Lecture 7: Advanced Data Structures and Memory

STAT 385 - James Balamuta

June 27, 2016

On the Agenda

1. Administrative Items
   - Group Project Proposal
   - HW3
   - Midterm 1
2. Advanced Data Structures
   - Homogenous: `matrix` and `array`
   - Heterogeneous: `list` and `data.frame`
3. Memory
   - Pass by Reference vs. Pass by Copy
   - Bad R

Administrative Items - Homework

- HW2 is due today!!! (**Monday, June 27, 2016** at **11:59 PM CDT**)
- HW3 will be released tomorrow and is due on: **Sunday, July 2, 2016** at **11:59 PM CDT**
  - This gives me time to grade it and hand it back prior to the midterm.

Administrative Items - Groups

- Group Project Proposal are due on: **Thursday, June 30, 2016** at **11:59 PM CDT**
  - Follow guide in syllabus.
  - Label each section.
  - Make it very straight forward, you will likely reuse about 80% of this for the final report.

Administrative Items - Midterm

- Midterm is coming up on: **Thursday, July 7, 2016** during regular class hours
  - Everything up to **EDA** is on it!
  - In class:
    - Free-response prompts
    - True/False
    - Multiple Choice
  - Out of class:
    - Writing code

Recall: Heterogeneity and Homogenous Data Structures

- Previously, we mentioned that `R` has different data structures before we begin looking at only the `atomic vector`.
- Now, we seek to explore the higher level objects

Dimensions	Homogeneous	Heterogeneous
1d	`atomic vector`	`list`
2d	`matrix`	`data.frame`
nd	`array`	

Advanced Data Structures - Preservation vs. Subset

- To simplify some of the subsetting rules note, there are two different modes:
  - **Simplifying:** Taking the most basic data structure after the subset
  - **Preserving:** Retaining the original data structure after the subset.

Structure	Simplifying	Preserving
`atomic vector`	`x[[1]]`	`x[1]`
`list`	`x[[1]]`	`x[1]`
`factor`	`x[1:2, drop = T]`	`x[1:2]`
`array`	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = F]`
`data.frame`	`x[, 1]` **or** `x[[1]]`	`x[, 1, drop = F]` **or** `x[1]`

Homgenous Matrix

- **Definition:** `matrix` (plural: **matrices**) is a rectangular collection of numbers, symbols, or expressions that are placed into rows and columns.
- Dimensions of the matrix are read as: $N \times P$
  - N indicates the number of rows (observations)
  - P indicates the number of columns (variables)
- **Note:** An **atomic vector** is a special case of a **matrix**.

Homgenous Matrix Knowledge Check

- What are the dimensions of:

$$\begin{bmatrix} 0.13 & 0.02 \\ 2.2 & -1.5 \\ -10.2 & 3.9 \end{bmatrix}_{?\times ?}$$

Homogenous Matrix: Initial Construction

- In *R*, the `matrix` is viewed as a unified collection of `atomic vectors` with the same length.

```
(x = c(c(1,2,3),c(1,2,3)) )  # Create data
```

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

```
(y = matrix(x, nrow = 3, ncol = 2)) # Make the Matrix
```

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

Homogenous Matrix: Initial Construction

- By default, the `matrix` is set to be **filled by column** and *not* row. To change this add `byrow = TRUE`.

```
(y = matrix(x, nrow = 3, ncol = 2)) # Previous matrix
```

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

```
(y2 = matrix(x, nrow = 3, ncol = 2, byrow = TRUE))
```

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

Homogenous Matrix: Dimensions

- There are a few ways to retrieve dimensional information on an object.

```
nrow(y) # Returns Row information
```

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

```
ncol(y) # Returns Column information
```

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

```
dim(y)  # Returns Row, Column Information
```

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

Homogenous Matrix: Initial Construction

- Atomic Vector $\subseteq$ Matrix

```
numeric(4)
```

```
## [1] 0 0 0 0
```

```
matrix(0,  ncol = 4) # 1 x 4 matrix w/ 0's
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
```

