What We Expect You to Know

Structured Programming I

Know the material from the last lecture! In particular

- What the different types of vectors are, and what differentiates them from lists
- Different ways of accessing and setting elements in vectors, lists, matrices, and data.frames
- Vectorization and recycling by operators
- Loops: for, while, repeat, next, break
- Conditionals: if, else, switch(), ifelse()
- Logical operators, differences between &,| and &&,|| and when each is needed

What We Expect You to Know

Structured Programming I

Learn about these useful functions included in R that you may not have known before. `help()` is always a good start for this!

- Sequences (similar to colon operator): `seq_len, seq_along, seq`
- Sets: `setdiff, union, unique, setequal, duplicated, anyDuplicated, table`
- Indexing & finding things: `which, which.max, which.min, max, min, pmax, pmin, match, row, col`
- Logic operations: `all, any, identical, all.equal, isTRUE, isFALSE, xor`
- Value transformations: `cut, hist, diff, floor, ceiling, round, trunc`
- Vector reordering: `head, tail, append, rep, rep_len, rev, sort, order, sample`
- Type and type conversion: `is.<TYPE>, as.<TYPE>, anyNA, is.na, is.finite, mode, type, typeof, ordered, factor, unlist, class`
- Basic string operations: `nchar, substr, sprintf, toupper, tolower, paste, paste0`

What We Expect You to Know

ProgR Code Style

- Code formatting
  - Limit line length to 120 characters
  - Use spaces, not tabs for indentation (this is the default in RStudio)
  - Use spaces before curly braces, and use a newline after opening braces except when using 'else':

```
if (x == 1) {
  x.is.one <- TRUE
} else {
  x.is.one <- FALSE
}
```

  - Opening parentheses, curly braces and square brackets add two spaces of indentation each, except if they are opened **and** closed on the same line:

```
vapply(1:10, function(x) {
    x^2
  },
  numeric(1)
)
```

    but

```
vapply(1:10, function(x) {
  x^2
}, numeric(1))
```

  - Use spaces around most infix operators and on the outside (but not inside) of parentheses; exception: no spaces before function call parentheses. Space after a comma, but not before. Two spaces before comments that share a line with code.

```
x <- (-1 + 1:2) * seq(from = 2, to = 5)  # comment
```

  - Do not have any trailing whitespaces at the end of lines
- Code conventions
  - Use `lowerCamelCase` for functions, `dotted.case` for variables, and `UpperCamelCase` for classes.
  - Use double quotes (`"`), not single quotes (`'`) for strings
  - Use `TRUE` and `FALSE`, not `F` or `T`
  - Use `<-` for assignment, not `=` or `->` or `<<-`
  - Do not use `L` to denote integers
- Avoiding common bugs
  - Use `seq_len()` and `seq_along()` instead of `1:n`, `1:length(x)`
  - Do not use `== NA`, use `is.na()`
  - Do not use `c()` for single values: No `c(1)`, `c(x)`, only for `c(1, 2)` or `c(x, y, z)`
  - Use double-square-brackets to access individual vector elements: `x[[1]]`, not `x[1]`
  - Use `vapply()`, not `sapply()`
  - All function arguments following the third should be passed by name
- ProgR homework specifics
  - Do not load any libraries, even indirectly (no `library()`, `require()`, `::`, ...)

What We Expect You to Know

Programming Style

Make your program easy to read by:

1. Avoiding unnecessary noise and keeping the appearance of your code uniform.
2. Using all communication channels available to you, like variable / function names, comments, and idioms that your reader is familiar with.
3. Assuming your audience has tunnel vision and should be able to understand small blocks of your code at a time

(And remember, the code you hand in is checked for style automatically)

What We Expect You to Know

Structured Programming II

- Matrices and data.frames are internally vectors and lists with special attributes
- Know different ways of accessing and setting elements in matrices, and data.frames
- Matrix creation and manipulation: `rbind, cbind, diag, expand.grid`
- Matrix / Dim info: `dim, colnames, rownames, dimnames, nrow, ncol, NROW, NCOL, length, names`

What We Expect You to Know

Functions

Know the following about functions

- How to define functions
- Control flow, `return` statement
- Functions are objects, and can be defined as anonymous functions
- Arguments, argument default values, missing arguments, and when arguments may be missing
- Function argument matching by position and by name
- How variable scoping works. What variables can a function access, what is variable shadowing, and what variables can a function modify?
- Copy semantics prevents functions from changing their arguments directly
- how the `...`-arguments work, how to get the number and values of `...`-arguments, how to pass them on to other function calls
- `do.call` to call functions with a list of arguments

