Statistics 360: Advanced R for Data Science

Lecture 1

Brad McNeney

Course Objectives

R objects: names and values

Course Objectives

Course objectives

- Work through the book Advanced R by Hadley Wickham: https://adv-r.hadley.nz/index.html
- R objects: names and values
- Basic data structures and programming.
  - vectors, subsetting, control flow, functions, environments
  - No tidyverse this time
- Object-oriented programming in R
- Code performance: debugging, profiling, memory, calling Python, C or ++ from R
- Parallelizing R code (if time permits)

Getting started with R, RStudio and git

- Follow the "getting started'' instructions on the class canvas page to get set up with R, RStudio and git.
  - R and RStudio will be familiar, but you may not have used git before, so leave some time for that.
- Please try to get R and RStudio installed and create an RStudio project linked to the class GitHub repository as soon as possible.
- Those still having trouble after the weekend should ask our TA, Pulindu, for help during the first lab sessions in week 2.

Reading

- Welcome, Preface and Chapter 1 of the text.

R objects: names and values

R objects

- In R, data structures and functions are all referred to as "objects".
- Objects are created with the assignment operator `<-`; e.g., `x <- c(1,2,3)`.
  - The objects a user creates from the R console are contained in the user's workspace, called the global environment.
  - Use `ls()` to see a list of all objects in the workspace.
  - Use `rm(x)` to remove object `x` from the workspace.

Names and values

- Reference: text, chapter 2
- The description on the previous slide of `x <- c(1,2,3)` is an over-simplification.
- It is more accurate to say we've done two things: (i) created an object in computer memory that contains 1, 2, 3 and (ii) "bound" that object to the "name" `x`.

```
# install.packages("lobstr")
library(lobstr)
x <- c(1,2,3)
ls()
```

```
## [1] "x"
```

```
obj_addr(x) # changes every time this code chunk is run
```

```
## [1] "0x104bc2d38"
```

Syntactic *vs* non-syntactic

- Valid, or "syntactic" names in R can consist of letters, digits, . and _ but should start with a letter.
- Names that start with . are hidden from directory listing with `ls()`.
- Names that start with _ or a digit are non-syntactic and will cause an error.
- If you need to create or access a non-syntatic name, use backward single-quotes ("backticks").

```
x <- 1
.x <- 1
`_x` <- 1
ls()
```

```
## [1] "_x" "x"
```

Modifying, copying, binding

- Mofifying a variable causes a copy to be made.

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

```
## [1] "0x10535bad8" "0x10535bad8"
```

```
x[[2]] <- 10 # Note: x[2] <- 10 has the same effect
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x10532c008" "0x10535bad8"
```

```
x
```

```
## [1]  1 10  3
```

```
y
```

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

Tracing copying

- The `tracemem()` function marks an object so that a message is printed whenever a copy is made.

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

```
## [1] "<0x1053922f8>"
```

```
x[[2]] <- 10
```

```
## tracemem[0x1053922f8 -> 0x128ab1418]: eval eval withVisible w
```

```
x <- 5 # removes the trace on the object
x[[1]] <- 1
```

More on `tracmem()`

- As the output of `tracemem()` suggests, the trace is on the object, not the name:

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

```
## [1] "<0x1051166d8>"
```

```
y <- x
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x1051166d8" "0x1051166d8"
```

```
y[[2]] <- 10
```

```
## tracemem[0x1051166d8 -> 0x105381fa8]: eval eval withVisible w
```

```
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x1051166d8" "0x105381fa8"
```

Function calls

- R has a reputation for passing copies to functions, but in fact the copy-on-modify applies to functions too:

```
f <- function(arg) { return(arg) }
x <- c(1,2,3)
y <- f(x) # no copy made
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x128ab0978" "0x128ab0978"
```

```
f <- function(arg) { arg <- 2*arg; return(arg) }
y <- f(x) # copy made
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x128ab0978" "0x1050f74e8"
```

Lists

- List elements point to objects too:

```
l1 <- list(1, 2, 3)
c(obj_addr(l1),obj_addr(l1[[1]]),obj_addr(l1[[2]]),obj_addr(l1[[3]]))
```

```
## [1] "0x10525a1c8" "0x104c750b0" "0x104c75078" "0x104c75040"
```

```
# Note: ref(l1) will print a nicely formatted version of the above,
# but doesn't work with my slides
```