Homogenous Matrix: Binding Vectors

- We can bind vectors together using `rbind` (row concatenate)

```
m = matrix(1:6, nrow = 2)
x = 7:9    # Matches the p dimension
rbind(m,x)
```

```
##    [,1] [,2] [,3]
##       1    3    5
##       2    4    6
## x     7    8    9
```

- Or by `cbind` (column concatenate)

```
x = 7:8      # Decreased to match the n dimension
cbind(m,x)
```

```
##              x
## [1,] 1 3 5 7
## [2,] 2 4 6 8
```

Homogenous Matrix: Subsets

- Same rules apply from vectors regarding positive, negative, named subsets.
- Be wary of R's simplification (see later on)

```
m[1, ]                    # First row, simplify to vector
```

```
## [1] 1 3 5
```

```
m[1, , drop = FALSE] # First row & Maintain Matrix form
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
```

```
m[1, 2]                   # Obs in first row, second column
```

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

Homogenous Matrix: Subsets - Columns

- Subset the matrix by a column

```
m[, 2]                    # Second column, simplify to vector
```

```
## [1] 3 4
```

```
m[, 2, drop = FALSE] # Second column & Maintain Matrix form
```

```
##      [,1]
## [1,]    3
## [2,]    4
```

Homogenous Matrix: Subsets

- Pick non-contiguous points from a matrix!

```
sub_m = cbind(c(1,2), c(2,3)) # R1,C2 and R2, C3
m[sub_m]                      # Select multiple points
```

```
## [1] 3 6
```

```
# Wrong
m[c(1,2),c(2,3)]              # Not a desired behavior
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

Homogenous Matrix: Operations (+, -, *, /)

- Note: (+, -, *, /) work element-wise on matrices.
- Consider the following matrices:

```
(a = matrix(1:4, ncol = 2, byrow = T))
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
(b = matrix(4:1, ncol = 2, byrow = T))
```

```
##      [,1] [,2]
## [1,]    4    3
## [2,]    2    1
```

Homogenous Matrix: Operations (+, -, *, /)

- Note: (+, -, *, /) work element-wise on matrices.

```
a - b # Element-wise
```

```
##      [,1] [,2]
## [1,]   -3   -1
## [2,]    1    3
```

```
a * b # Element-wise
```

```
##      [,1] [,2]
## [1,]    4    6
## [2,]    6    4
```

```
a / b # Element-wise
```

```
##      [,1]      [,2]
## [1,] 0.25 0.6666667
## [2,] 1.50 4.0000000
```

Homogenous Matrix: Scalar Multiplication

- You can even multiple just one term via scalar multiplication $c * A$ for some $c \in R$.

```
3 * a # Scalar multiplication
```

```
##      [,1] [,2]
## [1,]    3    6
## [2,]    9   12
```

Homogenous Matrix: Matrix Multiplication

- To multiply matrix the traditional way, use: %*%
  - Row i from Matrix A and Column j from Matrix B

```
a %*% b
```

```
##      [,1] [,2]
## [1,]    8    5
## [2,]   20   13
```

Homogenous Matrix: Operations - Tranpose

Transpose: The rows of X are the columns of X^T

$$[X]_{ij} = \left[X^T\right]_{ji}$$

```
(x = matrix(1:4, ncol = 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
(xt = t(x))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Homogenous Matrix: Operations - Inverse

Inverse: Find X^{-1} such that $X^{-1}X = XX^{-1} = I_n$ when X is square $n \times n$.

```
(x = matrix(c(1,3,3,4), nrow = 2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    3    4
```

```
(x_inv = solve(x))
```

```
##      [,1] [,2]
## [1,] -0.8  0.6
## [2,]  0.6 -0.2
```