What We Expect You to Know

`lapply()` and similar Functions

- sapply(): simplify to a vector / matrix, if possible
- vapply(): sapply, but with *type safety*: ensure the return-type of a function!
- mapply(): sapply, but going along more than one input list/vector
  - Map(): like mapply(), but not simplifying (like the difference sapply <--> lapply)
- apply(): sapply, but to to each row / column of a matrix
- tapply(): INDEX argument indicates *group*. apply group-wise:
- rapply(): recursive lapply (don't need to know this one)
- Reduce(): Apply a two-argument successively to elements and previous result
- Filter(): return only members for which a "predicate" is TRUE
- Find(): Like "Filter()" but return only first element
- Position(): Like "Find()", but return position instead of element

What We Expect You to Know

`checkmate` Functions

- Difference between testXxx, checkXxx, assertXxx functions, and why they are useful
- For a given constraint (type, minimum, length, null / NA ok) you should be able to find the corresponding checkmate assertXxx-function.
  - In particular: vectors, scalars, sets (i.e. atomic vectors with ordering disregarded), data.frame, matrix, function, NULL
- use assert(check(...), check(...), ..) to assert "or"-conditions
- `%???%`-operator to get first non-NULL value

If You Want to Go Beyond This

What we are not covering here: environments, frames and call stacks, closures, promises and lazy argument evaluation. If you want to learn more about that:

- check out Advanced R's chapters on functions and on environments.
- look at the R Language Definition sections 3.5, 4.3.3 and the R help pages of the functions mentioned there

What We Expect You to Know

- use stop(), warning(), message() for different signals
- ignore messages/warnings with suppressMessages / suppressWarnings
- change consequences of warnings using options(warn=)
- use tryCatch
  - to catch errors / stop()s
  - to catch custom errors created with errorCondition()
- know about try() (but preferably use tryCatch)
- use on.exit() to perform actions before leaving a function

What We Expect You to Know

- Solve big problems by breaking them down into subproblems, for which you then write functions.
- Functions should fulfill a clear purpose that can (and usually should) be explained in few sentences.
- Function and argument names should be descriptive
- Function arguments and return values should be documented, in particular their expected type

(This is not something we can readily test in our homework / exam and is therefore "not examinable", but keeping this in mind will make your life easier when you have your own projects)

What We Expect You to Know

- Pattern matching functions: **grep()**, **grepl()**, **sub()**, **gsub()**, **regexpr()**, **gregexpr()**, **regexec()**, **gregexec()**
- Function of their arguments **ignore.case**, **perl**, **fixed**
- Use **regmatches()** to extract matching strings
  - including (possibly named) groups with **regexec()**
- Regex special symbols: `.` `|` `+` `*` `?` `^` `$` `\\` `(...)` `[...]` `[^...]` `[a-z]` `[[:...:]]` `{n}` `{n,m}`
- Groups with `(...)`; named groups: `(?<name>:...)`; groups without reference: `(?:...)`
- Lookahead and lookbehind, both positive and negative: `(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)`

What We Expect You to Know

- Profiling
  - Know how to use `profvis` to find hotspots of your code
  - Be aware of lazy evaluation of function arguments
- Benchmarking
  - Know how to use `system.time()`
  - Know how to use `microbenchmark`
    - call it
    - interpret results
- Writing fast code
  - Avoid writing unnecessarily slow code
  - Don't sacrifice maintainability, code clarity, or your time for speed unless you know it gives relevant benefits
    - the code in question is actually a hot spot
    - the gain in speed is worth it

What We Expect You to Know

- Adopting a **reproducible workflow** for your projects can save you time and stress in the long run.
- Set up your **project folders** so that data, code, results etc. are kept separate.
- Document what you are doing and how to run things, both in the **README.md** file and in the script / report files.
- Use **git**, and set up your **.gitignore** so that data, intermediate results etc. are not added to the repository.
- Use RMarkdown (**.Rmd**) files to create reports.
- Make sure your result can be reached from raw data **without any manual editing** of files
- Make sure that all steps needed to get the results are actually in script files, and that **interactive commands in the R console are not necessary**.
- Make your code independent of your computer; in particular do not use absolute paths and setwd(), instead always use paths relative to the project's folder.
- Consider using **shell scripts** to do some data preparation or result post-processing.
- Consider using an automation and dependency management tool such as **Snakemake** or **Nextflow**, especially when you have expensive intermediate results.

