Advanced R Programming - Lecture 2

Krzysztof Bartoszek
(slides based on Leif Jonsson's and Måns Magnusson's)

Linköping University

krzysztof.bartoszek@liu.se

3 IX 2025 (U2)

Today

Questions since last time?

Program Control

Two main components

- Conditional statements
- Loops

See also extra video on program control on course page

Conditional statements

```
if(Boolean expression) {
# statements
} else if (Boolean expression) {
# statements
} else {
# statements
}
```

Brackets "{...}":

- not needed if single line follows `if`, `else if`, `else`
- but defensive programming

Loops

- for (entry controlled loop)
- while (entry controlled loop)
- repeat (exit controlled loop)

Brackets “{...}”:

- again not needed if single line follows `for`, `while`
- but defensive programming

See also extra video on program control on course page

For loop

Entry controlled loop: checks condition before entering loop

```
for (name in vector){
# statements
}
```

While loop

Entry controlled loop: checks condition before entering loop

```
while (Boolean expression){
# statements
}
```

Controlling loops

- break (loop)
- next (iteration)

Repeat loop

Exit controlled loop: checks termination condition at end of loop body

```
repeat {
# statements
}
```

- `repeat` needs `break` statement
- brackets "{...}" needed
- unless empty loop: `repeat break`

Functions revisited

```
my_function_name <- function(x, y){
  z <- x^2 + y^2
  return(z)
}
```

Function components

Function arguments
Function body
Function environment

These can be accessed in R by:
formals(f)
body(f)
environment(f)

`fix()` inbuilt editor

Lexical scoping

(or how does R find stuff?)

Current environment ⇒
Parent environment ⇒
...
Global environment ⇒
... along searchpath to...
Empty environment (fail)

Environment search path

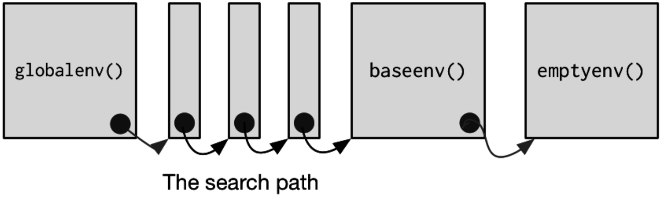

Figure: Environment search–path (H. Wickham, Adv. R, p.127)

```
parent.of.global<-parent.env(.GlobalEnv)
grandparent.of.global<-parent.env(parent.of.global)
```

Environment basics

"bag of names"

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```

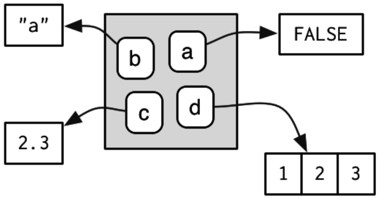

Figure: Environment (H. Wickham, Adv. R, p.125)

Environment relatives

Parents, but no children

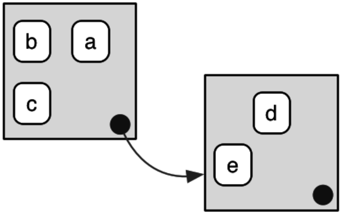

Figure: Env. relations (H. Wickham, Adv. R, p.126)

Working with environments

See environments as lists "of stuff in the bag"

```
ls()
ls(all.names=TRUE) #shows hidden

rm()
rm(list=ls()) #delete all
```

But memory is not released immediately, if required

```
gc(), gc(base)
```

Assignments

Shallow assignment
(inside current environment)
`<-`

Deep assignment
(inside parental environment, if not found, then assign in global)
`<<-`

Full control assignment
(manually specify environment)
`assign()`

allows for strange variable names, **not** recommended,
but what if automatically generated names

```
assign("x/x",3);ls();get("x/x")
```

Assignments

$=$ shallow assignment, equivalent to $<-$ in the most **external** scope
in function calls they a **different** meaning:

$$<- \text{ assigns values}$$
$$= \text{ defines parameter values}$$

```
> mean(x=1:100)
[1] 50.5
> mean(y=1:100)
Error in mean.default(y = 1:100) :
  argument "x" is missing, with no default
> mean(y<-1:100)
[1] 50.5
```

Lazy evaluation

Variables assigned values only when needed.

```
> f1<-function(x){globalVar<<-3;x}
> f2<-function(x){x<-x+0;globalVar<<-3;x}
## Are f1() and f2() the same?

> globalVar<-2
> f1(globalVar)
[1] 3


> globalVar<-2
> f2(globalVar)
[1] 2
```

Random number generators

Required for simulations—provide numbers mimicking (**pseudo–random** numbers) a given random variable

Theory will be discussed in Computational Statistics course

r`DistributionName`, e.g., `rnorm()`, `rexp()`

d`DistributionName`, **p**`DistributionName`, **q**`DistributionName`
density, cdf, quantile, e.g., `dexp(0)=1`, `pexp(0)=0`, `qnorm(0.5)=0`

discrete uniform (special dedicated function):
`sample(from_value,size,with_or_without_replacement,weights)`

Random seeds

When generating a pseudo–random number first the **random seed** is setup, in the variable `.Random.seed`.
It **defines** the sequence of random numbers that will be returned.
Set directly (`.Random.seed<-value`) or `set.seed(value)`.

```
> set.seed(1)
> rnorm(1)
[1] -0.6264538
> rnorm(1)
[1] 0.1836433
> set.seed(1)
> rnorm(1)
[1] -0.6264538
```

Setting up the generator

Random seed

Different generator types (algorithms).
Can specify generator for Gaussian, and discrete uniform pseudo–random numbers.

