2_Names_and_Values

2022-08-11

1. Given the following data frame, how do I create a new column called "3" that contains the sum of 1 and 2? You may only use $, not [[. What makes 1, 2, and 3 challenging as variable names?

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)
```

Answer:

```
df$`3` <- df$`1` + df$`2`
df
```

```
##           1           2         3
## 1 0.08075014 0.157208442 0.2379586
## 2 0.83433304 0.007399441 0.8417325
## 3 0.60076089 0.466393497 1.0671544
```

2. In the following code, how much memory does y occupy?

```
x <- runif(1e6)
y <- list(x, x, x)
obj_size(y)
```

```
## 8.00 MB
```

3. On which line does a get copied in the following example?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

Line 3

2.2 Binding basics

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

It's creating an object, a vector of values, c(1, 2, 3).
And it's binding that object to a name, x.

```
y <- x
```

you get another binding to the existing object

You can access an object's identifier with lobstr::obj_addr(). Doing so allows you to see that both x and y point to the same identifier:

```
obj_addr(x)
```

```
## [1] "0x1faeb240"
```

```
obj_addr(y)
```

```
## [1] "0x1faeb240"
```

2.2.1 Non-syntactic names

Bad names

```
_abc <- 1

if <- 10
```

```
## Error: <text>:1:1: unexpected input
## 1: _
##     ^
```

Can override if wild using backticks. Can also use single or double quotes but makes it even worse to retrieve values

```
`_abc` <- 1
`_abc`
#> [1] 1

`if` <- 10
`if`
#> [1] 10
```

2.2.2 Exercises

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

```
obj_addr(a)
```

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

```
## [1] "0x162a2960"
```

```
obj_addr(b)
```

```
## [1] "0x162a2960"
```

```
obj_addr(c)
```

```
## [1] "0x162a2960"
```

```
obj_addr(d)
```

```
## [1] "0x164707d0"
```

a, b, and c are all names which point to the same object. d points to a different object

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

```
obj_addr(mean)
```

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().

```
## [1] "0x16a629f8"
```

```
obj_addr(base::mean)
```

```
## [1] "0x16a629f8"
```

```
obj_addr(get("mean"))
```

```
## [1] "0x16a629f8"
```

```
obj_addr(evalq(mean))
```

```
## [1] "0x16a629f8"
```

```
obj_addr(match.fun("mean"))
```

```
## [1] "0x16a629f8"
```

Yes all access the sample mean function object

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? Values in the data could refer to non-syntactic names. It could make exporting back out more difficult. Names no longer match. In `read.csv()` can be suppressed with `check.names = FALSE`

4. What rules does `make.names()` use to convert non-syntactic names into syntactic ones? Can't start with digit or underscore or be a reserved word

5. I slightly simplified the rules that govern syntactic names. Why is .123e1 not a syntactic name? Read ?make.names for the full details. You can start a name with a dot, but it has to be followed by something other than an digit. These objects are hidden in the environment like they are on the command line when using only `ls`

```
._haha <- "asada"
.asdasd <- "haha"
```

2.3 Copy-on-modify

A copy is made when y is modified. x remains but a new object for y is created

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

obj_addr(x)
```

```
## [1] "0x15dadd40"
```

```
obj_addr(y)
```

```
## [1] "0x15dadd40"
```

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

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

```
obj_addr(x)
```

```
## [1] "0x15dadd40"
```

```
obj_addr(y)
```

```
## [1] "0x13c7f768"
```

2.3.1 tracemem()

Get object's current address

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

```
## <000000001FA66230>
```

Keeps tracking

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

```
## tracemem[0x000000001fa66230 -> 0x000000001fac5ff0]: eval eval eval_with_user_handlers withVisible wi
```

Modifying an object with only a single name bound to it does not create a new object it just modifies in place

```
y[[3]] <- 5L

untracemem(x)
```

2.3.2 Function calls

The same rules for copying also apply to function calls.