What We Expect You to Know

- Random value generation: `runif()`, `rexp()`, `rbinom()`, `rgeom()`, ...
- Sample and shuffle integers using `sample.int()`
- Use `sample()` on vectors if you are *sure* they are not numerics
- Use `set.seed()` to initialize the PRNG and get reproducible results
- Use `RNGversion()` to make sure the same PRNG is used in different R versions
- Get the PRNG-state from `.Random.seed`; you can save it, and then restore it by assigning a saved value to `.Random.seed`.
  - With some random functions, you have to call `set.seed(dummyvalue)` before to reset them.
  - `.Random.seed` is *not* the same value as given to `set.seed()`.
- Changing your code (or even the version of a package) can have an influence on PRNG call order (and therefore your results) in surprising ways; consider it possible that all values change once something in the code in between has changed.
- Using cached values or conditionally skipping parts of your code will lead to differing PRNG states; cache the `.Random.seed`, or call `set.seed()` again after optional chunks, to avoid the problem
- Do not run different parts of your code with the same seed.

What We Expect You to Know

- Construct with (`as.`)`data.table()` or `setDT()`
- `dt[i, j]`
- Both `i` and `j` are evaluated inside the data.table
  - exceptions: `i` with single symbols, `j` with constants
- `i` selects rows `logical()` or `numeric()` values
- `j` creates a new data.table out of lists (use `.()` as shortcut) or returns atomics
  - use `with = FALSE` to select columns directly
- Use `:=` in `j` to create or change columns
  - use `` `:=`() `` to assign multiple columns at once
  - use `(...) := ...` to assign to columns dynamically; possibly multiple at once
- Give both `i` and `j` to have the `j`-expression evaluated on a subset of the original table
- This all "adds up to normality" most of the time
- But you can get very elaborate in your expressions!
- For more useful printing, set

```
options(datatable.print.class = TRUE,
        datatable.print.keys = TRUE,
        datatable.print.trunc.cols = TRUE)
```

What We Expect You to Know

`data.table`: know how to...

- tell if an object is a `data.table` or just a `data.frame`
- change how data.table objects are printed
  - by giving `datatable.print.xxx` arguments to `print()` as described in `?print.data.table`
  - by using `options(datatable.print.xxx = OPTION)` to set the option globally
- get a `DT` from `data.frame`s, matrices, `list`s of rows: `as.data.table`, `setDT()`, `rbindlist()`
- get individual rows, columns, elements from a `DT`
- subset a DT: specific columns, specific rows, rows by a condition
- modify or add new columns based on calculations done on old columns using `` `:=` `` or `set()`
  - single col at a time & `[, (<col group>) := .(<value list>)]`
- handle in-place functions (that start with set... in data.table) as well as the `` `:=` `` operator in `[ ]`), know about reference-semantics and use `copy()` if needed
- use `fread()`, `fwrite()` for fast reading / writing of large files
- do aggregation with `` `by =` ``
  - count subgroup sizes with `.N`
  - calculate aggregate values for subgroups
  - advanced aggregation control with `.SD` and `.SDcols`
  - other special values: `.BY`, `.GRP`, `.NGRP`, `.I`, `.EACHI`
- work with list columns that may contain different kinds of data on different rows
- merge / join data.tables
  - both with `merge()` as well as with `X[Y, ...]` (with `X`, `Y` both being a `DT`)
  - understand the difference between inner, left/right, outer, anti join and how to do them in `DT`
- reshape `DT`s between "wide" format and "long" format using `dcast()` and `melt()`
- use keys
  - what are keys useful for?
    - automatic sorting
    - fast row subsetting
    - row selection using X[<value>]
  - `key()`, `indices()`, `haskey()`
  - difference between `setkey()` and `setindex()`
  - difference between `setkey()`/`setindex()` and `setkeyv()`/`setindexv()`

What We Expect You to Know

`data.table`: know about... (grey: not that important at our level)

