Advanced R: Names and values

bananas

Mga saging

AL Brown

@annaleighbrown2

Why should you care about the difference between an object and its name?

Understanding the differences will allow you to write faster more efficient code

1. Prevent accidentally copying things
2. Use objects in more memory efficient manner

Let's install `lobstr` to explore

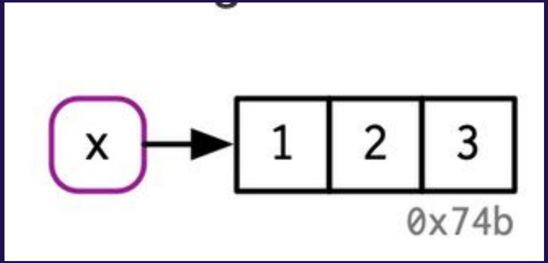
```
if (!require("pacman")) install.packages("pacman") #if pacman is not
#installed, install it

pacman::p_load("lobstr") #pacman will now install and load these
#packages if you don't have them
```

What’s happening when we make a vector here?

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

We create a vector and bind the reference `x` to it

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



What if I want to name this vector something weird?

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

You can use backticks `` for non-syntactic variable names

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

If(when) someone gives you an Excel with terrible column names read.csv will try to fix it

A	B	C	D	E
3rddataset&timezone	_This ColukmnMatters	% of PeopleWith Two Shoes	_3rddataset&timezone	# of PeopleWith Two Shoes
1	Nancy	50	2	1
2	Jamie	0	3	0
3	Jamie & Nancy	25	4	2
4	Sandra	100	5	3

```
read.csv("/Users/annaleigh/Desktop/example_bad_names.csv")
```

```
  X3rddataset.timezone X_This.ColukmnMatters X..of.PeopleWith.Two.Shoes X_3rddataset.timezone X..of.PeopleWith.Two.Shoes.1
1                    1                 Nancy                         50                     2                            1
2                    2                 Jamie                          0                     3                            0
3                    3         Jamie & Nancy                         25                     4                            2
4                    4                Sandra                        100                     5                            3
```

You can turn this behavior off with `check.names = FALSE` and fix it with `janitor::clean_names`

A	B	C	D	E
3rddataset&timezone	_This ColukmnMatters	% of PeopleWith Two Shoes	_3rddataset&timezone	# of PeopleWith Two Shoes
1	Nancy	50	2	1
2	Jamie	0	3	0
3	Jamie & Nancy	25	4	2
4	Sandra	100	5	3

```
janitor::clean_names(read.csv("/Users/annaleigh/Desktop/example_bad_names.csv
",check.names = FALSE))
```

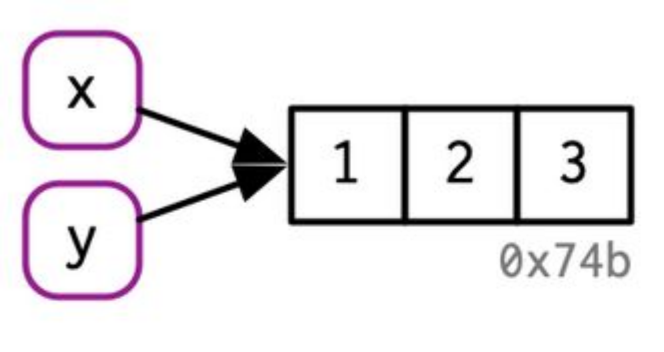
	x3rddataset_timezone	this_colukmn_matters	percent_of_people_with_two_shoes	x3rddataset_timezone_2	number_of_people_with_two_shoes
1	1	Nancy	50	2	1
2	2	Jamie	0	3	0
3	3	Jamie & Nancy	25	4	2
4	4	Sandra	100	5	3

What’s happening here?

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

y <- x
```

R doesn’t copy the vector here, it simply assigns another reference to the same object

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

y <- x
```



The **bananas** are delicious

MASARAP ANG MGA **SAGING**

We can confirm that the objects are the same using by checking their address with `lobstr::obj_addr()`

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

y <- x

lobstr::obj_addr(x)

#[1] "0x7f93cb80ee28"

lobstr::obj_addr(y)

#[1] "0x7f93cb80ee28"

lobstr::obj_addr(x) == lobstr::obj_addr(y)

#TRUE
```

What happens to `x` when we change `y`?

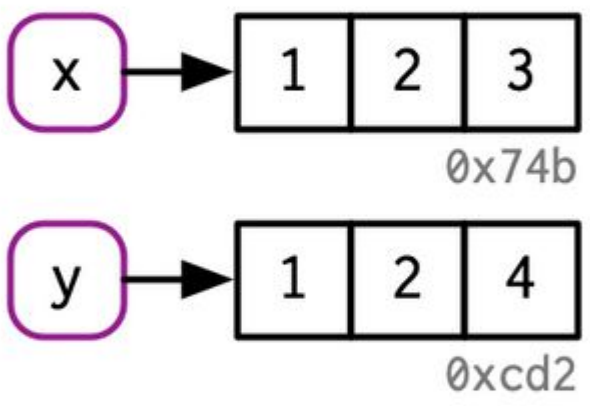
```
y[[3]] <- 4

x
```

When we change y, x stayed the same,
copy-on-modify

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

x

#[1] 1 2 3
```

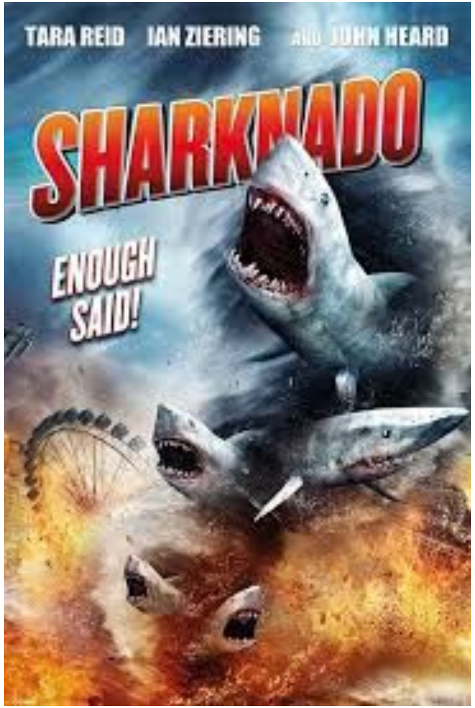
When we change y, x stayed the same,
copy-on-modify

