Advanced R. Chapter 7: Environments

Book by Hadley Wickham

Rladies Utrecht (Dewi, Barbara and Ale)

Presenter: Alejandra Hdz Segura

June 23, 2020

Welcome!

- This is a joint effort between RLadies Nijmegen, Rotterdam, 's-Hertogenbosch (Den Bosch), Amsterdam and Utrecht
- We meet every 2 weeks to go through a chapter
- Use the HackMD to present yourself, ask questions and see your breakout room
- We split in breakout rooms after the presentation, and we return to the main jitsi link after 20 min
- There are still possibilities to present a chapter :) Sign up at rladiesnl.github.io/book_club
- advanced-r-solutions.rbind.io has some anwers and we could PR the ones missing
- The R4DS book club repo has a Q&A section.
- This week's chapter: Environments!

Contents

Let's start!!

Environment basics

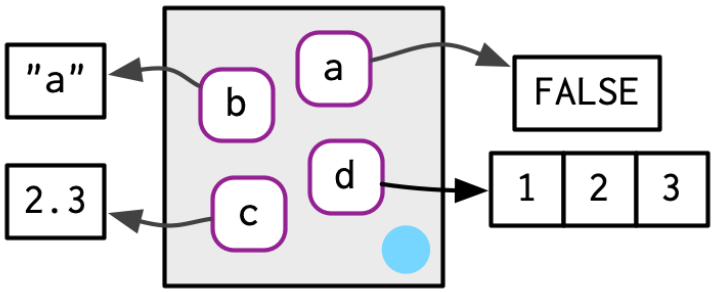
- You for sure have used environments
  - Functions
  - Packages
- Similar to lists... with some exceptions.
- In an environment:
  - Every name must be unique.
  - The names in an environment are not ordered.
  - An environment has a parent.
  - Environments are not copied when modified.

Let's explore environments

```
library(rlang)
```

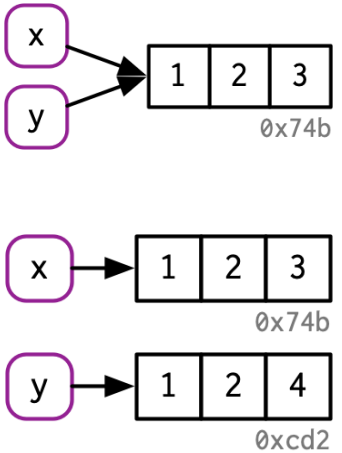
Creating an environment is like creating a list!

```
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3)
```

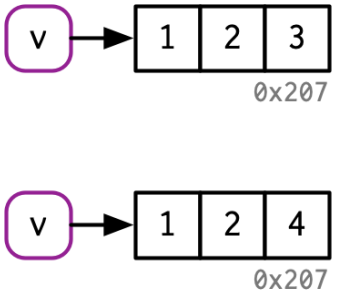


Copy on modify vs Modify in place

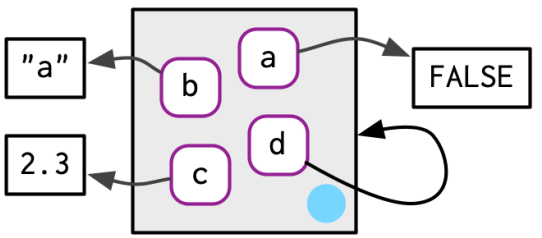
Copy on modify



Modify in place



Modified in place (not copy on modify) means also that environments can contain themselves!



Working with environments is special

- To print them

```
e1
```

```
<environment: 0x0000000013ea30b8>
```

```
env_print(e1)
```

```
<environment: 0000000013EA30B8>
parent: <environment: global>
bindings:
 * a: <lgl>
 * b: <chr>
 * c: <dbl>
 * d: <int>
```

- To see what they contain:

```
env_names(e1)
```

```
[1] "a" "b" "c" "d"
```

Important environments

- Current environment or `current_env()` is where your code is currently executing!
- Global environment or `global_env()` is where you can experiment interactively. also called 'workspace'.