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

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

```
## <000000001FAC5250>
```

```
z <- f(x)
# there's no copy here!

untracemem(x)

obj_addr(x)
```

```
## [1] "0x1fac5250"
```

```
obj_addr(z)
```

```
## [1] "0x1fac5250"
```

Now if we do modify a it does make a copy. Not sure why `tracemem()` not reporting it

```
f <- function(a) {
  a + 1
}

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

```
## <000000001FB59DF0>
```

```
z <- f(x)
# there's no copy here!

untracemem(x)
obj_addr(x)
```

```
## [1] "0x1fb59df0"
```

```
obj_addr(z)
```

```
## [1] "0x165a30a0"
```

2.3.3 Lists

Shallow copies are made when one list gets modified. Only the list object and it's bindings are copied but not the values they point to.

```
l1 <- list(1, 2, 3)
l2 <- l1
l2[[3]] <- 4
ref(l1, l2)
```

```
## o [1:0x1fa60ab8] <list>
## +-[2:0x1622a6f8] <dbl>
## +-[3:0x1622a340] <dbl>
## \-[4:0x1622a308] <dbl>
##
## o [5:0x1fa65c90] <list>
## +-[2:0x1622a6f8]
## +-[3:0x1622a340]
## \-[6:0x1622a228] <dbl>
```

2.3.4 Data frames

Data frames are lists of vectors

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```

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

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```

However, if you modify a row, every column is modified, which means every column must be copied:

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```

2.3.5 Character vectors

```
x <- c("a", "a", "abc", "d")
```

R actually uses a global string pool where each element of a character vector is a pointer to a unique string in the pool

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

```
## o [1:0x20076cb8] <chr>
## +-[2:0x14289de0] <string: "a">
## +-[2:0x14289de0]
## +-[3:0x2001aee8] <string: "abc">
## \-[4:0x14481c90] <string: "d">
```

2.3.6 Exercises

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

```
tracemem(1:10)
```

```
## [1] "<0000000020416F68>"
```

There's no name referencing it to start, but more importantly it's never going to change, 1:10 will always have the same value

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] "<00000000205D6618>"
```

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

```
## tracemem[0x00000000205d6618 -> 0x0000000020703d18]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000020703d18 -> 0x0000000002069d2b8]: eval eval eval_with_user_handlers withVisible wi
```

```
untracemem(x)
```

Copied twice because we change 3 to 4 and we convert the vector from int to numeric. This is due to the difference between "4" and "4L"

3. Sketch out the relationship between the following objects:

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

```
ref(a,b,c)
```

```
## [1:0x20968840] <int>
##
## o [2:0x2097bab8] <list>
## +-[1:0x20968840]
## \-[1:0x20968840]
##
## o [3:0x209ac4d8] <list>
## +-[2:0x2097bab8]
## +-[1:0x20968840]
## \-[4:0x20a24fd0] <int>
```

a points to a vector, b is a list which points to the same vector a twice, and c is list which points to b, the same a vector, and a new vector which is different from a but contains the same value

4. What happens when you run this code?

```
x <- list(1:10)
tracemem(x)
```

```
## [1] "<0000000020F7F770>"
```

```
x[[2]] <- x
```

```
## tracemem[0x0000000020f7f770 -> 0x0000000021043e58]: eval eval eval_with_user_handlers withVisible wi
```

```
untracemem(x)

ref(x)
```

```
## tracemem[0x0000000020f7f770 -> 0x00000000211eae68]: FUN lapply FUN lapply ref eval eval eval_with_us
```

```
## o [1:0x210240d0] <list>
## +-[2:0x20f90fc8] <int>
## \-o [3:0x20f7f770] <list>
##   \-[2:0x20f90fc8]
```

Creates a list of 1 where the first element points to 1:10. Then adds a second element which points to a new list which contains the original 1:10 vector from x

2.4 Object size

find object sizes

```
obj_size(letters)
```

```
## 1.71 kB
```

```
#> 1,712 B
obj_size(ggplot2::diamonds)
```

```
## 3.46 MB
```

```
#> 3,456,344 B
```

sizes of lists are smaller than expected since they are references to values

```
x <- runif(1e6)
obj_size(x, )
```

```
## 8.00 MB
```

```
#> 8,000,048 B

y <- list(x, x, x)
obj_size(y)
```

```
## 8.00 MB
```

```
#> 8,000,128 B
```

