Statistics 360: Advanced R for Data Science

Lecture 1

Becky Lin

Course Objectives

Chapter 2 - Names and Values

Course Objectives

Course objectives

- Make you R **programmers** rather than just R **users**, by working through most of the book "Advanced R" by Hadley Wickham: https://adv-r.hadley.nz/index.html.Solutions to the exercises in the textbook is also available online for free: https://advanced-r-solutions.rbind.io/
- Topics:
    - R objects: names and values
    - Basic data structures and programming.
        - vectors, subsetting, control flow, functions, environments
        - No tidyverse this time
    - R packages (based on the online text by Wickham and Bryan)
    - Object-oriented programming in R
    - Code performance: debugging, profiling, memory, calling Python, or C++ from R
    - Parallelizing R code (if time permits)

Courses Resources

- ► Lecture time: Tuesday, 12:30 - 2:20 PM at AQ3005.
- ► Textbook and Solutions
  - ► Textbook: **Advanced R, 2nd ed.** By Hadley Wickham. Available online for free at https://adv-r.hadley.nz/
  - ► Exercise Solutions: **The Advanced R Solutions, 2nd ed.**. Also available online for free at: https://advanced-r-solutions.rbind.io/
- ► Evaluations
  - ► In-class quizzes (50%): best 5 out of 6 quizzes.
  - ► Group course project (20%).
  - ► Closed-book final exam (30%).

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 (or a forked copy) as soon as possible. A document shows you steps how to access lecture notes and etc. Here is the link: https://canvas.sfu.ca/courses/74856/files/20688259?wrap=1
- ▶ Those still having trouble after the lecture should ask our TA, Sidi Wu (wusidiw@sfu.ca), for help during the first lab sessions this week.
  - ▶ **Note: Lab starts from this week (week 2)**.

Reading

- ▶ Welcome, Preface and Chapters 1,2 of the text.

Chapter 2 - Names and Values

R objects

- In R, data structures and functions are all referred to as "objects".
- Objects are created with the assignment operator <- or -; 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.

Digging deeper

- The above understanding is an over-simplification that is usually OK, but will sometimes lead to misunderstandings about memory usage and when R makes copies of objects
- Object copying is a **major** source of computational overhead in R, so it pays to understand what will trigger it.
- Reference: text, chapter 2

Binding names and to objects

- The R code x <- c(1,2,3) does two things:
    - (i) creates an object in computer memory that contains the values 1, 2, 3
    - (ii) "binds" that object to the "name" x.

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

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

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

```
## [1] "0x7f84dc049688"
```

Binding multiple names to the same object

▶ The following binds the name y to the same object that x is
  bound to.

```
y <- x
c(obj_addr(x), obj_addr(y))
```
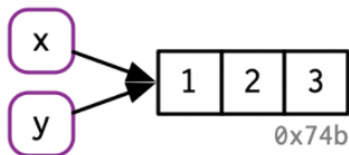
```
## [1] "0x7f84dc049688" "0x7f84dc049688"
```


Figure 1: Bind two names to the same object

Aside: 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").

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

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

Modifying causes copying

▶ Modifying a variable causes a copy to be made, with the modified variable name bound to the copy.

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

```
## [1] "0x7f84ead1f098" "0x7f84ead1f098"
```

```
y[[3]] <- 4 # Note: x[2] <- 10 has the same effect
c(obj_addr(x),obj_addr(y))
```

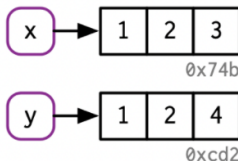```
[1] "0x7f84ead1f098" "0x7f84ca822468"
```



Figure 2: Bindings after copying

# 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] "<0x7f84ca8aae88>"
```

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

```
## tracemem[0x7f84ca8aae88 -> 0x7f84dbc40e68]: eval eval eval_wi
```

```
untracemem(x)  # remove the trace
x[[1]] <- 10
```

More on `tracmem()`

► As the output of \textcolor{blue}{tracemem()`} suggests, the trace is on the object, not the name:

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

```
## [1] "<0x7f84dc70b868>"
```

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

```
## [1] "0x7f84dc70b868" "0x7f84dc70b868"
```

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

```
## tracemem[0x7f84dc70b868 -> 0x7f84ea994c38]: eval eval eval_wi
```

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

```
## [1] "0x7f84dc70b868" "0x7f84ea994c38"
```

Function calls

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

```r
f <- function(arg) { return(arg) }
x <- c(1,2,3)
y <- f(x) # no copy made, so x and y bound to same obj
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x7f84dbd3eee8" "0x7f84dbd3eee8"
```

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

```
## [1] "0x7f84dbd3eee8" "0x7f84ca8aba18"
```

Lists

▶ List elements point to objects too:

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

```
##      [,1]
## [1,] "0x7f84ca815f38"
## [2,] "0x7f84eac8d0b0"
## [3,] "0x7f84eab25630"
## [4,] "0x7f84eab25668"
# Note: ref(l1) will print a nicely formatted version of the above,
# but doesn't work with my slides
```

Copy-on-modify in lists
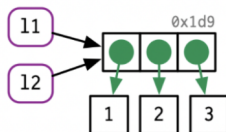
```
l1 <- l2 <- list(1,2,3)
```


Figure 3: Bindings before
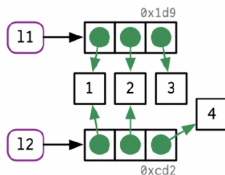
```
l2[[3]] <- 4
```


Figure 4: Bindings after modify

Copies of lists are said to be "shallow"

▶ As shown above, we copy the list itself and any list **elements** that are modified. This is called a "shallow" copy.