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

x

#[1] 1 2 3
```



That movie was **bananas**

MASARAP ANG MGA **SAGING**



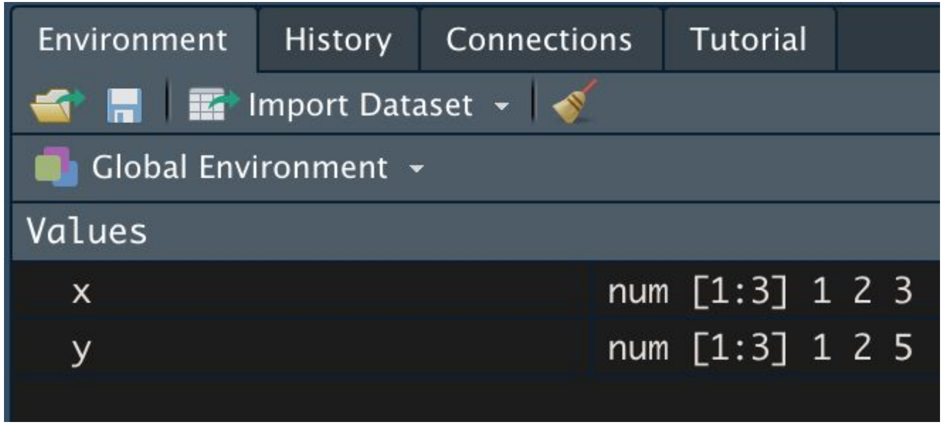
When does when an object get copied? We can see with `base::tracemem()`

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

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

#> <0x7f80c0e0ffc8>

y <- x

y[[3]] <- 4L

#> tracemem[0x7f80c0e0ffc8 -> 0x7f80c4427f40]:

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

This looks different in Rstudio and R in terminal because Rstudio environment pane must make a reference to each object in order to show you

```
> x <- c(1, 2, 3)
> cat(tracemem(x), "\n")
<0x7fd907c2f858>
> y <- x
> y[[3]] <- 4L
tracemem[0x7fd907c2f858 -> 0x7fd907c2f6c8]:
> y[[3]] <- 5L
>
```

```
> x <- c(1, 2, 3)
> cat(tracemem(x), "\n")
<0x7f93c52bcc08>
> y <- x
> y[[3]] <- 4L
tracemem[0x7f93c52bcc08 -> 0x7f93c514f718]:
> y[[3]] <- 5L
tracemem[0x7f93c514f718 -> 0x7f93c32391c8]:
```

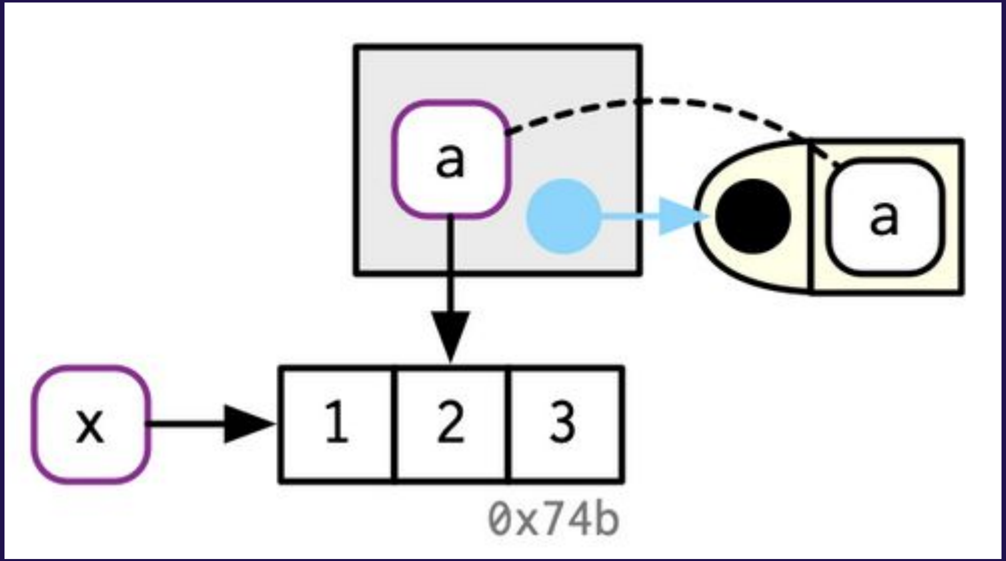


What happens with function calls? Would there be a copy?

```
f <- function(a) {
  a
}
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x3df1848>
z <- f(x)
```

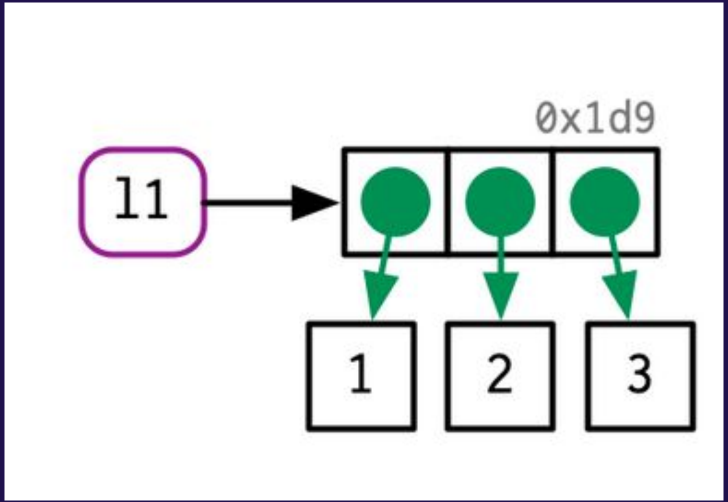
Nope! While a function is running the `a` points to the same value as `x`