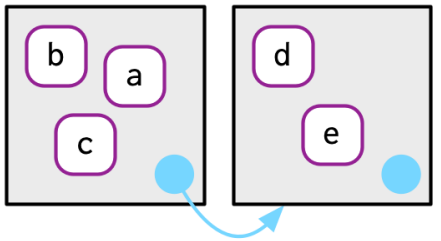
```
identical(global_env(), current_env())
```

```
[1] TRUE
```

- Note that we did not use normal `==` operator!

Parents

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```



Finding parents

```
lobstr::obj_addr(e2a)
```

```
[1] "0x1d9ed648"
```

```
lobstr::obj_addr(e2b)
```

```
[1] "0x1da51d90"
```

Parent of `e2b`:

```
env_parent(e2b)
```

```
<environment: 0x000000001d9ed648>
```

Parent of `e2a`:

```
env_parent(e2a)
```

```
<environment: R_GlobalEnv>
```

The orphan environment 😢

- All environments have parents except the empty environment or `empty_env()`

```
env_parent(empty_env())
```

```
Error: The empty environment has no parent
```

Actually, the `empty_env()` is sort of everyone else's great-grandma!!

HEY!! Don't forget about me!!

Almost full ancestry

```
env_parents(e2b)
```

```
[[1]]   <env: 000000001D9ED648>
[[2]] $ <env: global>
```

True full ancestry

```
env_parents(e2b, last = empty_env())
```

```
 [[1]]   <env: 000000001D9ED648>
 [[2]] $ <env: global>
 [[3]] $ <env: package:rlang>
 [[4]] $ <env: package:knitr>
 [[5]] $ <env: package:stats>
 [[6]] $ <env: package:graphics>
 [[7]] $ <env: package:grDevices>
 [[8]] $ <env: package:utils>
 [[9]] $ <env: package:datasets>
[[10]] $ <env: package:methods>
[[11]] $ <env: Autoloads>
[[12]] $ <env: package:base>
[[13]] $ <env: empty>
```

Did you notice? The ancestors of the global_env() include every attached package!!

Super assignment <<-

```
x <- 0 # Normal assignment
x
```

```
[1] 0
```

```
f <- function() {
  x <<- 1 # Super assignment! Modifies x outside the function!
}
f()
x
```

```
[1] 1
```

Subsetting environments

The good way

```
e3 <- env(x = 1, y = 2)
e3$x
```

```
[1] 1
```

```
e3[["y"]]
```

```
[1] 2
```

Subsetting environments

The bad way

```
e3[[1]]
```

```
Error in e3[[1]]: wrong arguments for subsetting an environment
```

```
e3$z
```

```
NULL
```

```
env_get(e3, "xyz") #If you want an error instead of NULL
```

```
Error in env_get(e3, "xyz"): object 'xyz' not found
```

Add/Remove bindings

Add

```
e3$z <- 3

env_poke(e3, "a", 100)

env_bind(e3, b = 10, c = 20)
```

```
env_names(e3)
```

```
[1] "x" "y" "z" "a" "b" "c"
```

Add/Remove bindings

Remove does not work like lists!!

```
e3$a <- NULL
env_has(e3, "a")
```

```
   a
TRUE
```

```
e3$a
```

```
NULL
```

You need to unbind instead!

```
env_unbind(e3, "a")
env_has(e3, "a")
```

```
    a
FALSE
```

Advanced bindings...

Delayed bindings are evaluated the first time they are accessed!

```
env_bind_lazy(current_env(), b = {Sys.sleep(1); 1})
system.time(print(b))
```

```
[1] 1
```

```
   user  system elapsed
      0       0       1
```

```
system.time(print(b))
```

```
[1] 1
```