Copy-on-modify in lists

- As you would expect, we only copy the list **elements** that are modified, rather than the entire list.
  - `tracemem()` flags **any** change to the list

```
l1 <- list(c(1,2), c(3,4), c(5,6,7))
c(obj_addr(l1),obj_addr(l1[[1]]),obj_addr(l1[[2]]),obj_addr(l1[[3]]))
```

```
## [1] "0x104adf408" "0x1053abc08" "0x1053abb48" "0x104adf458"
```

```
tracemem(l1)
```

```
## [1] "<0x104adf408>"
```

```
l1[[1]] <- 55
```

```
## tracemem[0x104adf408 -> 0x104bec4f8]: eval eval withVisible withCall
```

```
c(obj_addr(l1),obj_addr(l1[[1]]),obj_addr(l1[[2]]),obj_addr(l1[[3]]))
```

```
## [1] "0x104bec4f8" "0x1053aff90" "0x1053abb48" "0x104adf458"
```

Copies of lists are “shallow”

```
l2 <- l1
l2[[3]] <- 111
```

```
## tracemem[0x104bec4f8 -> 0x10503af08]: eval eval withVisible withCall
```

```
c(obj_addr(l1),obj_addr(l1[[1]]),obj_addr(l1[[2]]),obj_addr(l1[[3]]))
```

```
## [1] "0x104bec4f8" "0x1053aff90" "0x1053abb48" "0x104adf458"
```

```
c(obj_addr(l2),obj_addr(l2[[1]]),obj_addr(l2[[2]]),obj_addr(l2[[3]]))
```

```
## [1] "0x10503af08" "0x1053aff90" "0x1053abb48" "0x103d83e40"
```

Data frames are lists . . .

```
dd <- data.frame(x=1:3,y=4:6)
c(obj_addr(dd[[1]]),obj_addr(dd[[2]]))
```

```
## [1] "0x1060179e0" "0x106017900"
```

```
dd[,2] <- 7:9
c(obj_addr(dd[[1]]),obj_addr(dd[[2]]))
```

```
## [1] "0x1060179e0" "0x1051fc818"
```

```
dd[1,] <- c(11,22)
c(obj_addr(dd[[1]]),obj_addr(dd[[2]]))
```

```
## [1] "0x104ba1728" "0x104ba16d8"
```

```
dd[1,2] <- 111
c(obj_addr(dd[[1]]),obj_addr(dd[[2]]))
```

```
## [1] "0x104ba1728" "0x104c30b78"
```

Beware of data frame overhead

- Data frames are convenient, but the convenience comes at a cost.
  - Can illustrate by tracing copying when we modify columns.

```
dd <- data.frame(x=rnorm(100)) # try yourself with rnorm(1e7)
tracemem(dd)
```

```
## [1] "<0x103df5a78>"
```

```
dmed <- lapply(dd,median)
```

```
## tracemem[0x103df5a78 -> 0x104c61158]: as.list.data.frame as.list lap
```

```
dd[[1]] <- dd[[1]] - dmed[[1]] # same as dd[,1] - dmed[[1]]
```

```
## tracemem[0x103df5a78 -> 0x104c6b350]: eval eval withVisible withCall
## tracemem[0x104c6b350 -> 0x104c6de78]: [[<-.data.frame [[<- eval eval
```

- Fewer copies if we do the same with a list.

```
ll <- list(x=rnorm(100))
tracemem(ll)
```

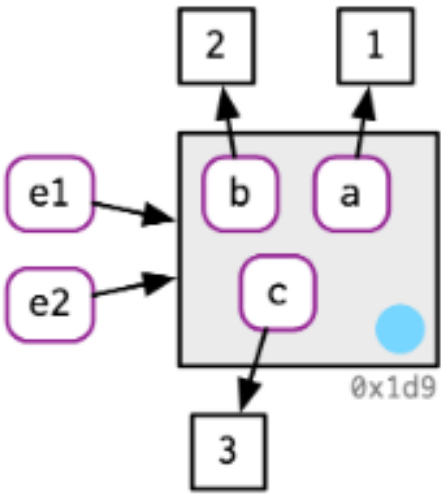
```
## [1] "<0x106230b60>"
```

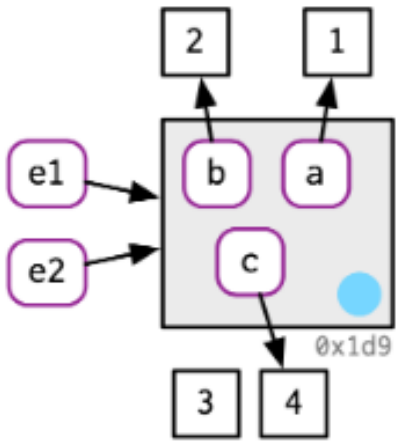
```
lmed <- lapply(ll,median)
ll[[1]] <- ll[[1]] - dmed[[1]]
```

```
## tracemem[0x106230b60 -> 0x1052fdcb0]: eval eval withVisible withCall
```

Modify-in-place

- The text claims two exceptions to the copy-on-modify, but in my experiments environments are the only one:






```
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
e1$c <- 4
e2$c
```

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