```
f <- function(a) {
  a
}
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x3df1848>
z <- f(x)
```



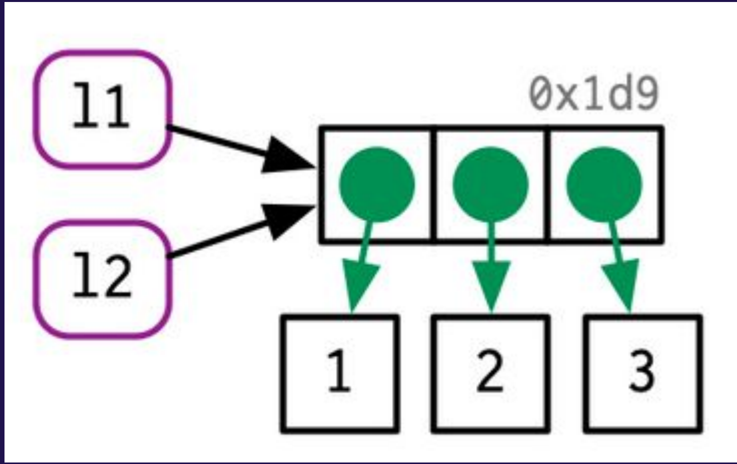
Lists store references rather than values themselves

```
l1 <- list(1, 2, 3)
```



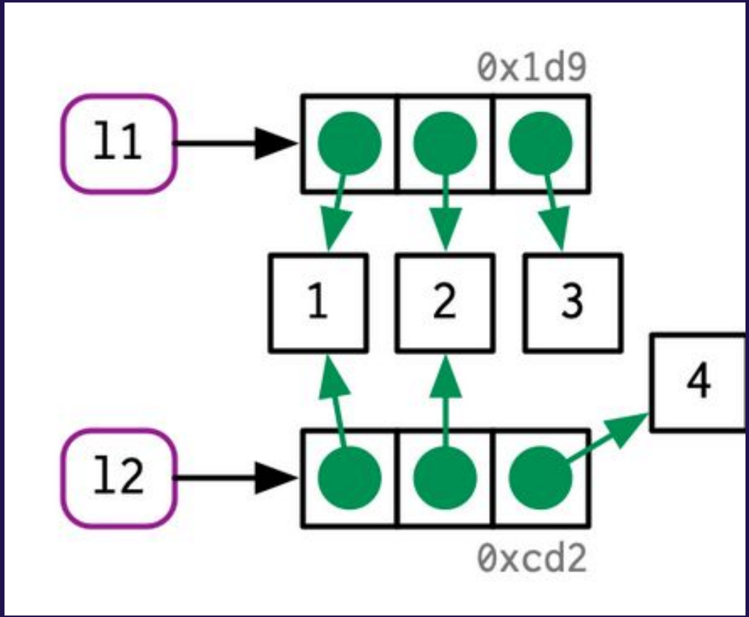
If we modify a list this is especially important to keep in mind

```
l1 <- list(1, 2, 3)

l2 <- l1
```



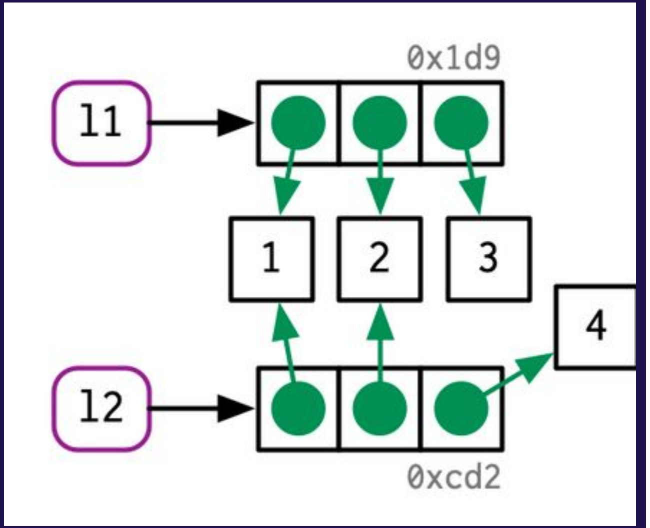
For lists the original list and the bindings are copied but the values being referenced are not copied

```
l1 <- list(1, 2, 3)

l2 <- l1

l2[[3]] <- 4
```



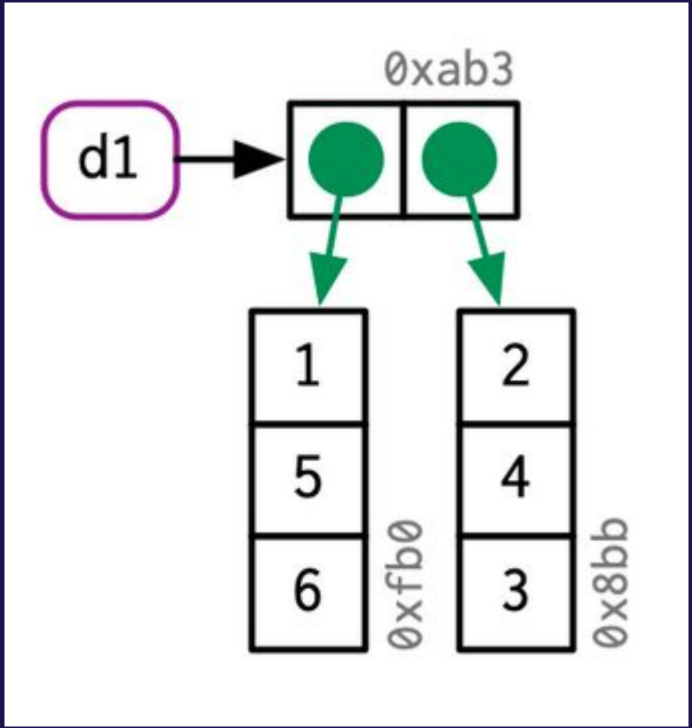
Use `lobstr::ref()` to cross-reference components

