1 Start R

- run GUI
- change working directory on GUI
  - quicklaunch icon>preferences>start in, type path there
  - R console>file/change dir
- alternative, 2-in-1 on command prompt: go into work dir and give command "R"
- help facilities
  - start them all from typing on R console:
  - help.start() – HTML help
  - help(keyword), ??keyword, help.search(keyword) – sometimes needs to be enclosed in " " (double quotes)
- have separate subfolders for sessions/projects!! as data and history will be saved in unnamed .Rdata and .Rhistory files – all work objects are saved!
- reusable code can be saved into .R script files, run them with GUI or source("filename.R")
- R console...
  - in editor, you can use arrow keys, up and down to browse previous commands, left and right to go back and edit (can't highlight and replace but can delete and insert characters)
  - .Last.value variable stores last output
  - calculation only prints output, assignment saves but doesn't print(!)
- packages
  - list installed packages with library()
  - access help with ?package.name
  - load package with library(package.name) (attach to search path)

- open previous workspace: file>load workspace and find the .Rdata file; or open the .Rdata file directly from Windows – makes available the objects and command history with arrows (will not print console log)
- console log must be saved *separately* with file>save to file

2 Data structures

2.1 Symbols, operators

- objects
  - in R's head, everything is an object – variables, user defined functions, etc.
  - many functions, analysis etc. give back complex objects, and we can get data out of them with more functions
  - good practice: do not overclutter the workspace, as all objects are kept in memory, as well as saved in save files!
  - object attributes
    - mode(): data type, character, number, logical, etc.
    - length(): length
    - attributes(): all other attributes
    - logical function to check mode: is.character()
    - coercing into given mode: as.character()
    - change given attribute attr(object,attribute name)=
    - class: goes beyond mode, allows for object oriented programming, specific printing functions for different object types, etc.
    - temporarily remove class with unclass()
    - ls() or objects() lists current objects
    - delete and free up memory with rm() or remove(), list object names to remove in argument
- special symbols
  - comment: # – can be used almost anywhere, effect till end of line

  - () is used for function arguments
  - [ ] or [[ ]] is used for accessing elements of vectors, matrices etc.
  - several commands in one line separated by ;
  - grouping with {}
  - accessing a named element/component with $
- variable and function names
  - can contain a-zA-Z0-9._ (dot and undesrscore)
  - function names have . where most languages would have _ !
  - R is case sensitive!
- assignment of value
  - x <- expr
  - expr -> x
  - x = expr (not common!)
- possible values
  - numbers – integer, double (=double precision floating point), complex (e.g. 2-3i)
  - characters and character vectors ("strings") – C type strings, 0 terminated, has \n, \t, etc.
  - logical types: TRUE, FALSE, NA (not available, missing data), NaN (not a number, e.g. 0/0 = ??)
- logical relational operators
  - <, >, <=, >= as usual
  - != not equal, == is it equal
  - %in% is it in the set
  - is.na() is it NA – TRUE for both NA and NaN – *always* use this for checking NA values!! == gives silly answers
  - is.nan() – only TRUE for NaN

- operators on logical expressions
  - & and, — or, ! negation, xor(), and use ()'s to group – all binary!
  - of longer vectors, use any() and all() instead of — and &

2.2 Vectors

- most basic and common
- a variable is a 1-element vector too
- all elements of same type – numbers may be coerced into characters
- for different types, need lists – see later
- create vector
  - with concatenation function c(), e.g. c(2,5,3,-2)
  - algebraic series with seq()
    - 4 arguments, but only 3 needed
    - from: from=, or unnamed at pos 1
    - to: to=, or unnamed at pos 2
    - stepsize: by=
    - length, total number of elements: len= or length=
  - : syntax – e.g. 2:5 = all numbers from 2 to 5, shorthand for seq(2,5,by=1) – has high precedence in order of operations
- how operators work on vectors
  - operators and functions are always applied *elementwise*!
  - vectors of different length are allowed, the shorter will be recycled as many times as needed
- maths operators
  - +, -, *, / as usual (division gives floating point numbers)
  - power: ^ or **
  - div, whole division %/%

