Names, Values, and Environments

Programming Structures

Gaston Sanchez

STAT 33B, Fall 2025

About

In this lecture we examine some underlying principles that have to do with the way R works, namely:

- Names and Values
- Copy-on-modify policy
- R's motto
- Environments

This is a rather technical lecture but its content has important practical implications for writing code in R.

```
# suggested packages
library(lobstr)
library(rlang)
```

Names and Values

Names and Values

Consider the following code:

```
a <- c(1, 2, 3)
```

We have one variable or **name** `a` that is **bound** (referenced) to the **value** of the numeric vector `1, 2, 3`

```
# some info about a
typeof(a)               # data type
lobstr::obj_size(a)     # size
lobstr::obj_addr(a)     # memory location
lobstr::ref(a)          # tree of references
```

```r
# size; similar to utils::object.size()
lobstr::obj_size(a)
```

```
## 80 B
```

```r
# memory location; similar to utils::tracemem()
lobstr::obj_addr(a)
```

```
## [1] "0x14cf48828"
```

```r
# tree of references
lobstr::ref(a)
```

```
## [1:0x14cf48828] <dbl>
```

Names and Values

Now consider the case in which we have a and b both pointing to the same underlying data

```
a <- c(1, 2, 3)

b <- a
```

It seems that R has created a copy b of a.

Names and Values

- In the first line, `a <- c(1, 2, 3)`, the variable or **symbol** `a` is *bound* (referenced) to the **value** of the numeric vector `1, 2, 3`
- In the second line, `b <- a`, we get the impression that R has created a new object `b` that seems to be a copy of `a`
- But in reality both `a` and `b` are bound to the same vector value . So no copy has been created
- So far R has two names , `a` and `b`, associated to the same vector value `1, 2, 3`.
- In technical terms we say that there are 2 references to the numeric vector.

Memory location

R has two references, `a` and `b`, to the numeric vector `c(1, 2, 3)`

```
a <- c(1, 2, 3)
b <- a

lobstr::obj_addr(a)
```

```
## [1] "0x11ee66b58"
```

```
lobstr::obj_addr(b)
```

```
## [1] "0x11ee66b58"
```

Names and Values

Now let's consider this small change to `a`:

```
a[1] <- 0
```

- By modifying the first element in `a`, R notices that the numeric vector—to which both `a` and `b` are bound to—needs to be changed to `0, 2, 3`;
- R also knows that there is more than one name associated to the same vector.
- Because of this, R then creates a copy of `1, 2, 3` that it gets modified to `0, 2, 3`, which then gets associated to `a`
- We now have two symbols, `a` and `b`, bound to two different values (i.e. two different vectors).

Memory location

Now a and b are bound to two different values (i.e. two different vectors)

```
a[1] <- 0

lobstr::obj_addr(a)
```

```
## [1] "0x14cf0e188"
```

```
lobstr::obj_addr(b)
```

```
## [1] "0x11ee66b58"
```

Copy-on-modify policy

The previous example illustrates a concept known under slightly different ways:

- "R uses copy-on-write semantics"
- "R uses copy-on-modify semantics"
- "R uses call-by-value semantics"

Copy-on-modify (if necessary) policy

Ross Ihaka (one of the creators of R) calls this policy **copy on modify (if necessary)**

> *"Copying is done only when objects are modified. The (if necessary) part means that if we can prove that the modification cannot change any non-local variables then we just go ahead and modify without copying."*

https://stat.ethz.ch/pipermail/r-help/2000-February/009952.html

Does this mean that every time you modify an object, R creates a copy?

The answer is: "not necessarily", or conversely, "yes, only if needed".

So when does R know if a copy is necessary?

Names, Values, and Copy-on-modify

- R uses a **counting heuristic** to have an idea of the number of names bound to a value
- R knows when a name has:
  - no binds (0)
  - one bind (1)
  - or multiple binds (2)
- However, if there are multiple binds, R does not know exactly how many; it only knows there is more than 1 bind.

Names with no binds

What about a name that is unbound (i.e. a name with 0 binds)?

```
rm(b)
```

- When R detects a name that is not associated to any value, then it will get rid of it.
- The `rm()` function removes the binding between `y` and `1, 2, 3`, but it does not automatically frees up the memory space occupied by the numeric vector.
- R uses a **Garbage Collection** (GC) mechanism to take care of this (i.e. to handle memory management).