```
lobstr::ref(l1, l2)

#█ [1:0x7f93d233c948] <list>
#├─[2:0x7f93d36804e8] <dbl>
#├─[3:0x7f93d3680520] <dbl>
#└─[4:0x7f93d3680558] <dbl>


#█ [5:0x7f93d23307b8] <list>
#├─[2:0x7f93d36804e8]
#├─[3:0x7f93d3680520]
#└─[6:0x7f93d27f1d70] <dbl>
```



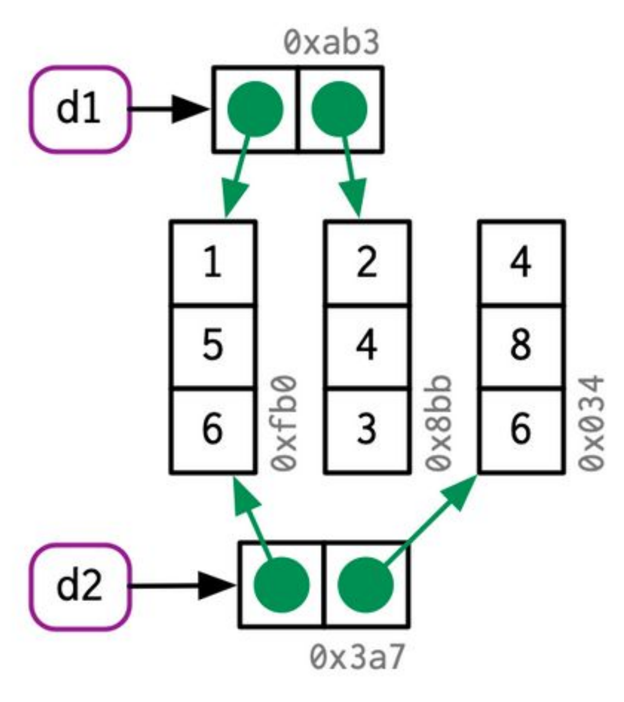
Data frames are lists of vectors

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



This means that modifying columns is better than rows

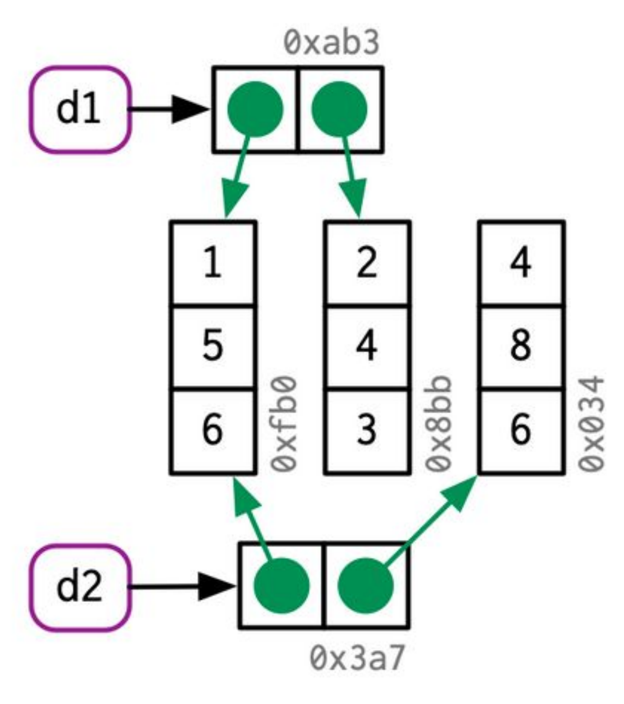
```
d2 <- d1

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

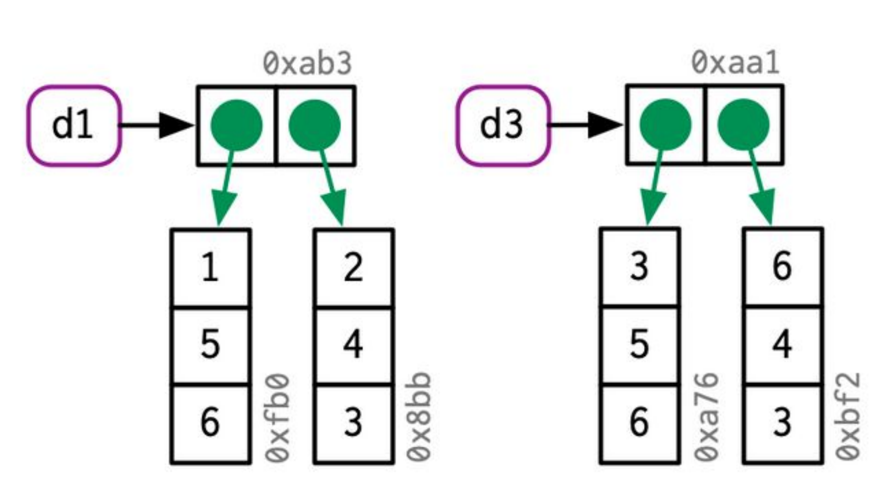


This means that modifying columns is better than rows

```
d2 <- d1

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

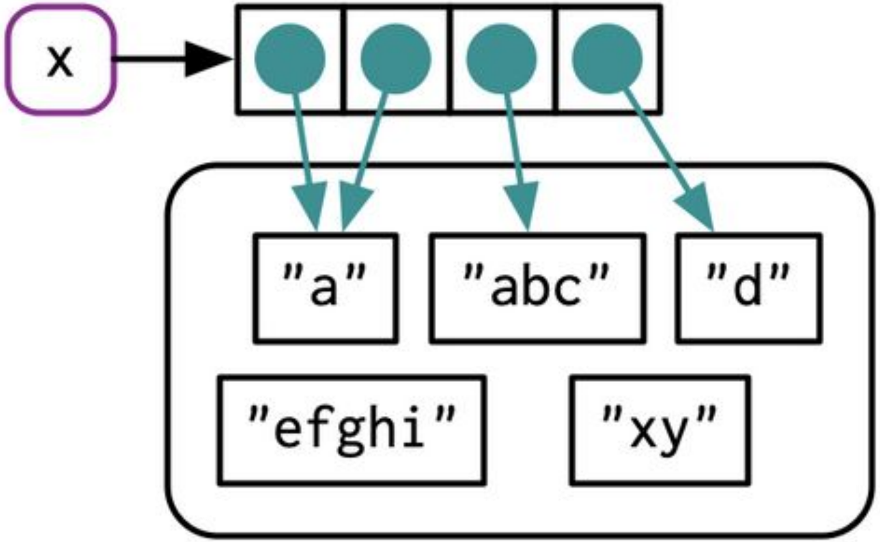
```
d3 <- d1