- using `[]` as a suffix to print `data.table` in-place operation results even when they are "invisible"
- functions that treat `DT`s like sets of rows to do set operations and sorting on them
  - `fintersect()`, `fsetdiff()`, `fsetequal()`, `funion()`: set-operations that treat data table rows as sets
  - `duplicated()`, `unique()`, `anyDuplicated()`: find duplicate rows / restrict to unique rows
    - `uniqueN()`: short for `nrow(unique(x))`
    - also note these have a "`by`" argument
  - `frank()`, `frankv()`: `rank()` on data.table
  - `split()`: split data.table into list of smaller tables (but it is usually better do do aggregate operations with '`by`' in `[ ]`.)
  - `na.omit()`: exclude rows with NAs
- Further `set...()` functions
  - `setattr()`, `setnames()` -- change attributes by reference
  - `setcolorder()` -- reorder columns
  - `setorder()`, `setorderv()` -- reorder rows, similar to `setkey()`/`setkeyv()`, but without setting a key
- helpful operators for the `i` (i.e. row-selector) argument
  - `between()`, `%between%` -- between to values
  - `inrange()`, `%inrange%` -- in any of multiple ranges
  - `like()`, `%like%`, `%flike%`, `%ilike%`: faster `grepl()`; `%flike%`: fixed (not regex), `%ilike%`: ignores case
- general helper functions
  - `first()`, `last()` -- like `head()`/`tail()`, but get just one item
  - `shift()` -- lead or lag a vector
  - `transpose()` -- transpose lists, `data.frames`, `data.tables`
  - `tstrsplit()` -- `transpose()` of `strsplit()`
  - `fcoalesce()`: vectorized: give first non-NA value
  - `nafill()`, `setnafill()` -- fill missing values
  - `CJ()` -- cross product DT
- System info functions and global settings
  - `address()` -- address of an object
  - `setDTthreads()`, `getDTthreads()` -- change cpu parallelization threads
  - `tables()` -- summarize metadata of all 'data.table' objects in memory
  - `getNumericRounding()`, `setNumericRounding()` -- rounding mode for equality checks
  - `timetaken()` -- time difference to result of call `proc.time()`

What We Don't Really Expect You To Know, But Include Here for Completeness Sake

- fast version of R function, optimized for character vectors
  - chgroup(): like order(), but only groups together duplicates instead of sorting
  - chmatch(): character version for match()
  - chorder(): character version of order()
  - %chin%: character version of %in%
- other fast / more robust versions of R functions
  - fifelse(): ifelse(), preserves attributes
  - frank(), frankv(): faster rank(), but also ranks lists, data.frames and data.tables
- Helpers for aggregation and joining
  - groupingsets(), rollup(), cube() -- aggregate by different columns
  - Id column generators
    - rowid(), rowidv(): unique rowid
    - rleid(), rleidv(): run-length encoding
    - SJ(), CJ(): Join helpers
- Experimental (usage of these functions might change)
  - foverlaps(): fast overlap join
  - truelength(), alloc.col(), setalloccol(): over-allocation of column memory
  - frollmean(), frollsum(), frollapply(): rolling window aggregates
  - fsort(): faster sort through multicore
  - (Experimental) date/time class -- mostly a wrapper for POSIXct and Date
    - IDate, ITime: classes
    - as.IDate(), as.ITime(), IDateTime(): conversion
    - year(), quarter(), month(), week(), isoweek(), yday(), mday(), wday(), hour(), minute(), second(): get specific aspect from object

What We Expect You to Know

- **Classes** specify the structure of **Objects** which have **fields** and **methods**
- OOP in R:
  - S3: copy-semantics, implicit structure
  - R6: reference-semantics, explicit structure
  - some others

R6

- Define class:
  `R6Class(<name>, public = list(..), active = list(..), private = list(..))`
  - public: accessible from outside
  - private: accessible through special variable "`private`"
  - active: "active binding" function, looks like a field from outside
- Instantiate: `<Object Generator>$new()`
- Inheritance:
  - `R6Class(..., inherit = <superclass>, ...)`
  - Methods and fields from superclass if not overwritten
- Special variables inside methods: `self`, `private`, `super`.
- Special methods: `initialize()`, `deep_clone()`.
- Deep copy: `<Object>$clone(deep = TRUE)` , calls `deep_clone()` for all fields & methods

What We Expect You to Know

S3

- **attributes**: Additional information hanging on to objects in R
  - `"names"`: names of lists / vectors, `"dim"`: dimension of matrix / array
  - `"class"`: S3 class
  - Access through `attr(<obj>, <name>)` or `attributes(<obj>)$<name>`
  - Set conveniently with `<-` or `structure(<obj>, <name> = <value>, ...)`
- Create S3-object by setting `"class"` attribute
- Define class: Constructor function
  `<ClsName> <- function(..) { .. structure(list(..), class = "<ClsName>" ) }`
- **Generic** function: `<fname> <- function(...) UseMethod("<fname>")`
- **S3 Method**: `<fname>.<ClsName> <- function(...) { .. }`
  - Should have compatible signature (i.e. arguments) with generic
- Special method `print.<ClsName> <- function(x, ...)`
  - called automatically when object is displayed.
  - should have "`x`" and "`...`" arguments and must return `invisible(x)`.
- Inheritance through multiple entries in `"class"` attribute vector. Subclass first, superclass next.
- `NextMethod()` : call to superclass method
- assert through `assertClass(<obj>, "<ClsName>")` and assume internal structure is valid