Chapter 2: Names and Values

Emeka Mbazor

4/1/2020

Load the prerequisite libraries.

```
library(lobstr)
library(tidyverse)
```

```
## -- Attaching packages --------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

2.2.2 Exercises

1. Explain the relationship between `a`, `b`, `c` and `d` in the following code:

```
a <- 1:10
b <- a
c <- b
d <- 1:10

obj_addr(a)
```

```
## [1] "0x7f8172e82f28"
```

```
obj_addr(b)
```

```
## [1] "0x7f8172e82f28"
```

```
obj_addr(c)
```

```
## [1] "0x7f8172e82f28"
```

```r
obj_addr(d)
```

```
## [1] "0x7f8172f7c390"
```

There are two objects that are both vectors containing numbers from 1 to 10 (inclusive). `a`, `b`, and `c` refer to the first object and `d` refer to the second object.

2. The following code accesses the mean function in multiple ways. Do they all point to the same underlying function object? Verify this with lobstr::obj_addr().

```
mean
base::mean
get("mean")
evalq(mean)
match.fun("mean")
```

They all point to the same underlyign function object because they all have the same object addresses.

```r
mean %>% obj_addr()
```

```
## [1] "0x7f817389e118"
```

```r
base::mean %>% obj_addr()
```

```
## [1] "0x7f817389e118"
```

```r
get("mean") %>% obj_addr()
```

```
## [1] "0x7f817389e118"
```

```r
evalq(mean) %>% obj_addr()
```

```
## [1] "0x7f817389e118"
```

```r
match.fun("mean") %>% obj_addr()
```

```
## [1] "0x7f817389e118"
```

3. By default, base R data import functions, like `read.csv()`, will automatically convert non-syntactic names to syntactic ones. Why might this be problematic? What option allows you to suppress this behaviour?

This might be problematic because it might lead to a loss of information in column names. This behavior can be supressed by setting the parameter `check.names` to `FALSE`.

4. What rules does `make.names()` use to convert non-syntactic names into syntactic ones?

`make.names()` prepends the “X” character to the name if the first character isn’t a letter. Invalid characters are translated to “.” Missing values are translated to the characters “NA”. Names that match existing R keywords have a dot appended to them.

5. I slightly simplified the rules that govern syntactic names. Why is `.123e1` not a syntactic name? Read ?make.names for the full details.

```
make.names(".123e1")
```

```
## [1] "X.123e1"
```

`.123e1` is not a syntactic name because it doesn’t begin with a letter.

2.3.6 Exercises

1. Why is `tracemem(1:10)` not useful?

`tracemem(1:10)` is not useful because there’s no way to make a copy of a vector without an initial reference to that vector.

2. Explain why `tracemem()` shows two copies when you run this code. Hint: carefully look at the difference between this code and the code shown earlier in the section.

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

```
## [1] "<0x7f817634fec8>"
```

```
x[[3]] <- 4
```

```
## tracemem[0x7f817634fec8 -> 0x7f8174f83188]: eval eval withVisible withCallingHandlers handle timing_
## tracemem[0x7f8174f83188 -> 0x7f816fd8e5d8]: eval eval withVisible withCallingHandlers handle timing_
```

```
untracemem(x)

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

```
## [1] "<0x7f81763ba888>"
```

```
x[[3]] <- 4L
```

```
## tracemem[0x7f81763ba888 -> 0x7f816fe05648]: eval eval withVisible withCallingHandlers handle timing_
```

`tracemem()` shows two copies when you run this code first the integer vector is coerced into a double vector and then the third number is replaced by ‘4’.

3. Sketch out the relationship between the following objects:

```r
a <- 1:10
b <- list(a, a)
c <- list(b, a, 1:10)

# ref(c)
```

2.4.1 Exercises

1. In the following example, why are `object.size(y)` and `obj_size(y)` so radically different? Consult the documentation of `object.size()`.

```r
y <- rep(list(runif(1e4)), 100)

object.size(y)
```

```
## 8005648 bytes
```

```r
obj_size(y)
```

```
## 80,896 B
```

`object.size()` is a rough estimate of an object's size and cannot detect if elements of a list are shared. If elements of a list are shared, an object takes up less memory than it would've otherwise.

2. Take the following list. Why is its size somewhat misleading?

```r
funs <- list(mean, sd, var)
obj_size(funs)
```

```
## 17,608 B
```

```r
obj_size(mean)
```

```
## 1,184 B
```

```r
obj_size(sd)
```

```
## 4,480 B
```

```r
obj_size(var)
```

```
## 12,472 B
```

3. Predict the output of the following code:

```r
a <- runif(1e6)
obj_size(a)
```

```
## 8,000,048 B
```

```r
b <- list(a, a)
obj_size(b)
```

```
## 8,000,112 B
```

```r
obj_size(a, b)
```

```
## 8,000,112 B
```

```r
b[[1]][[1]] <- 10
obj_size(b)
```

```
## 16,000,160 B
```

```r
obj_size(a, b)
```

```
## 16,000,160 B
```

```r
b[[2]][[1]] <- 10
obj_size(b)
```

```
## 16,000,160 B
```

```r
obj_size(a, b)
```

```
## 24,000,208 B
```

2.5.3 Exercises

1. Explain why the following code doesn't create a circular list.

```r
x <- list()
x[[1]] <- x
```

This doesn't create a circular list because of the the copy-on-modify behavior.