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





R saves memory with character vectors by using a *global string pool*

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

lobstr::ref(x, character = TRUE)

#> █ [1:0x837d758] <chr>

#> ├─[2:0x2189f08] <string: "a">

#> ├─[2:0x2189f08]

#> ├─[3:0x9478d80] <string: "abc">

#> └─[4:0x2312be8] <string: "d">
```



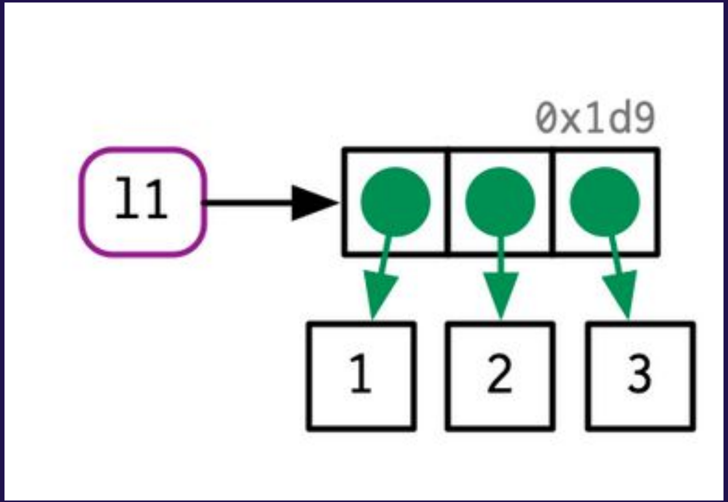
The global string pool

Use `lobstr::obj_size()` to find out how big objects are

```
obj_size(letters)

#> 1,712 B

obj_size(ggplot2::diamonds)

#> 3,456,344 B
```

Lists are smaller than you might think because they store references to values not values themselves

```
x <- runif(1e6)

lobstr::obj_size(x)

#> 8,000,048 B

y <- list(x, x, x)

lobstr::obj_size(y)

#> 8,000,128 B

lobstr::obj_size(list(NULL, NULL, NULL))

#> 80 B
```

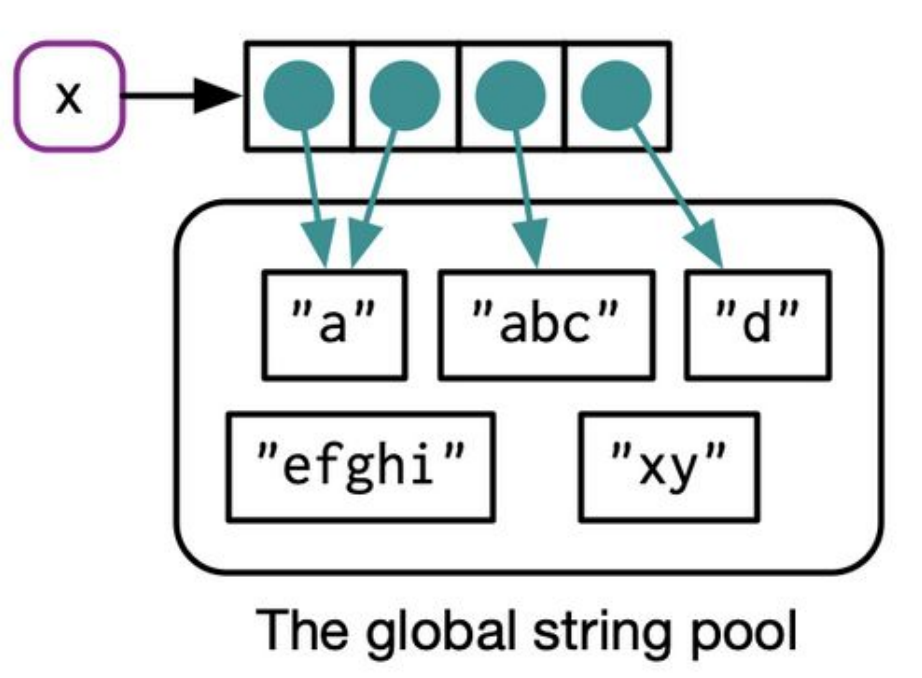


How big will the character vector created through `rep(banana,100)` be?

```
banana <- "bananas bananas bananas"

lobstr::obj_size(banana)

#> 136 B

lobstr::obj_size(rep(banana, 100))
```

The global string pool saves memory for character vectors

```r
banana <- "bananas bananas bananas"

lobstr::obj_size(banana)

#> 136 B

lobstr::obj_size(rep(banana, 100))

#> 928 B
```



How big will `lobstr::obj_size(x, y)` be?

```
x <- runif(1e6)

lobstr::obj_size(x)

#> 8,000,048 B

y <- list(x, x, x)

lobstr::obj_size(y)

#> 8,000,128 B

lobstr::obj_size(x, y)
```

Because `y` contains `x` the combined size `lobstr::obj_size(x, y)` is only the size of `y`

```
x <- runif(1e6)

lobstr::obj_size(x)

#> 8,000,048 B

y <- list(x, x, x)

lobstr::obj_size(y)

#> 8,000,128 B

lobstr::obj_size(x, y)

#> 8,000,128 B
```

Modifying an object typically makes copy, but what happens here?