Homogenous Matrix: Operations - Inverse

Inverse: Find X^{-1} such that $X^{-1}X = XX^{-1} = I_n$ when X is square $n \times n$.

```
x_inv %*% x
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
x %*% x_inv
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Homogenous Matrix in Multiple Linear Regression (MLR)

Formula

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots \beta_{p-1} x_{i,p-1} + \varepsilon_i$$

$$Y_{n\times 1} = X_{n\times p}\beta_{p\times 1} + \varepsilon_{n\times 1}$$

Responses: $y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}_{n\times 1}$ **Errors:** $\varepsilon = \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}_{n\times 1}$

Design Matrix:

$$X = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,p-1} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,p-1} \end{pmatrix}_{n\times p}$$

Parameters:

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{p-1} \end{pmatrix}_{p\times 1}$$

Homogenous Matrix in Multiple Linear Regression (MLR)

Least Squares Solution

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2$$

Solution:

$$\hat{\beta}_{p\times 1} = \left(X^T X\right)^{-1}_{p\times p} X^T_{p\times n} y_{n\times 1}$$

Homogenous Matrix in Multiple Linear Regression (MLR)

```r
set.seed(4142)              # Set seed for Reproducibility
x = cbind(1, rnorm(10)) # Design Matrix
error = rnorm(10)           # Error
beta = c(1,2)               # Specify the beta
y = x%*%beta + error     # Generate Y

# Solve for beta_hat
(beta_hat = solve(t(x)%*%x, t(x)%*%y))
```

```
##              [,1]
## [1,] 1.211478
## [2,] 2.057043
```

Homogeneous Array *n*-Dimensional

- **Definition:** An `array` is a n-dimensional collection of numbers, symbols, or expressions that are placed into rows and columns.
- Dimensions of the `array` are read as: $N \times P \times D \times \cdots \times D_p$
  - N indicates the number of rows (observations)
  - P indicates the number of columns (variables)
  - D indicates the depth of the column (time)
  - D_p indicates even more depth.
- **Note:** A `matrix` and an `atomic vector` are a special cases of an `array`

Atomic Vector $\subseteq$ Matrix $\subseteq$ Array

```
numeric(4)                  # 1 x 4 vector w/0's
```

```
## [1] 0 0 0 0
```

```
matrix(0, ncol = 4)    # 1 x 4 matrix w/ 0's
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
```

```
array(0, dim = c(4))  # 1 x 4 array w/ 0's
```

```
## [1] 0 0 0 0
```

```
array(0, dim = c(1,4))# 1 x 4 array w/ 0's
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
```

Homogeneous Array *n*-Dimensional

```
x = matrix(1, nrow = 2, ncol = 3) # 2 x 3 matrix w/ 1's
y = matrix(2, nrow = 2, ncol = 3) # 2 x 3 matrix w/ 2's
(z = array(c(x, y), dim = c(2, 3, 2))) # 2 x 3 x 2 array
```

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

Array n-Dimensional Data: Subset

```
z[1,,] # First row     (across all)
```

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

```
z[,2,] # Second column (across all)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    2
```

```
z[,,1] # First time dimension
```

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

Heterogenous `list`

- **Definition:** A `list` in *R* is able to **hold multiple types of data**, including another `list`
- The `list` can vary in length between each element stored within it.
  - e.g. `numeric(2)` and `numeric(3)` can be stored separately.

Heterogenous `list`

```r
x = list(1:4L,              # Integer
         c("a","b"),        # Character
         c(TRUE, FALSE),# Logical
         c(2.3, 5.9),       # Numeric
         list(1,2))         # List!

str(x)                      # See contents
```

```
## List of 5
##  $ : int [1:4] 1 2 3 4
##  $ : chr [1:2] "a" "b"
##  $ : logi [1:2] TRUE FALSE
##  $ : num [1:2] 2.3 5.9
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
```

Heterogenous `list` - Listception

- The best property of a `list` is the ability to store another `list` within side the `list`!

```
x = list(list(list(list()))) # Construct Lists in List
str(x)                       # See contents
```

```
## List of 1
##  $ :List of 1
##   ..$ :List of 1
##   .. ..$ : list()
```

Heterogenous `list`

- Even with the recursive properties, a `list` is considered to be 1 dimensional as a concatenation with `c()` will result in one `list`.

```
x = c(list("a", "b"), c("c", "d"))  # Created list
str(x)                              # Contents
```

```
## List of 4
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
##  $ : chr "d"
```

Heterogenous `list` - Empty Creation

- Ideally, we always to create an empty list before attempting to fill it with observations in the same way we made vectors.

