orig <- clinical_mod[,1:5] # extract all
columns that you want to keep (from the 1st to the 5th)

> head(clinical_orig) # check resulting data
```

Exporting Data to a File

The functions **write.table()** and **write.csv()** allow to write a data frame to a file.

Example:

```
> write.table(clinical_updated, file="clinical_updated.csv",
  quote=FALSE, sep=",",row.names=FALSE)
```

- Important optional arguments (check ?write.table for more):
  - **file** is the file path for the output file (if file name without a path is given, will be stored in current working directory).
  - **append** allows to append to an existing file (default is FALSE).
  - **quote** specifies whether the elements of character or factor columns should be surrounded by double quotes in the printed output (default is TRUE).
  - **sep** specifies the field separator to be used, e.g., comma (",") or tab ("\t").
  - **row.names** specifies whether or not the row names are written (default is TRUE). Alternatively, accepts a character vector with new row names to be written.
  - **col.names** specifies whether the column names are written (default is TRUE).

In a Nutshell

- How to import data into data frames (R's typical container for data)
- How to check the imported data, summarize it , access part of it, and manipulate it.
- How to export data to files

- Next step tomorrow: How to represent data graphically?

Let’s practice - 6

A clinical dataset from patients with lung cancer is available in the file ***clinical_data2.csv.*** Let's explore the dataset to see what it contains.

1) Open a new script file in R studio, comment it and save it.
2) Have a look at the csv file in R studio's file explorer. What do you need to check in order to be able to read in the file correctly?
3) Read the file into R, assign its content to object "clinical_data2". Examine the object.
4) How many observations and variables does the dataset have?
5) What is the structure of the dataset? What are the names and classes of the variables?
6) Which variables appear to be categorical? Convert them to factors.
7) Get the summary statistics of "clinical_data2"

Let's practice – 6bis

8) Use the function table() to compute the number of samples in different patient groups. a) How many samples are included of each gender (male, female)? b) How many samples are included per level of response to treatment (PD, SD, PR, CR)? c) Make a 2x2 table gender and level of response to treatment.

   Hint : try some of the example in the help(table) page.

9) Isolate the samples from male patients using subset(). Compute a summary statistics just for the weights of the subset. Then do the same for the samples from female patients. Export the data of each subgroup to a csv file.

10) Compute the means and standard deviations for male and female patient weights using tapply(). Then do the same by level of response to treatment.

R Style: Google's R Style Guide

Different authorities have different style recommendations for naming things, spacing, operator symbols, layout, commenting etc.

Example:
https://web.stanford.edu/class/cs109l/unrestricted/resources/google-style.html

Summary of selected styles from above guide (relevant to course content):

File names: Use meaningful names, ending with file extension .R `(predict_ad_revenue.R)`

Identifiers: Variable names should have all lower case letters, words separated with dots `(avg.clicks)`

Line length: maximum 80 characters

Indentation: two spaces, no tabs

Assignment: use <-, not =

Semicolon: don't use them

R Style: Google's R Style Guide (II)

Spacing:

Place spaces around all binary operators (=, +, -, <, etc.)

Do not place a space before a comma, but always place one after a comma.

Otherwise, do not place spaces around code in parentheses or square brackets

Good:

```
Total <- sum(x[, 1])      # spaces around <- and after comma
```

Bad:

```
Total<-sum(x[,1])          # no spaces
Total <- sum ( x[ , 1] ) # too many spaces
```

R Style: Google's R Style Guide (III)

Spacing - Exceptions:

Spaces around ='s are optional when passing parameters in a **function call**.

```
write.table(clinical_updated,
file="clinical_updated.csv", quote=FALSE,
sep=",",row.names=FALSE)
```

Extra spacing is okay if it improves alignment of equal signs (=) or arrows (<-).

```
write.table(x           = clinical_updated,
            file        = "clinical_updated.csv",
            quote       = FALSE,
            sep         = ",",
            row.names   = FALSE)
```

This is twice the same function call:
styled for brevity and styled for readability.
Both versions conform to Google R style.