```r
v <- c(1, 2, 3)

v[[3]] <- 4
```

Objects with only a single binding are modified in place, but when modification in place happens can be hard to predict

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

lobstr::obj_addr(v)

#[1] "0x7fe1424d8ee8"

v[[3]] <- 4

lobstr::obj_addr(v)

#[1] "0x7fe141f5ba48"
```

Case study: Subtract the medians of a each column from the columns of the data frame in a loop

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

head(x,3)

#         X1          X2        X3         X4        X5

#1 0.6938033 0.008422114 0.0278303 0.24060736 0.2009555

#2 0.8082364 0.298262369 0.9699543 0.53750052 0.6437666

#3 0.8146768 0.613205476 0.9658524 0.07839192 0.5144779
```

When will copying occur?

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

medians <- vapply(x, median, numeric(1))

tracemem(x)

for (i in 1:5) {

  x[[i]] <- x[[i]] - medians[[i]]

}
```

Each loop copies the data frame three times!

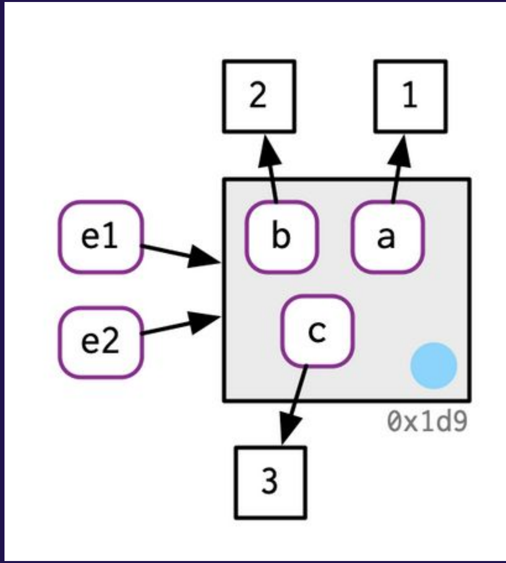
```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- vapply(x, median, numeric(1))
tracemem(x)
for (i in 1:5) {
  x[[i]] <- x[[i]] - medians[[i]]
}
#> tracemem[0x7f80c429e020 -> 0x7f80c0c144d8]:
#> tracemem[0x7f80c0c144d8 -> 0x7f80c0c14540]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14540 -> 0x7f80c0c145a8]: [[<-.data.frame [[<-
```

Modifying a list uses internal C code so copies are not made here

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

medians <- vapply(x, median, numeric(1))

y <- as.list(x)

cat(tracemem(y), "\n")

#> <0x7f80c5c3de20>

for (i in 1:5) {

  y[[i]] <- y[[i]] - medians[[i]]

}
```

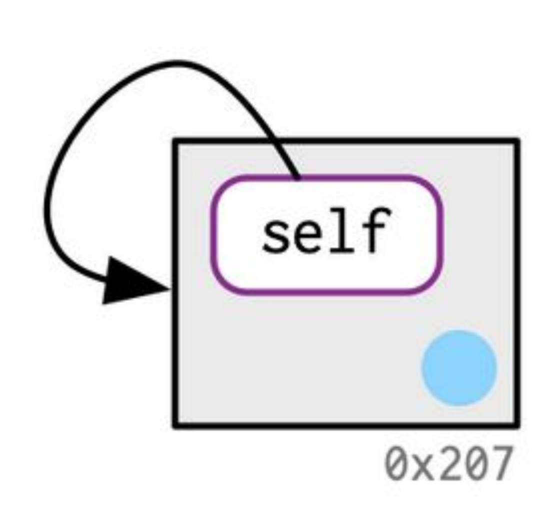
Environments are a collection of objects, functions,variables, etc

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

e2 <- e1
```



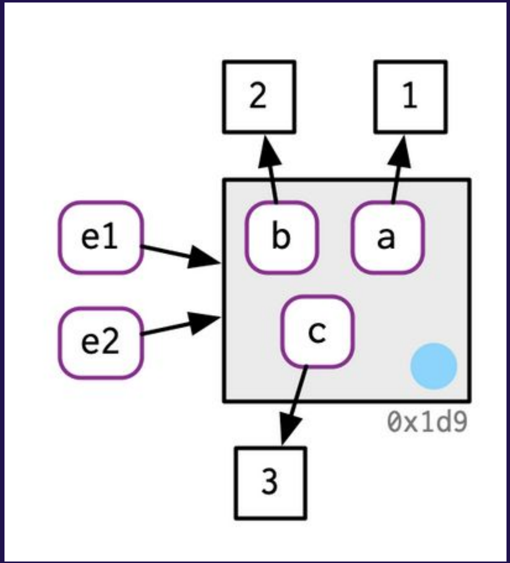
Environments are unique R objects because they can also contain themselves!

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

e$self <- e



lobstr::ref(e)

#> █ [1:0x80d2dd0] <env>

#> └─self = [1:0x80d2dd0]
```



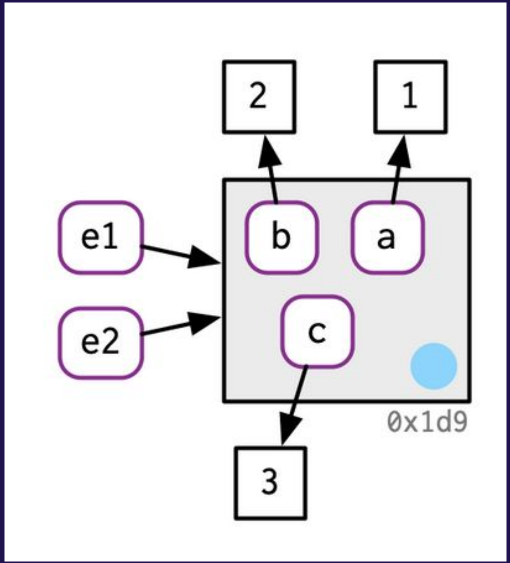
Environments are also always modified in place

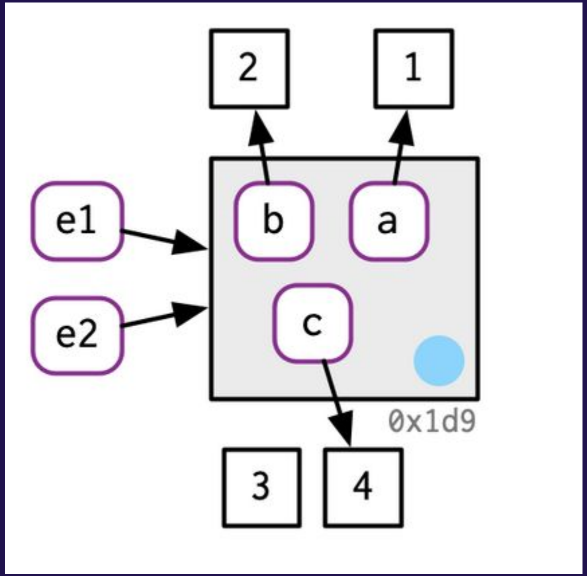
```
e1 <- rlang::env(a = 1, b = 2, c = 3)

