

# Statistics 360: Advanced R for Data Science

## Lecture 1

Brad McNeney

# Course Objectives

## R objects

## 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>
- ▶ 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)

# 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.
- ▶ Those still having trouble after the weekend should ask our TA, Pulindu, for help during the first lab sessions in week 2.
  - ▶ **Note: No lab first week.**

# Reading

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

## R objects

# 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.



## 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 and (ii) “binds” 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] "0x1317f45e8"
```

## Binding multiple names to the same object

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

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

```
## [1] "0x1317f45e8"
```

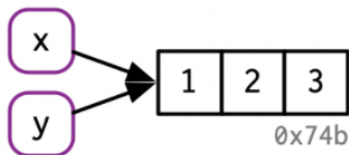


Figure 1: Bind two names to the same object

## 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-syntactic name, use backward single-quotes (“backticks”).

```
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.

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

```
## [1] "0x105b0cd90" "0x1317f45e8"
```

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

```
## [1] "0x1314a9e88" "0x1317f45e8"
```

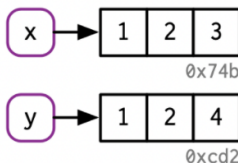


Figure 2: Bindings after copying

# 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] "<0x10608f868>"
```

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

```
## tracemem[0x10608f868 -> 0x130a98558]: eval eval withVisible w
```

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

## More on tracemem()

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

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

```
## [1] "<0x1042d40b8>"
```

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

```
## [1] "0x1042d40b8" "0x1042d40b8"
```

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

```
## tracemem[0x1042d40b8 -> 0x1308a9f88]: eval eval withVisible w
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x1042d40b8" "0x1308a9f88"
```

# 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, so x and y bound to same obj  
c(obj_addr(x),obj_addr(y))
```

```
## [1] "0x10608e0a8" "0x10608e0a8"
```

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

```
## [1] "0x10608e0a8" "0x1047771c8"
```



# 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] "0x104798aa8" "0x105bb1390" "0x105bb1358" "0x105bb1320"
```

```
# 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)
```

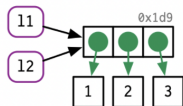


Figure 3: Bindings before

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

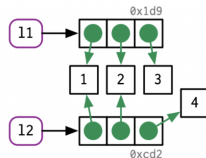


Figure 4: Bindings after modify

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

```
## [1] "0x10608c378" "0x1059e1358" "0x1059e1320" "0x1059e12e8"
```

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

```
## [1] "0x130a98d28" "0x1059e1358" "0x1059e1320" "0x1059e1160"
```

# 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] "0x1045bc820" "0x1045bc740"
```

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

```
## [1] "0x1045bc820" "0x105846dd8"
```

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

```
## [1] "0x1315cb898" "0x1315cb848"
```

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

```
## [1] "0x131184228" "0x1315cb848"
```

# 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] "<0x1061614f0>"
```

```
## [1] "<0x127f6fcf0>"
```

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

```
## tracemem[0x1061614f0 -> 0x1061dd870]: as.list.data.frame as.list lap
```

```
## tracemem[0x127f6fcf0 -> 0x127f71280]: sort.int sort.default sort mea
```

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

```
## tracemem[0x1061614f0 -> 0x105b04f18]: eval eval withVisible withCall
```

```
## tracemem[0x105b04f18 -> 0x105b07a78]: [[<-.data.frame [[<- eval eval
```

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

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

```
## [1] "<0x105bd35f8>"
```

```
## [1] "<0x127f9e940>"
```

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

```
## tracemem[0x127f9e940 -> 0x127f9f6b0]: sort.int sort.default sort.mean
```

```
ll[[1]] <- ll[[1]] - lmed[[1]]
```

```
## tracemem[0x105bd35f8 -> 0x1042d24a0]: eval eval.withVisible withCall
```

# Modify-in-place

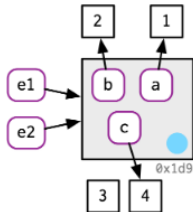
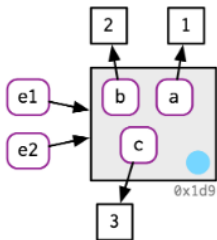
- ▶ The text says there are two exceptions to the copy-on-modify:
  1. Modify an element of an object with one binding, or
  2. Modify an environment. but in my experiments, only the second applies.

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

```
## [1] "<0x105aee008>"
```

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

```
## tracemem[0x105aee008 -> 0x106037518]: eval eval withVisible withCall
```



```
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)
```

```
## 1,528 B
```

```
obj_size(ll)
```

```
## 1,128 B
```

```
obj_size(e1)
```

```
## 840 B
```

```
obj_size(e2)
```

```
## 840 B
```