  - mod, remainder %%
  - has some built-in functions, such as
    * elemtwise: sqrt(), sin(), cos(), tan(), log(), exp(), abs()
    * "scalars": min, max
    * range=c(min,max)
    * pmin, pmax: select min and max of each vector separately for several inputs
    * scalars: sum, prod, mean, var (adjusted empirical variance!), length

- sort vector with sort()

- access and replace elements
  - access with [] by index, or by name if it's defined
  - replace with vector[index] =
  - can be used to add new elements too (missing in between indices are filled with NA)
  - access several elements with : or c()
  - c(), or external index vector can be used to take elements out of order as well, possibly with repetition, possibly resulting in a longer vector than original etc.
  - take everything but given elements with - : or -c()
  - that's what we use for "removal" as well (no specific function to remove) and save as new variable/replace old
  - or manually set length() shorter to cut off the end of the vector
  - can also select elements by logical condition(s) with vector[(condition)]
  - can select elements with sticking a logical vector in [] as well – logical vector will be repeated as many times as needed

- combine and repeat vectors
  - concatenation with c()
  - rep(vector,times=2) – concatenate with itself 2 times
  - rep(vector,each=2) – repeat *elementwise* 2 times
  - paste(string1, string2, sep=) – more or less outer product of the character vectors; default separator is space

2.2.1 Factors

- a factor represents a *categoric* variable, that is, a finite selection of values are allowed and it's meant for grouping rather than calculations
- a factor can be ordered or unordered – e.g. grades would be ordered, but counties would not
- order is given by sorting, e.g. increasing numbers of alphabetical order of strings
- convert a variable into a factor with factor() – handled differently internally, printed differently, etc.
- levels() queries the created categories/possible values of the factor
- factors and tapply(data,factor,function) allow us to split data according to categories and make calculations
  - assuming data and factor are vectors of the same length and the same index pairs are data records
  - tapply splits the data into subvectors according to factor and then applies function to each subvector
  - usage example: data contains height of students, factor contains their sex (in the same order), and we want to calculate average height of boys and girls separetely
  - the combination of data + factor here can be thought of as a ragged array: categories may be of different size
- create ordered factor of a vector with the function ordered() – in case a natural ordering exists, e.g. numerical grades
- cut continuous data into categories
  - cut(data,breaks), where
  - either breaks=integer to determine how many categories should be created
  - or breaks=vector that contains the cut points
  - optional argument: ordered_result=TRUE, make the result an ordered factor

  - automatic labels with the resulting intervals

- frequency table: table(factor)
- can be used for two- or more-way frequencies by passing more factor arguments

2.2.2 Array, matrix

- similar to vectors, elements of same mode, but may have multiple indices $\sim$ multiple dimensions, matrix is specifically 2-dim, array can be higher dimensional
- coerce/convert a vector into e.g. an n$\times$m matrix by defining its dimensions: dim(v) = c(n,m) – v must have exactly nm elements!
- default: filled by columns(!) – matrix can have byrow=TRUE argument to fill by row
- create them: matrix(data,nrow,ncol), array(data,dim) – data is vector (will be repeated if necessary)
- elements can be accessed with multiple indeces: v[i,j] (i$\leq$n, j$\leq$m)
- omitted index means "take all", get entire rows and columns with v[i,] and v[,j]
- we can use c() or : to select multiple rows/columns – result is an array of all intersections of selected rows and columns
- single index (or c() or :) returns values of the *underlying vector*
- for a 2-dim array, a 2-column index array can be used for indexing: each row is taken as the i,j coordinates of an element, and thus a vector of elements can be queried or overwritten
- matrix operations
  - size: nrow(), ncol()
  - matrix multiplication %*% – works for matrix times vector as well

    - for vector %*% vector, it's ambiguous! usually taken to be inner product; one vector can be forced to column or row with cbind() or rbind(); but crossprod() and outer() is recommended
  - transpose t()
  - crossprod(x,y) = t(x) %*% y, but more efficient; crossprod(x) = crossprod(x,x)
  - diag(matrix) = vector of diagonal entries; diag(vector) = generate diagonal matrix with values of vector; diag(n) where n is number: n×n identity matrix (!)
  - solve(A,b): calculates solution x of Ax=b LEQ; solve(A) calculates inverse of A (!) – but it's counterrecommended, inefficient
  - eigenvalues: eigen(matrix), contains named variables "values" and "vectors" in a list – get named components with $, e.g. ev = eigen(A); evalues = ev$values
  - determinant: det()
  - singular value decomposition svd(A), gives a named list of "u", "d" and "v"

