Advanced R by Hadley Wickham

Chapter 2: Names and Values

Asmae Toumi

@asmae_toumi

2020-04-11

What's in Chapter 2:

- Section 2.2: distinction between names and values
- Section 2.3: describes when R makes a copy
- Section 2.4: explores how much memory an object occupies
- Section 2.5: describes the two important exceptions to copy-on-modify
- Section 2.6: concludes the chapter with a discussion of the garbage collector

Prerequisites

To understand how R represents objects, we'll need to install the **lobstr** package:

```
library(lobstr)
```

Binding basics

How would you read the following?

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

- Create an object named ‘x’, containing the values 1, 2, and 3: 💩
- It’s creating an object, a vector of values (1, 2, 3) **and it’s binding that object to a name, x** : 😁

Copy-on-modify

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

```
x
```

```
## [1] 1 2 3
```

How can we see what's happening under the hood?

You can call `obj_address()` to see this object's identifier:

```
obj_addr(x)
```

```
## [1] "0x1c1a4a496b8"
```

```
y <- x
```

```
obj_addr(y)
```

```
## [1] "0x1c1a4a496b8"
```

What happens to `x` when you modify `y` ?

```
y[[3]] <- 4
```

```
x
```

```
## [1] 1 2 3
```

- Changing `y` did not modify `x`.
- This is due to a behavior called **copy-on-modify**.

```
obj_addr(x)
```

```
## [1] "0x1c1a4a496b8"
```

```
obj_addr(y)
```

```
## [1] "0x1c1a48e8848"
```

What about functions?

The same copy-on-modify behavior applies for functions.

We can use `tracemem()` to track when an object gets copied. It allows us to do that because every time an object gets copied, a message containing the address of the object will be printed.

```
f <- function(a) {
  a
}
```

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
```

```
## <000001C1A471EC88>
```

```
z <- f(x)
```

We got no message here, which means no new copy was generated.

If f did modify x, then a new copy would get generated and thus a message would get printed by `tracemem()`.

Lists

Like vectors, lists also use copy-on-modify behaviour.

```
list_1 <- list(1, 2, 3)
```

```
list_2 <- list_1
```

```
obj_addr(list_1)
```

```
## [1] "0x1c1a418fb18"
```

```
obj_addr(list_2)
```

```
## [1] "0x1c1a418fb18"
```

```
list_2[[3]] <- 4
```

```
obj_addr(list_2)
```

```
## [1] "0x1c1a39bf648"
```

Lists (continued)

We can use `lobstr::ref()` to print the memory address of each object along with a local ID so that we can easily cross-reference shared components.

```
ref(list_1, list_2)
```

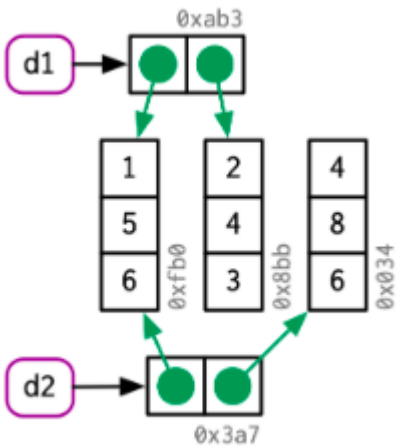
```
## o [1:0x1c1a418fb18] <list>
## +-[2:0x1c19e82f910] <dbl>
## +-[3:0x1c19e82f980] <dbl>
## \-[4:0x1c19e82f9b8] <dbl>
##
## o [5:0x1c1a39bf648] <list>
## +-[2:0x1c19e82f910]
## +-[3:0x1c19e82f980]
## \-[6:0x1c1a3a3e3b0] <dbl>
```

This shows that `list_1` and `list_2` have shared components, namely integers 2 and 3 corresponding to the 2nd and 3rd element in their vectors.

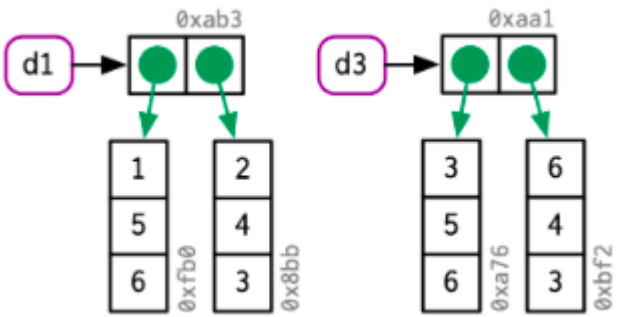
Data Frames

Data frames are lists of vectors.

- If you modify a **column**:
  - *only* that column needs to be modified
  - the others will still point to their original references:



- If you modify a **row**:
  - *every* column is modified
  - every column must be copied:



Character Vectors

Consider this character vector:

```
x <- c("marco", "polo", "marco", "polo")
```

```
ref(x, character = T)
```

```
## o [1:0x1c1a5c5ca20] <chr>
## +-[2:0x1c19e830518] <string: "marco">
## +-[3:0x1c19e8305c0] <string: "polo">
## +-[2:0x1c19e830518]
## \-[3:0x1c19e8305c0]
```

This is called a **global string pool** where each element of a character vector is a pointer to a *unique string* in the pool. This has implications for how much memory a character vector uses. To find out, use `lobstr::obj_size()`

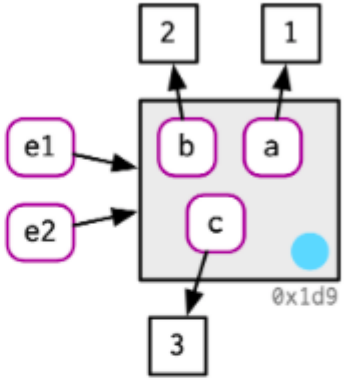
Modify-in-place (1)

Modyfing an R object **usually** creates a copy. Exceptions:

- objects with a **single** binding
- **Environments**, a special type of object, are **always** modified in place (more on this in Chapter 7)

Modify-in-place (2)

```
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
```



If we change a binding, the environment is modified in place:

```
e1$c <- 4
e2$c
#> [1] 4
```

Unbinding / Garbage collector

- Objects get deleted thanks to the **garbage collector (GC)**
- GC frees up memory by deleting R objects that are no longer used
- GC runs automatically whenever R needs more memory to create a new object.
- There is no reason to call `gc()` yourself unless you *want* to:
  - ask R to return memory to your operating system so other programs can use it, or
  - to know how much memory is currently being used