R's Motto

R's motto (inherited from S)

*Everything that **exists** in R is an **object**.*

*Everything that **happens** in R is the result of a **function call**.*

John Chambers.

Everything is an object

R follows the same motto of the S language:

> *Everything is an object.*

"The arguments, the value, and in fact the function and the call itself: all of these are defined as objects" (John Chambers).

Think of objects as collections of data of all kinds.

In S, as in R, there is one and only one way to refer to objects: by the combination of a name and an environment (or context) in which the name is evaluated.

Everything that happens is a function

This code

```
w <- c(2, 4, 6)
```

is actually a call to the assignment function `<-()`

```
"<-"(w, c(2, 4, 6))
w
```

```
## [1] 2 4 6
```

Everything that happens is a function

This code

```
w[2]
```

is actually a call to the extraction function `[()`

```
"["(w, 2)
```

```
## [1] 4
```

Everything that happens is a function

This code

```
w[1] <- 0
```

is actually a call to the replacement function `[<-()`

```
w = "[<-"(w, 1, 0)
w
```

```
## [1] 0 4 6
```

Everything that happens is a function

This code

```
2 + 3
```

is actually a call to the addition function `+()`

```
"+"(2, 3)
```

```
## [1] 5
```

Environments

About Environments

- The environment is the data structure that powers scoping.
- Recall that scoping is the act of finding the value of a variable.
- Understanding environments is not necessary for day-to-day use of R.
- But they are important for understanding features like lexical scoping, namespaces, and evaluation aspects.
- We'll be using functions from the `"rlang"` package for this part.

```
library(rlang)
```

Creating Environments

You can use the "rlang" function `env()` to create an environment

```
e1 <- env(
  a = c(2, 4, 6),
  b = TRUE,
  c = "hi",
  d = 3.3)
```

Creating Environments

You can use the "rlang" function `env()` to create an environment

```
e1 <- env(
  a = c(2, 4, 6),
  b = TRUE,
  c = "hi",
  d = 3.3)
```

An environment is similar to a named `list` except that:

- Every name must be unique.
- The names in an environment are **not ordered**.
- An environment has a **parent**.
- Environments are **not copied when modified**.

About Environments

Printing an environment just displays its memory address:

```
e1
```

```
## <environment: 0x11d2bafd0>
```

Instead, use `env_print()` which gives you more information:

```
env_print(e1)
```

```
## <environment: 0x11d2bafd0>
## Parent: <environment: global>
## Bindings:
## * a: <dbl>
## * b: <lgl>
## * c: <chr>
## * d: <dbl>
```

About Environments

- The job of an environment is to associate, or bind, a set of names to a set of values.
- Think of an environment as a bag of names or collection of named obejcts
- The names have no implied order (i.e. it doesn't make sense to ask which is the first element in an environment)
- Unlike most R objects, when you modify them, you modify them in place, and don't create a copy

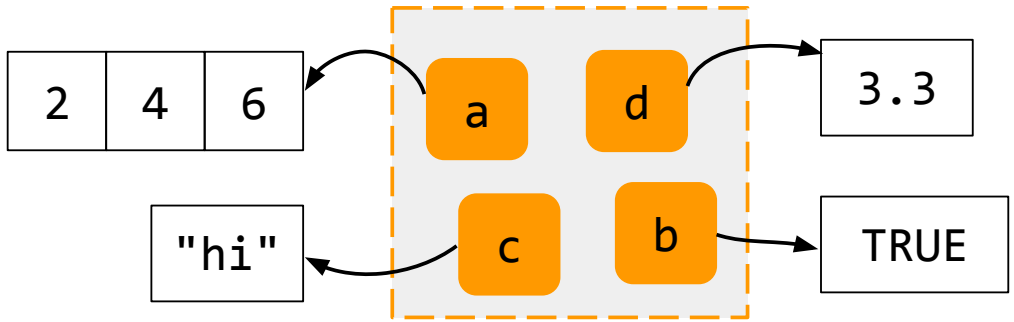
Environment Example

Let's bring back environment `e1`

```
e1 <- env(
  a = c(2, 4, 6),
  b = TRUE,
  c = "hi",
  d = 3.3)
```