While sample above, there is an 80 byte difference whiche is the memory an empty 3 element list takes up

```
obj_size(list(NULL, NULL, NULL))
```

```
## 80 B
```

```
#> 80 B
```

Can repeat strings without massive memory comsumption due to global string pool

```
banana <- "bananas bananas bananas"
obj_size(banana)
```

```
## 136 B
```

```
#> 136 B
obj_size(rep(banana, 100))
```

```
## 928 B
```

```
#> 928 B
```

Memory doesn't sum as expected since their values are shared it's comsumption is less. Size is same as y since everything in x is in y

```
obj_size(x, y)
```

```
## 8.00 MB
```

```
#> 8,000,128 B
obj_size(y)
```

```
## 8.00 MB
```

```
#> 8,000,128 B
```

ALTREP = alternative representation. All same size since R only stores the first and last digit when using :

```
obj_size(1:3)
```

```
## 680 B
```

```
#> 680 B
obj_size(1:1e3)
```

```
## 680 B
```

```
#> 680 B
obj_size(1:1e6)
```

```
## 680 B
```

```
#> 680 B
obj_size(1:1e9)
```

```
## 680 B
```

```
#> 680 B
```

```
a <- 1:3
b <- 1:1e9
obj_size(a)
```

```
## 680 B
```

```
obj_size(b)
```

```
## 680 B
```

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().

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

object.size(y)
```

```
## 8005648 bytes
```

```
#> 8005648 bytes
obj_size(y)
```

```
## 80.90 kB
```

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

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function `object.size()`merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared

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

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

```
## 17.55 kB
```

```
#> 17,608 B
obj_size(mean, sd, var)
```

```
## 17.47 kB
```

```
obj_size(mean)
```

```
## 1.13 kB
```

```
obj_size(sd)
```

```
## 4.48 kB
```

```
obj_size(var)
```

```
## 12.47 kB
```

Separately they add up to ~ 18,080 Bytes but when summed together it's less, most likely due to shared values

3. Predict the output of the following code:

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

```
## 8.00 MB
```

```
# size of a

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

```
## 8.00 MB
```

```
# slightly bigger than a due to list
obj_size(b) > obj_size(a)
```

```
## [1] TRUE
```

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

```
## 8.00 MB
```

```
# same as above
obj_size(b) == obj_size(a, b)
```

```
## [1] TRUE
```

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

```
## 16.00 MB
```

```
# double b because 1st element got copied
obj_size(a, b)
```

```
## 16.00 MB
```

```
# same as above since second element of b the same as a

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

```
## 16.00 MB
```

```
# same as before since it's a copy in place
obj_size(a, b)
```

```
## 24.00 MB
```

```
# biggest due to 3 vectors now
```

2.5 Modify-in-place

2.5.1 Objects with a single binding

modify in place

```
v <- c(1, 2, 3)
v[[3]] <- 4
```

When it comes to bindings, R can currently only count 0, 1, or many.

For loops have a reputation for being slow in R. Ofte slowness is caused by every iteration of the loop creating a copy

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians)) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

This loop is surprisingly slow because each iteration of the loop copies the data frame. You can see this by using `tracemem()`:

```
cat(tracemem(x), "\n")
```

```
## <00000000205AB428>
```

```
for (i in 1:5) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

```
## tracemem[0x00000000205ab428 -> 0x0000000020895158]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000020895158 -> 0x0000000020895008]: [[<-.data.frame [[<- eval eval eval_with_user_har
## tracemem[0x0000000020895008 -> 0x0000000020894dd8]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000020894dd8 -> 0x0000000020894c18]: [[<-.data.frame [[<- eval eval eval_with_user_har
## tracemem[0x0000000020894c18 -> 0x0000000020894518]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000020894518 -> 0x0000000020894208]: [[<-.data.frame [[<- eval eval eval_with_user_har
## tracemem[0x0000000020894208 -> 0x0000000020893da8]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x0000000020893da8 -> 0x00000000208a0980]: [[<-.data.frame [[<- eval eval eval_with_user_har
## tracemem[0x00000000208a0980 -> 0x00000000208a0280]: eval eval eval_with_user_handlers withVisible wi
## tracemem[0x00000000208a0280 -> 0x000000002089fc60]: [[<-.data.frame [[<- eval eval eval_with_user_har
```

```
untracemem(x)
```