Can specify version of of generator (version of R).

```
RNGkind(kind=alg_kind, normal.kind=Gaussian_alg,
sample.kind=discrete_uniform_alg)
RNGversion(R_version)
set.seed(seed_value, kind=algorithm_kind,
normal.kind=Gaussian_alg, sample.kind=disc_unif_alg)
```

Random numbers: be careful

As of R 3.6.0 there is a change in the random number generator, seeds are not compatible between versions $<$ 3.6.0 and $\geq$ 3.6.0.
Use `RNGversion("R version string")` to set random number generator, see also `?RNGkind`.
Change due to “sample noticeably non-uniform on large populations” (from `?RNGversion`), see also
`https://bugs.r-project.org/bugzilla/show_bug.cgi?id=17494` .

```
suppressWarnings(RNGversion(min(as.character(getRversion()),"3.5.3")))
RNGversion(min(as.character(getRversion()),"3.6.0"))
set.seed(42, kind = "Mersenne-Twister", normal.kind = "Inversion")
```

RDieHarder: `R` Interface to the “DieHarder” RNG Test Suite
A huge number of tests described in NIST SP 800-22
`https://csrc.nist.gov/pubs/sp/800/22/r1/upd1/final` .

Assignments: be careful

The object .Random.seed is only looked for in the user's workspace.

(`https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html`)

`732A94_AdvancedRHT2025_Lecture02_Slide25.R`

Function arguments

copy-on-modify semantics

“modifying a function argument does not change the original value”

specify arguments by...

position

complete name

partial name

```
myfun(1,2)
myfun(firstarg=1,secondarg=2)
myfun(f=1,s=2)
```

Just in case: partial names cannot be used in function body

Function arguments (cont)

copy-on-modify semantics

`do.call()`
`missing()`
...
Default values

```
do.call(myfun,list(1,2))
do.call("myfun",list(f=1,s=2))
```

`missing()`: check if argument passed

Return values: the last expression evaluated in a function

Multiple values using lists

Pure functions “map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no side effects: they don’t affect the state of the world in any way apart from the value they return.” (H. Wickham, Adv. R, p.94)

Return values: the last expression evaluated in a function

Multiple values using lists

Pure functions "map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no side effects: they don't affect the state of the world in any way apart from the value they return." (H. Wickham, Adv. R, p.94)

on.exit()

"gets called when the function exits, regardless of whether or not an error was thrown. This means that its main use is for cleaning up after risky behaviour." `https://stackoverflow.com/questions/28300713/how-and-when-should-i-use-on-exit`

return()

Specials

“Most functions in R are prefix operators: the name of the function comes before the arguments.” (H. Wickham, Adv. R)

infix functions

“function name comes in between its arguments, like + or -” (H. Wickham, Adv. R)

Specials

"Most functions in R are prefix operators: the name of the function comes before the arguments." (H. Wickham, Adv. R)

infix functions

"function name comes in between its arguments, like + or -" (H. Wickham, Adv. R)

replacement functions

"Replacement functions act like they modify their arguments in place, and have the special name `xxx<-` . . . I say they "act" like they modify their arguments in place, because they actually create a modified copy." (H. Wickham, Adv. R)

Infix functions (p. 90)

- Useful to define arithmetic operations
- Example (inbuilt): `%*%`, `%%`, `%/%`
- Example (without %): `+`, `&&`, `<-`, `$`, `@`

Infix functions (p. 90)

```
> x<-1
> <-(x,2)
Error:  unexpected assignment in "<-"
> x+1
[1] 2
> +(x,1)
Error:  unexpected ',' in "+(x,"
> `+`(x,1)
[1] 2
> `<-`(x,3)
> x
[3]
```

Infix functions (p. 90)

- User defined infix functions must start and end with %
- Name of function has to be put in **backticks** when defining
- Example :

```
`%+%`<-function(a,b) paste0(a,b)
"new" %+% " string"
> [1] "new string"
```

- paste0 just concatenates without the separator

roperators package

Assignment operators %+=%
"Modifies the stored value of the left-hand-side object by the right-hand-side object. Equivalent of operators such as `+=` `-=` `*=` `/=` in languages like C++ or Python.
%+=% and %-=% can also work with strings."