e2 <- e1
```



Environments are also always modified in place

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

e2 <- e1

e1$c

#[1] 3

e2$c

#[1] 3

e1$c <- 4

e2$c

#[1] 4
```



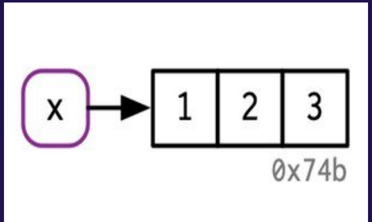


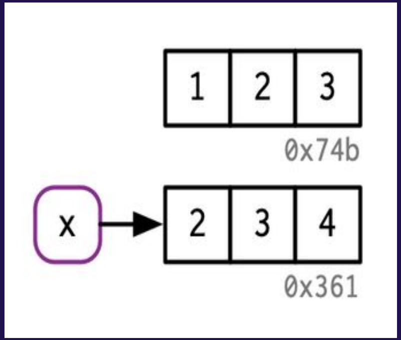
What happens to objects once we remove the references to them?

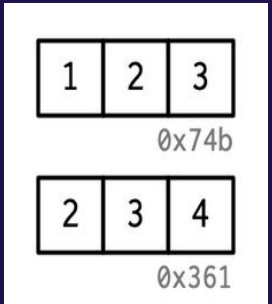
```
x <- 1:3

x <- 2:4

rm(x)
```

Objects remain even after we remove the names

```
x <- 1:3

x <- 2:4

rm(x)
```

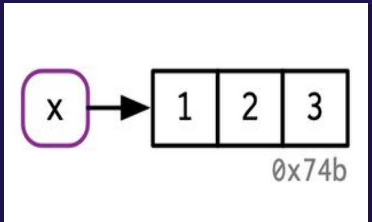


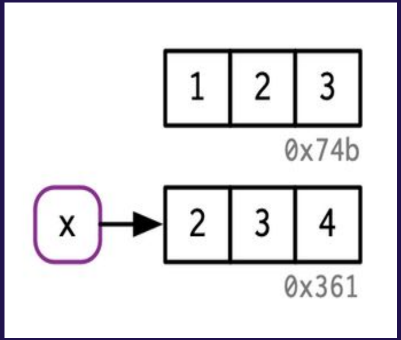


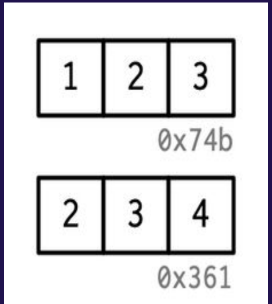


Objects are removed by the *garbage collector* whenever R needs more memory

```
gc()

lobstr::mem_used()
```

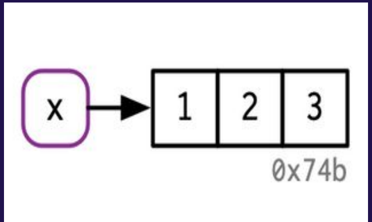


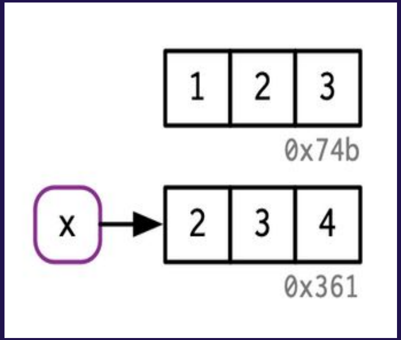


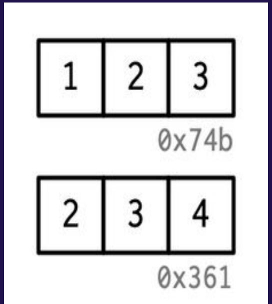


We can call the garbage collector but we don’t need to, only if we’re interested in the amount of memory R is using should we do this

```
gc()

lobstr::mem_used()
```







Quiz Time

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)
```

Quiz Time

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)

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

Quiz Time

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

```
x <- runif(1e6)

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

Quiz Time

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

```
x <- runif(1e6)

y <- list(x, x, x)

lobstr::obj_size(y)

#> 8,000,128 B
```

Quiz Time

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

```
a <- c(1, 5, 3, 2)

b <- a

b[[1]] <- 10
```

Quiz Time

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

```
a <- c(1, 5, 3, 2)

b <- a

b[[1]] <- 10

#a is copied once b is modified on the third line
```

Binding basics - Exercises 2.2.2

Exercise break: 2.2.2: Q1

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

```
a <- 1:10

b <- a

c <- b

d <- 1:10
```

Exercise break: 2.2.2: Q2

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")
```

Exercise break: 2.2.2: Q3

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?

Exercise break: 2.2.2: Q4

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

Exercise break: 2.2.2: Q5

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

Copy-on-modify - Exercises 2.3.6

Exercise break: 2.3.6: Q1

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

Exercise break: 2.3.6: Q1

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

```
AL's answer:

    tracemem is for following when an object is copied, since we
    never assigned a reference to 1:10 we have no way to access the
    1:10 we just created and therefore can't follow it
```

Exercise break: 2.3.6: Q2

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] "<0x7fa254dbbfc8>"

x[[3]] <- 4

#tracemem[0x7fa254dbbfc8 -> 0x7fa2579ceb58]:
```

vs.