- array arithmetic
  - default for the basic operators: element by element operations(!!)
  - outer product: A %o% B, outer(A,B,"*"), more generally outer(A,B,function)
  - generalized transpose: for array, aperm(array,permutation) shuffles the dimensions according to permutation (which must be permutation of number of dimensions)

- concatenate matrices and arrays
  - cbind(arr1,arr2) concatenates them as consisting of column vectors, side by side – they must have same number of rows
  - rbind() analogously with row vectors
  - c() strips back data into vector form to concatenate!

3 List, data frame

3.1 List

- list: vector-like object, but elements need not be of same mode, and they may be named (think Python dictionary)
- elements can always be referred to by number
- elements may even be lists – recursive type
- create list with list(name1=element1,unnamed element,...) (similar to c() except for the names)
- refer to single elements (only the dictionary value!) by list[[index]] or list[["name"]] or list$name – in list[["name"]] syntax, "name" can be read from a character vector
- the [] syntax also works, but with different effect – e.g. brings the elements name (dictionary key) with it
- if an element is a list/vector/array, add another [index] to access its sub-elements
- names(list) is an overwritable attribute of the list containing the element names
- concatenate lists with c(), result is list again

3.2 Data frame

- data frame (mode data.frame) is a crossover between matrix and list
- well suited to record experiment data, measurement/data type per column and experiement per row
- its columns are vectors, aka must be the same type (may contain NAs), and they may be named
- rows may be named, by default indexed by positive integers
- data frames are built column by column, data sources may be:

  - vectors, factors
  - matrices
  - other data frames
  - lists – of named vectors/factors
  - can use several, but sizes must match (vector/factor lengths and nomber of rows in matrixes)
  - maximal information is inherited (e.g. column names from source data frame)
- create with data.frame(list of sources,name=vector,...)
- optional argument for naming rows: row.names=
  - default NULL, rows are automatically numbered
  - single column index or column name to take row names from
  - vector of numbers or character vectors to use
- coerce another structure into data frame with as.data.frame()
- little namespace trickery
  - having to type my.data.frame$column.name gets tedious
  - attach("my.data.frame") adds my.data.frame to the search path of looking for variables – aka it's enough to type column.name to *read* the variable (for writing, we still need $!)
  - detach("my.data.frame") removes it from the search path to "hide" the variables/columns again
  - can also attach lists to make named components visible
  - check search path with search()
  - detach() works with position number as well
  - ls(position number) lists variable names in the given namespace
- accessing elements and modifying data frames
  - access columns with my.data.frame$column.name or with index
  - access rows with my.data.frame[index,]

  - access single elements with double index – can use index or row/column name too
  - can always add new column with my.data.frame$new.column.name =
  - can redefine column with same syntax, e.g. turn vector into factor
  - add new rows:
    * my.data.frame[index,] or my.data.frame["row.name",] = list(...)
    * the comma is important, as a single index number will give back column instead of row!
    * if existing index or "row.name" is given, the row is overwritten (!)
    * if "row.name" is new, attached at the bottom
    * if index is well out of range, extra rows with NAvalues are created
  - graphically edit data frame like a spreadsheet with fix(my.data.frame) or edit(my.data.frame) – can modify values, add columns, add rows, etc. – difference: fix edits the original, while edit doesn't change the original, just returns a modified copy
  - remove rows or columns: use indexing seen before at vectors/lists to select parts of the data frame, which can be saved as new data frame (or overwrite the old)

- external sources of data
  - read data from (suitably formatted) external file into a data frame with read.table("source.data")
    * assuming a data frame format in the source file as well: first row is column headers, first element missing, other rows have row name as first element
    * otherwise with optional argument header=TRUE just assume no row names but first row contains column headers
  - many datasets come with R itself, as predefined data frame objects!
  - data() command lists all available

- ?name.of.dataset opens online help with explanation of sample size, variables and interpretation
- or use fix()/edit() to take a look
- even more datasets available from other R packages
  - load with data(dataset.name, package="package.name")
  - or load/attach package with libary(package.name), then its dataset(s) become available