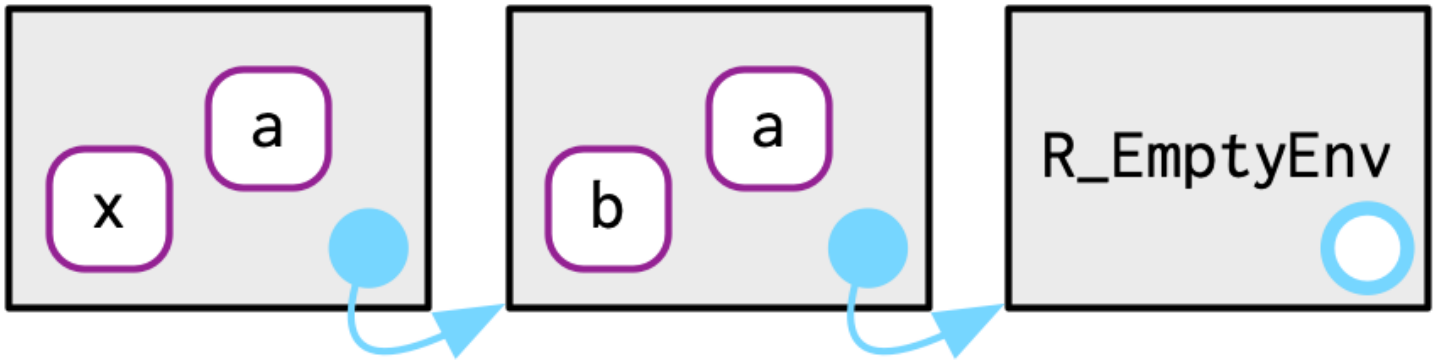
```
   user  system elapsed
      0       0       0
```

Active bindings are re-computed every time they are accessed:

```
env_bind_active(current_env(), z1 = function(val) runif(1))
```

Recursing over environments

How to find a variable?



e4b --> e4a --> empty_env

Three possible scenarios:

1. `where("a", e4b)` will find a in e4b.
2. `where("b", e4b)` doesn't find b in e4b, so it looks in its parent, e4a, and finds it there.
3. `where("c", e4b)` looks in e4b, then e4a, then hits the empty environment and throws an error.

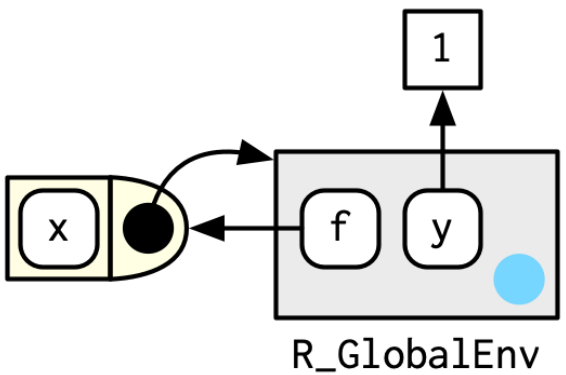
Special environments

Function environment

- A function binds the current environment when it is created
- Functions in R capture or enclose their environments
    - function = *closure*

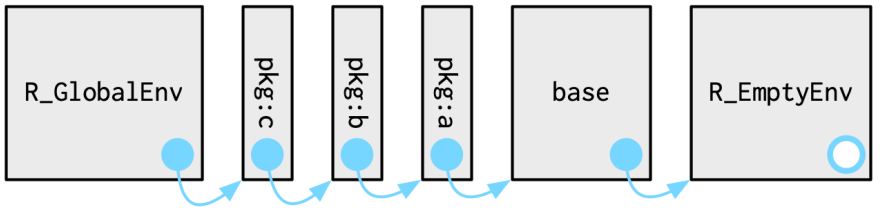
```
y <- 1
f <- function(x) x+y
fn_env(f)
```

```
<environment: R_GlobalEnv>
```



Package environments

Packages also have parents and are parents themselves!