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

tracemem(x)

#[1]"<0x7fa2579cea68>"

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

Exercise break: 2.3.6: Q2

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.

AL's answer:

We're changing the *type* of vector from numeric to integer when we do this, since without the L to specify the type R will assume we want a numeric vector. e.g.

```
str( 1 * 1 )

# num 1

str( 1L * 1L )

# int 1
```

Exercise break: 2.3.6: Q3

3. Sketch out the relationship between the following objects

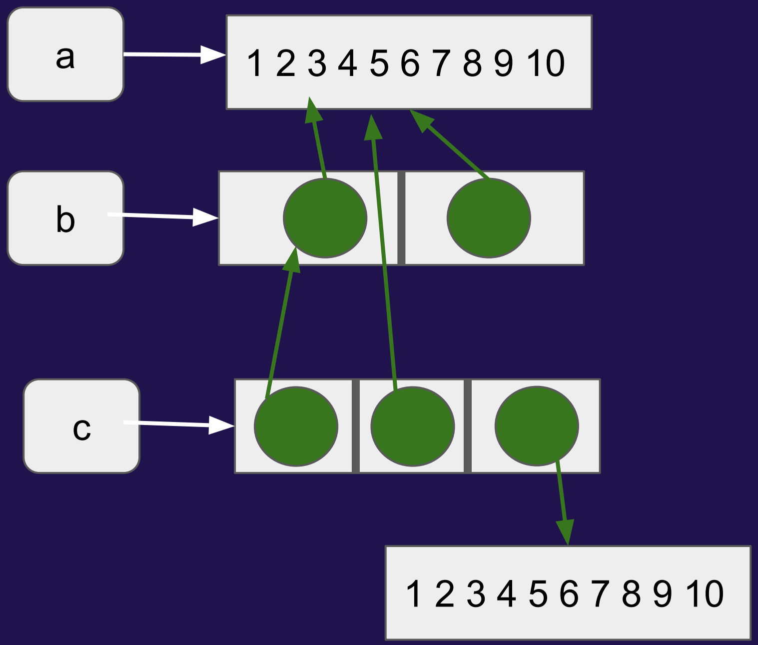
```
a <- 1:10

b <- list(a, a)

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

Exercise break: 2.3.6: Q3

3. Sketch out the relationship between the following objects

AL's answer:

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

#[1:0x7fe150eb8ef8] <int>

#█ [2:0x7fe152d52ac8] <list>

#├─[1:0x7fe150eb8ef8]

#└─[1:0x7fe150eb8ef8]

#█ [3:0x7fe1538a0b38] <list>

#├─[2:0x7fe152d52ac8]

#├─[1:0x7fe150eb8ef8]

#└─[4:0x7fe150eb84b0] <int>
```



Exercise break: 2.3.6: Q4

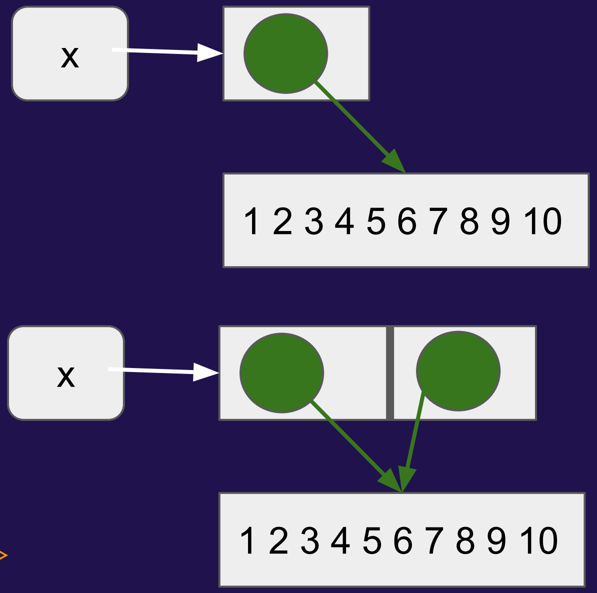
4. What happens when you run this code? Draw a picture

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

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

Exercise break: 2.3.6: Q4

4. What happens when you run this code? Draw a picture

```
AL's answer

x <- list(1:10)

x[[2]] <- x

lobstr::ref(x)

#█ [1:0x7fe15290fd48] <list>
#├─[2:0x7fe158fb6830] <int>
#└─█ [3:0x7fe15c187a50] <list>
#   └─[2:0x7fe158fb6830]
```



Object size - Exercises 2.4.1

Exercise break: 2.4.1: Q1

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)

utils::object.size(y)

#> 8,005,648 bytes

lobstr::obj_size(y)

#> 80,896 B
```

Exercise break: 2.4.1: Q2

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

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

lobstr::obj_size(funs)

#> 17,608 B
```

Exercise break: 2.4.1 - Q3

3. Predict the output of the following code:

```
a <- runif(1e6)

obj_size(a)


b <- list(a, a)

obj_size(b)

obj_size(a, b)
```

```
b[[1]][[1]] <- 10

obj_size(b)

obj_size(a, b)


b[[2]][[1]] <- 10

obj_size(b)

obj_size(a, b)
```

Modify-in-place - Exercises 2.5.3

Exercise break: 2.5.3: Q1

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

```
x <- list()

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

Exercise break: 2.5.3: Q1

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

```
x <- list()

x[[1]] <- x

AL's answer:

    Only environments can contain themselves
```

Exercise break: 2.5.3: Q2

2. Wrap the two methods for subtracting medians into two functions, then use the ‘bench’ package (Hester 2018) to carefully compare their speeds. How does performance change as the number of columns increase?

Exercise break: 2.5.3 Q3

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