For strings: %+%, %-% (string addition and subtraction), %s*%, %s/% (string multiplication and division)

```
> x<-1; x%+=%10 ##11
> "ab"%+%"c" ##abc
> "abc"%-%"b" ##ac
> "ac"%s*%2 ##acac
> "acac"%s/%"c" ##2
```

roperators's manual

Replacement functions (p. 91)

- When defining, replacement function's name has to be put in **backticks** (<- is in name **!**)
- Typically 2 arguments: object to modify, with what to modify
- Additional arguments go in the middle
- Does **not** modify in place, creates copy! **Performance issues**

Type manually, copy–paste, `source()` and inside called function.

`732A94_AdvancedRHT2025_Lecture02_Slide34.R`

Behaves differently on different versions of R. With newer versions of R modification in place seems impossible (textbook from 2015).

Functionals: “apply family of functions”

Higher order functions

Common in mathematics and functional languages

Functionals

Pros

(Often) faster alt. to loops
Easy to parallelize
Encourages you to think about independence (see above point)

Functionals

Cons

Can't handle serially dependent algorithms
Can make code more difficult to read

Common Functionals

`lapply()`
`vapply()`
`sapply()`
`apply()`
`tapply()`
`mapply()`

`rapply()`: nested lists
`eapply()`: elements of environment

USE `simplify` argument **!**

outer(): apply function to pairs

```
> outer(1:3,1:3) ## product by default
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
> outer(c("a","b","c"),c("a","b","c"))
Error in tcrossprod(x, y) :
  requires numeric/complex matrix/vector arguments
> outer(1:3,1:3,paste,sep="*")
     [,1]  [,2]  [,3]
[1,] "1*1" "1*2" "1*3"
[2,] "2*1" "2*2" "2*3"
[3,] "3*1" "3*2" "3*3"
```

Common Functionals: `library(parallel)`

```
parLapply()
parSapply()
parApply()
parRapply()
parCapply()
parLapplyLB()
parSapplyLB()
```

USE `simplify` argument **!**

Functional programming

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

— John Chambers"

Programming paradigm
Foundation in R
Key abstraction is "the function"
Especially *without side effects!*

R is *not* purely functional, few languages are

Anonymous functions

Functions without names
Often used in functionals

```
sapply(1:n,function(i){i^2},simplify=TRUE)
```

Closures: functions written by functions

"An object is data with functions. A closure is a function with data."
John D. Cook

Closure example 732A94_AdvancedRHT2025_Lecture02_Slide44.R

```
counter_factory <- function(){
  i <- 0
  f <- function(){
    i <<- i + 1
    i
  }
  f ## What is the returned object?
}
## ``function has own parent environment''
first_counter <- counter_factory()
second_counter <- counter_factory()
first_counter()
first_counter()
second_counter()

ls(environment(first_counter))
environment(first_counter)$i
```

Exception handling

Trap error instead of code crashing.

```
tryCatch({
    RcodeToRun
},error=function(e){R code to handle error e},
warning=function(w){R code to handle warning w},
message=function(m){R code to handle message m})
```

Errors should be handled locally if possible or passed on.
Language of description (check locale).
Can be used in parallel code, but then take care of description and returned value—output of threads might be merged.

throw in code: `stop(e)`, `warning(w)`, `message(m)`

R packages

An environment with functions and/or data
The way to share code and data

4 000 developers (date?)
nearly 12500 packages (as of 4 May 2018)
18428 packages (as of 15 August 2022) according to CRAN

```
available.packages(available_packages_filters = c("CRAN"))
```

Package basics

Usage

`library()`

`::`

`:::`

Installation

`install.packages()`

`devtools::install_github()`

`devtools::install_local()`

Package namespace

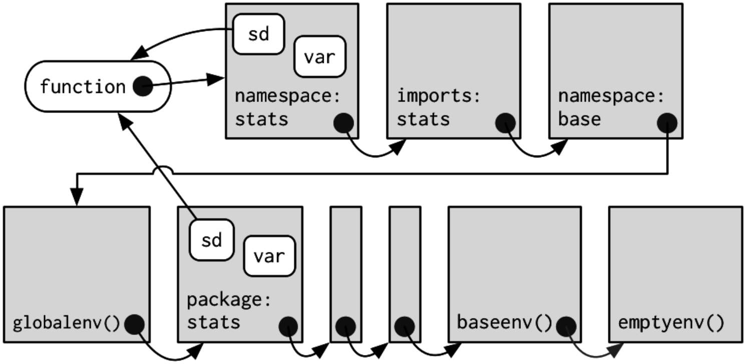

Figure: Package namespace (H. Wickham, Adv. R, p.136)

Which are good packages

Examine the package

1. Who?
2. When updated?
3. In development?

Semantic versioning

"Dependency hell"

`[MAJOR].[MINOR].[PATCH]`

(See reference on course page)

REMEMBER
ALWAYS
CHECK INPUT!

The End... for today.
Questions?
See you next time!