Ancestry follows the order in which they have been attached! --> *search path*

```
search()
```

```
 [1] ".GlobalEnv"        "package:rlang"     "package:knitr"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

Wait a minute...

We know that search path depends on how you loaded packages...

Does that mean that the package will find different functions if packages are loaded in a different order?

Namespace

Make sure that every package works the same way regardless of what packages are attached by the user.

An example:

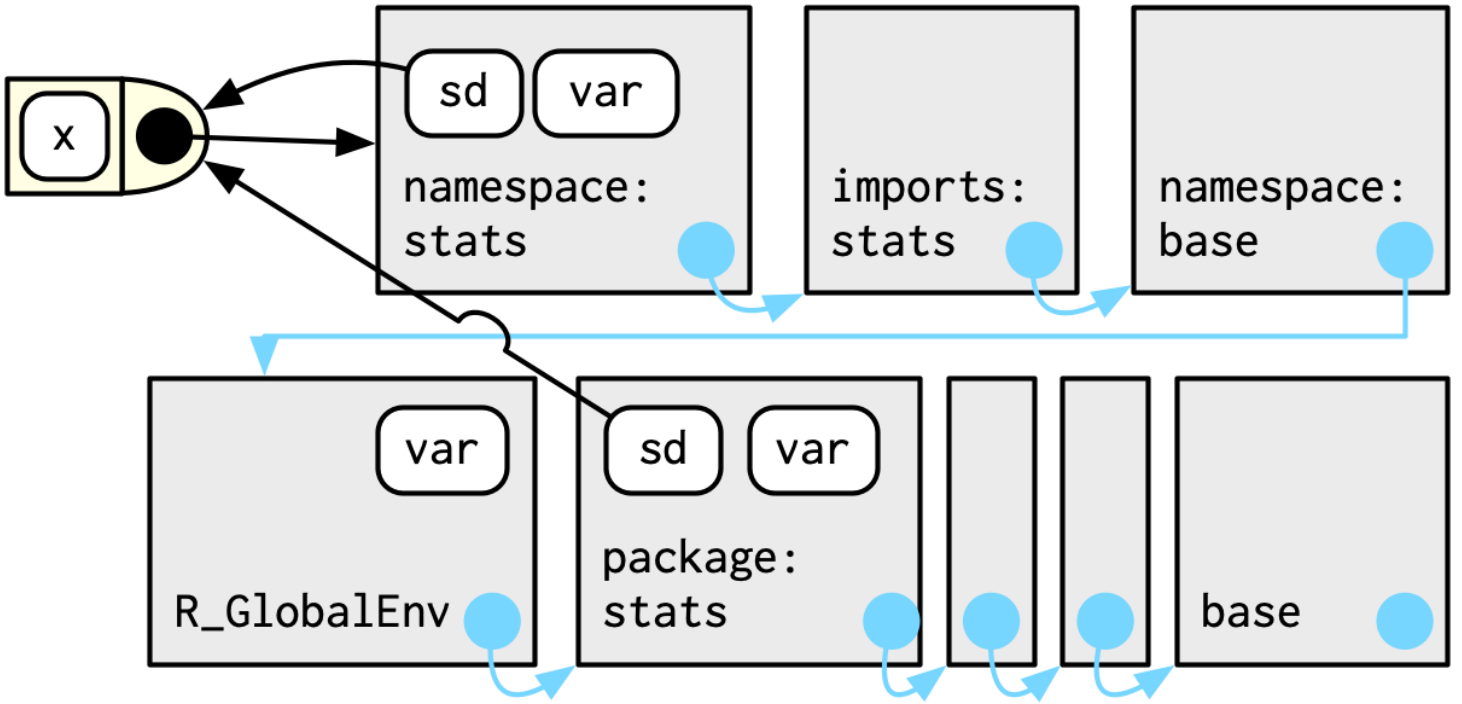
```
sd
```

```
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
    na.rm = na.rm))
<bytecode: 0x000000001e1d99c0>
<environment: namespace:stats>
```

`sd` (like any package function) is associated with:

- The *package environment* that is determined by search path
    - `stats::sd`
- The *namespace environment* that controls how the function finds its variables

Understanding how package/namespaces work:



Execution environments

```
g <- function(x) {
  if (!env_has(current_env(), "a")) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
```

What will the function return the first time it's run?

```
g(10)
```

```
[1] 1
```

What will happen if I call the function a second time?