Drawing Environments

Conceptually, environment e1 could be drawn like this

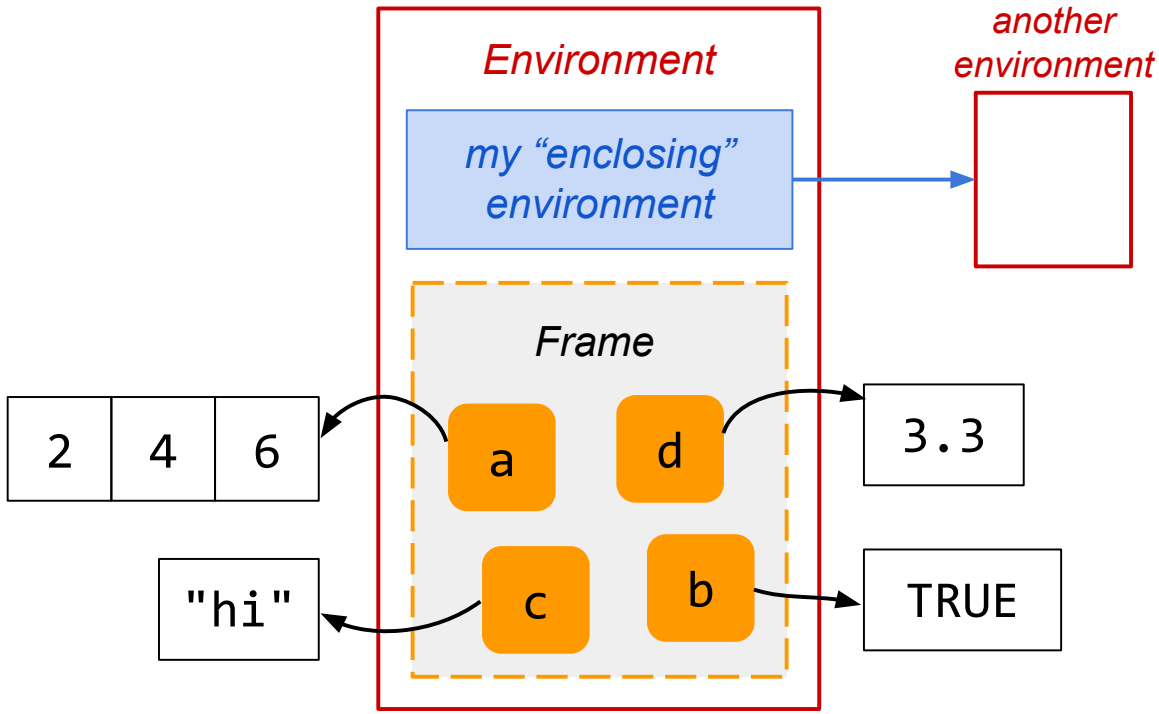


- the dashed gray square is the unordered "bag of words"
- the names are bound to their values (which are not in the environment)

However, a diagram like this does not capture all the features of an environment.

Drawing Environments

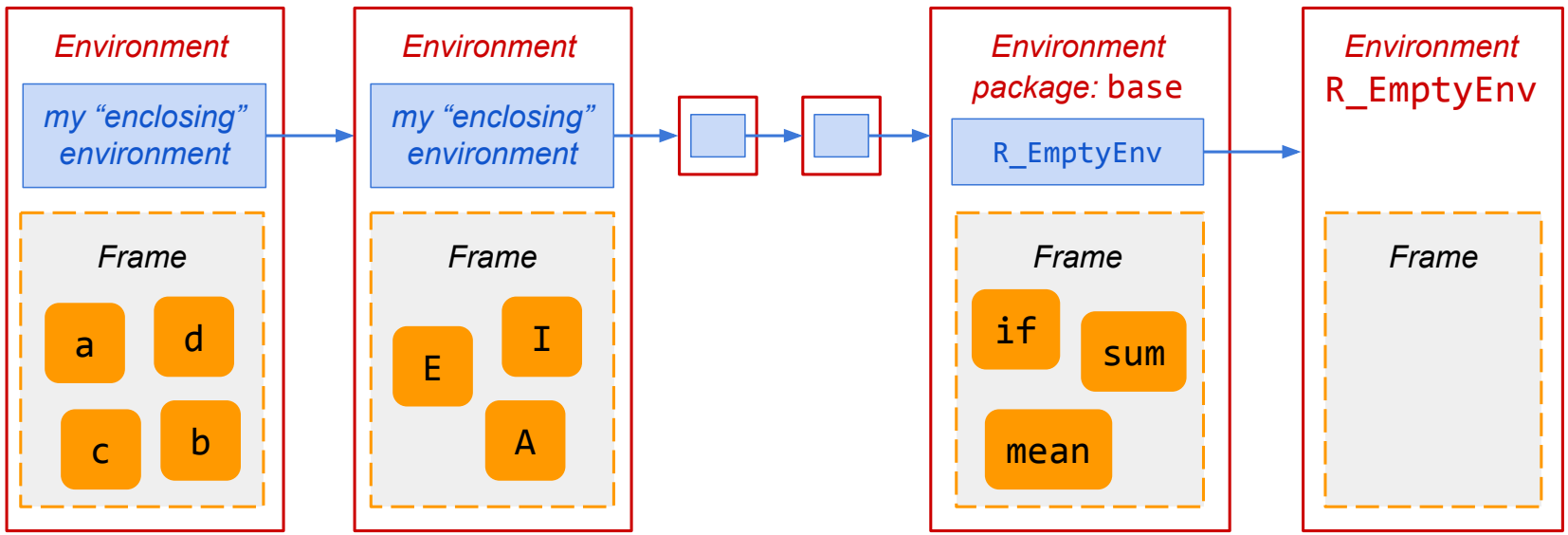
A more complete mental diagram could look like this