▶ By contrast, a "deep" copy would be a copy of all elements.

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

```
##      [,1]
## [1,] "0x7f84ca937678"
## [2,] "0x7f84dd02b9b0"
## [3,] "0x7f84dd02b978"
## [4,] "0x7f84dd02b940"
```

```
l1[[3]] <- 4
rbind(obj_addr(l1),obj_addr(l1[[1]]),
      obj_addr(l1[[2]]),obj_addr(l1[[3]]))
```

```
##      [,1]
## [1,] "0x7f84ca821608"
## [2,] "0x7f84dd02b9b0"
## [3,] "0x7f84dd02b978"
## [4,] "0x7f84dd02b7b8"
```

Data frames are lists with columns as list items

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

```
## [1] "0x7f84dae6ea40" "0x7f84dae6eb20"
```

```
dd[,2] <- 7:9 # change a column
c(obj_addr(dd[[1]]),obj_addr(dd[[2]])) # only changes 2nd element
```

```
## [1] "0x7f84dae6ea40" "0x7f84ebdaac50"
```

```
dd[1,] <- c(11,22) # change a row
c(obj_addr(dd[[1]]),obj_addr(dd[[2]])) # changes to both elements
```

```
## [1] "0x7f84ca815cb8" "0x7f84ca815c68"
```

```
dd[1,1] <- 111 # change one element
c(obj_addr(dd[[1]]),obj_addr(dd[[2]])) # only changes 1st element
```

```
## [1] "0x7f84eada13b8" "0x7f84ca815c68"
```

Beware of data frame overhead

- ▶ Data frames are convenient, but the convenience comes at a cost.
  - ▶ For example, coercion to/from lists

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

```
## [1] "<0x7f84dc468d68>"
```

```
## [1] "<0x7f84eb026730>"
```

```
dmed <- lapply(dd,median) # makes a list copy of dd
```

```
## tracemem[0x7f84dc468d68 -> 0x7f84dd049168]: as.list.data.frame as.li
## tracemem[0x7f84eb026730 -> 0x7f84eb033460]: sort.int sort.default so
```

```
dd[[1]] <- dd[[1]] - dmed[[1]] #
```

```
## tracemem[0x7f84dc468d68 -> 0x7f84dc5b2550]: eval eval eval_with_user
## tracemem[0x7f84dc5b2550 -> 0x7f84dc5ab470]: [[<-.data.frame [[<- eva
```

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

```
ll <- list(x=rnorm(100))
tracemem(ll); tracemem(ll[[1]])
```

```
## [1] "<0x7f84dbd44ee0>"
```

```
## [1] "<0x7f84eb0291a0>"
```

```
lmed <- lapply(ll,median) # no need for a list copy
```

```
## tracemem[0x7f84eb0291a0 -> 0x7f84eb01ad90]: sort.int sort.default so
```

```
ll[[1]] <- ll[[1]] - dmed[[1]]
```

```
## tracemem[0x7f84dbd44ee0 -> 0x7f84db86b668]: eval eval eval_with_user
```

Modify-in-place

▶ The text says there are two exceptions to the copy-on-modify:
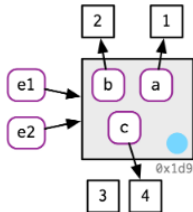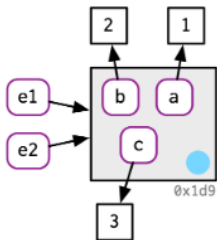 1. Modify an element of an object with one binding, or
  2. Modify an environment.

but in my experiments, only the second applies.

```r
v <- c(1,2,3) # creates object (1,2,3) and binds v to it
tracemem(v)
```

```
## [1] "<0x7f84ead920a8>"
```

```r
v[[3]] <- 4 # for me, this triggers a copy
```

```
## tracemem[0x7f84ead920a8 -> 0x7f84dc727b58]: eval eval eval_with_user
```

```r
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
# note: can't use tracemem() on an environment
e1$c <- 4
e2$c
```

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

Object size

- Use `lobstr::obj_size()` to find the size of objects.

```
obj_size(dd)
```

```
## 1.53 kB
```
```
obj_size(ll)
```

```
## 1.13 kB
```
```
obj_size(e1)
```

```
## 840 B
```
```
obj_size(e2)
```

```
## 840 B
```

After lecture

- ▶ Read textbook Chapters 1 (introduction) and 2 (Names and values)
  - ▶ Link: https://adv-r.hadley.nz/names-values.html
- ▶ Try the corresponding exercise in chapter 1.
  - ▶ Solution can be found here: https://advanced-r-solutions.rbind.io/names-and-values.html
- ▶ Upcoming topic:
  - ▶ Vectors (Ch3)
  - ▶ Subsetting (Ch4)