```
g(10)
```

```
[1] 1
```

A new execution environment is created every time you call the function!

Making execution environments less *ephemeral*

- Returning the environment

```
h2 <- function(x) {
  a <- x * 2
  current_env()
}


e <- h2(x = 10)
env_print(e)
```

```
<environment: 000000001D70FAF0>
parent: <environment: global>
bindings:
 * a: <dbl>
 * x: <dbl>
```

- Return an object with a binding to that environment (**function factories**)

Call stacks

Functions have two contexts:

- **Execution environment:** depends on where the function was created
- **Call stack:** depends on where the function was called
    - Useful default whenever you write a function that takes an environment as an argument.
    - `rlang::caller_env()`

Simple call stacks

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  lobstr::cst()
}


f(x = 1)
```

```
   x
1. \-global::f(x = 1)
2.   \-global::g(x = 2)
3.     \-global::h(x = 3)
4.       \-lobstr::cst()
```

Call stack with lazy evaluation

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x


a(f())
```

```
   x
1. +-global::a(f())
2. | \-global::b(x)
3. |   \-global::c(x)
4. \-global::f()
5.   \-global::g(x = 2)
6.     \-global::h(x = 3)
7.       \-lobstr::cst()
```

Dynamic scoping

Looking up variables in the calling stack rather than in the enclosing environment

"Unique" of R.

Environments as Data Structures

Using them as data structures, they can help to solve some problems:

1. Avoiding copies of large data
    - For other/better ways check R6 objects (Ch. 14)
2. Managing state within a package
    - Objects in a package are locked, so you can't modify them directly unless you create a function that can access them (through environments)
3. As a hashmap
    - Data structure that takes constant time to find an object based on its name
    - Environments provide this behaviour by default (out of the scope)

ARE YOU AN EXPERT IN ENVIRONMENTS NOW?!!

Time to practice!!

Question 1

What is the difference between `lobstr::cst()` and `traceback()`?

z) `traceback()` does not work for environments

j) The order of `traceback()` and `lobstr::cst()` is reversed

m) `lobstr::cst` only gives you information about the last environment

Question 2

Let's say you first load package `tseries` and then `chron`, both containing a function named `is.weekend`.

From which package does R use the function and why?

Also: what does this have to do with environments in R?

i) From the package `tseries`

o) From the package `chron`

Question 3

Which of the following characteristics is not a difference between an environment and a named list?

h) Environments are copied when modified, lists are not

m) Every name in an environment must be unique

t) The names in an environment are not ordered

Question 4

Which of the following code will show you the full ancestry of environment e2b?

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```

n) `env_parents(e2b, last = empty_env())`

o) `env_parents(e2b, last = global_env())`

v) `env_parent(e2b)`

z) `env_full_ancestry(e2b)`

Question 5

What is the difference between assignment (<-) and super assignment (<<-)? And when is this useful?

w) Assignment modifies a variable, while super assignment always creates a new variable.

h) To create a variable in a new environment you always needs to use super assignment; regular assignment does not work in this case.

s) Assignment creates a variable in the current environment and super assignment modifies an existing variable found in a parent environment.

Question 6

Which of the following gives an error?

```
e8 <- env(x=1, y=2, u=4)
```

m) `e8$w`

k) `e8[[1]]`

v) `e8[["u"]]`

h) `e8[c("x", "y")]`

u) none of the above

w) all of the above

e) m, k and h

o) k and h

Question 7

Which of the following code removes an element from an environment?

s) `rm(e2a$x)`

n) `env_unbind(e2a, x)`

k) `e2a$x <- NULL`

x) both n and k

c) both s and n

Extra questions

- How does R look for objects? Why is this important?
- How do you determine the environment from which a function was called?
- If you have an environment `e2` that contains another environment `e1`. What would happen if you change or add a variable in `e1`. Would `e2` be affected? What use do you see in this behavior?

Answer

JOHNSON

Full name: Katherine Johnson

Who is she? American mathematician and one of the first African-American women to work as a NASA scientist.