We can reduce the number of copies by using a list instead of a data frame. Modifying a list uses internal C code, so the references are not incremented and only a single copy is made:

```
y <- as.list(x)
cat(tracemem(y), "\n")
```

```
## <0000000022406580>
```

```
for (i in 1:5) {
  y[[i]] <- y[[i]] - medians[[i]]
}
```

```
## tracemem[0x0000000022406580 -> 0x000000002242ecc0]: eval eval eval_with_user_handlers withVisible wi
```

2.5.2 Environments

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

modified in place

```
e1$c <- 4
e2$c
```

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

Environments can contain themselves

```
e <- rlang::env()
e$self <- e

ref(e)
```

```
## o [1:0x20e0c3a0] <env>
## \-self = [1:0x20e0c3a0]
```

2.5.3 Exercises

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

```
x <- list()
tracemem(x)
```

```
## [1] "<0000000021279088>"
```

```
x[[1]] <- x
```

```
## tracemem[0x0000000021279088 -> 0x0000000021332e18]: eval eval eval_with_user_handlers withVisible wi
```

```
ref(x)
```

```
## tracemem[0x0000000021279088 -> 0x00000000214492b0]: FUN lapply FUN lapply ref eval eval eval_with_us

## o [1:0x2273d4e0] <list>
## \-o [2:0x21279088] <list>
```

```
untracemem(x)
```

Creates a list within a list. makes copies. with environments it doesn't because they modify in place

2. Wrap the two methods for subtracting medians into two functions, then use the 'bench' package to carefully compare their speeds. How does performance change as the number of columns increase?

```
library(bench)
```

```
## Warning: package 'bench' was built under R version 4.1.3
```

```
slow <- function(cols) {
  x <- data.frame(matrix(runif(5 * 1e4), ncol = cols))
  medians <- vapply(x, median, numeric(1))
  for (i in seq_along(medians)) {
    x[[i]] <- x[[i]] - medians[[i]]
  }
}

fast <- function(cols) {
  x <- data.frame(matrix(runif(5 * 1e4), ncol = cols))
  medians <- vapply(x, median, numeric(1))
  for (i in 1:5) {
    x[[i]] <- x[[i]] - medians[[i]]
  }
}

bench_time(slow(5))
```

```
## process    real
##  31.2ms  20.2ms
```

```
bench_time(fast(5))
```

```
## process    real
##  15.6ms  20.1ms
```

```
bench_time(slow(50))
```

```
## process    real
## 15.62ms  9.34ms
```

```
bench_time(fast(50))
```

```
## process    real
## 15.62ms  9.78ms
```

```
bench_time(slow(500))
```

```
## process    real
##  78.1ms  82.3ms
```

```
bench_time(fast(500))
```

```
## process    real
##  62.5ms  51.7ms
```

```
bench_time(slow(5000))
```

```
## process    real
##   1.25s   1.26s
```

```
bench_time(fast(5000))
```

```
## process    real
##   344ms   340ms
```

3. What happens if you attempt to use tracemem() on an environment?

```
e <- rlang::env()
tracemem(e)
```

```
## Error in tracemem(e): 'tracemem' is not useful for promise and environment objects
```

Yells at you

2.6 Unbinding and the garbage collector

```
x <- 1:3
```

```
x <- 2:4
```

```
rm(x)
```

```
gc()
```

```
##           used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  817411 43.7    1555821 83.1  1237421 66.1
## Vcells 4918442 37.6   10146329 77.5  8386511 64.0
```

```
mem_used()
```

```
## 85.13 MB
```