- an environment has a **parent** or enclosing environment
- the "bag-of-words" is technically called a **frame**

Drawing Environments

An even more complete mental diagram of R environments



- The chain of enclosing environments stops at a special environment called the **Empty Environment**, denoted `R_EmptyEnv`, and accessed by `emptyenv()`.

About Environments

- Every environment has a **parent**, another environment.
- The parent is what's used to implement lexical scoping: if a name is not found in an environment, then R will look in its parent (and so on)
- Only one environment doesn't have a parent: the **empty environment** (`R_EmptyEnv`).
- The ancestors of every environment eventually terminate with the empty environment
- One of the special environments is the Global Environment
- The ancestors of the global environment include every attached package

Package environments and the search path

- Each package attached by `library()` or `require()` becomes one of the parents of the global environment.
- The immediate parent of the global environment is the last package you attached
- The order in which every package has been attached is known as the **search path**
- You can see the names of these environments with `base::search()`, or the environments themselves with `rlang::search_envs()`

Some Remarks

I'm using the concepts of *ownership* and *containment* loosely

It would be more appropriate to say *pointer* instead of *own* or *contain*

I will say that environments *own* objects, particularly functions. In truth, functions are instructions stored somewhere in memory, and they are accessed by symbols residing in environments.

Environments and Functions

Environments and Functions

- Each time a function is called, a new environment is created to host execution. We refer to this as the **execution environment**.
- The parent of the execution environment is the function environment.
- An execution environment is usually ephemeral; once the function has completed, the environment will be garbage collected.

Importance of Environments

- When R executes an expression, there is always one *local* or *current* (or *active*) environment
- At any moment R can ask "hey, what's the local environment?"
- R asks this question a lot. In fact, it asks this question every time it needs to find a named object.
- R creates a new environment every time it runs a function (and remember that every thing that *happens* in R is the result of a function call).
- When you run code, functions call other functions, and environments spawn and die.

https://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

Evaluation of Functions

A call to an R function is evaluated in three steps:

1. The argument expressions in the call (the *actual arguments*) are matched to the formal arguments in the function definition.
2. A new environment is created and an assignment is made there for each of the formal arguments, containing the actual argument if any, and also any default expression if the argument was missing and there was a default. The enclosing environment of the new environment is the environment of the function object.
3. The body of the function is evaluated in the new environment, and the result is returned as the value of the function call.

Evaluation of Functions

Leaving out a few details, the essentials of function evaluation (in pseudo-code) are:

```
New.frame(amatch(definition, expr), expr)
value <- Eval(body)
Pop.frame()
value
```

Explanation in next slide.

Evaluation of Functions

- `expr` is the actual call
- `definition` is the function object
- `body` is the function body of that object
- `amatch()` carries out the argument matching
- `New.frame()` puts the matched arguments into a new frame, and makes that the current frame, in which `Eval()` will evaluate the body of the function
- after that, `Pop.frame()` gets rid of the frame

Evaluation of Functions

In R there is a crucial step (as opposed to S): the addition of the function environment in step (2), which affects the way names are looked up.

The function's environment is the environment in which it's created, for regular functions.

R searches names in the function environment.