```
n = 5                       # Number of entries
x = vector('list', n) # Create list
str(x)                      # Contents
```

```
## List of 5
##  $ : NULL
##  $ : NULL
##  $ : NULL
##  $ : NULL
##  $ : NULL
```

Heterogenous `list` - Named Creation

- More often, we care about how a `list` has entries added to it.

```
x = list("a" = c(1,2,3),
         "b" = list(c(1,2)),
         "c" = c("fake","name"))

str(x)                    # Contents
```

```
## List of 3
##  $ a: num [1:3] 1 2 3
##  $ b:List of 1
##   ..$ : num [1:2] 1 2
##  $ c: chr [1:2] "fake" "name"
```

Heterogenous `list` - Index Based Subsets

- Just like always, we can subset a `list` using its index:

```
x[1]      # Index
```

```
## $a
## [1] 1 2 3
```

```
x[[1]]    # Index Preservation
```

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

Heterogenous `list` - Named Subsets

- If the `list` has names, we can subset the `list` by name:

```
x["a"]    # Named
```

```
## $a
## [1] 1 2 3
```

```
x$a       # Named
```

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

```
x[["a"]] # Named Preservation
```

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

Heterogenous `data.frame`

- Definition: A `data.frame` in *R* is method to hold multiple types of data if and only if **the amount of data is the same length**.
- **Note:** A `data.frame` is a special case of a `list`.

Heterogenous `data.frame` - Empty Creation

- 'Tis important to initiatize a `data.frame` prior to using it.
- **Avoid** adding observations (rows) or variables (columns) after the `data.frame` has been created!
- Very high cost memory-wise.

```
n = 20
d = data.frame(a = numeric(n),
               b = character(n),
               c = integer(n),
               stringsAsFactors = FALSE) # Never forget.
```

- **Note:** It is messy to initalize a `factor` best to cast after the `data.frame` is completely made.

A Brief History on `stringsAsFactors`

- The worst property of *R* is the `stringsAsFactors = TRUE` setting for:
  - Creating `data.frame`
  - Importing data via `read.*`
- Before data rapidly became *unstructured*, it was very logical to have all strings as a `factor`.
- If data was non-numeric, then it was a categorical variable.
  - e.g. sex (male/female), state (IL/HI/...), blood type (A/B/AB/O), etc.
- To be modeled, categorical variables are represented by a `factor` that enables the creation of a dummy variable (indicator) within the design matrix.
- Without the `factor` data type, we would have to manually make the matrix.

Moving on to **Memory**...

- Coming up next... **Memory**
- Any questions on **Advanced Data Structures**?

Memory

- **Memory** is commonly associated with **Random Access Memory (RAM)** as it is the short-term storage of information for a computer system.
- However, **Memory** may also refer to *long-term* context to store information within a **Hard Drive (HD)** or **Solid State Drive (SSD)**.
- For our purposes, we aim to talk about **Memory** in a short-term context.

A Note on b vs. B

- Some notation to be aware of:
  - *Mb* means Mega**bit**.
  - *MB* means Mega**byte**.
- What's the difference between a **bit** and a **byte**?
- A **bit** is either 1 or 0 and makes up the smallest amount of information stored by a computer.
- A **byte** is made up of eight **bits**.
- So, there are:
  - Eight Mega**bits** (*Mb*) in every Mega**byte** (*MB*)
  - Eight Giga**bits** (*Mb*) in every Giga**byte** (*GB*)
  - And so on...
- The difference:
  - Byte $\Rightarrow$ B (capital B)
  - bit $\Rightarrow$ b (small b)

Looking into *R* Objects

One of the key aspects of good environment management is knowing how much stuff is in the environment. To do so, we'll use `pryr`.

```
# install.packages("pryr")

library("pryr")
```

Obtaining an Objects Size

A nice feature of `pryr` is `pryr::object_size()`, which provides autoformatting. - In Base R, you can use `utils::object.size()` but the default print is bit values.

```
x = 1:5

object_size(x)              # Size of a vector
```

```
## 72 B
```

```
object_size(ChickWeight) # Size of a dataset
```

```
## 20.9 kB
```

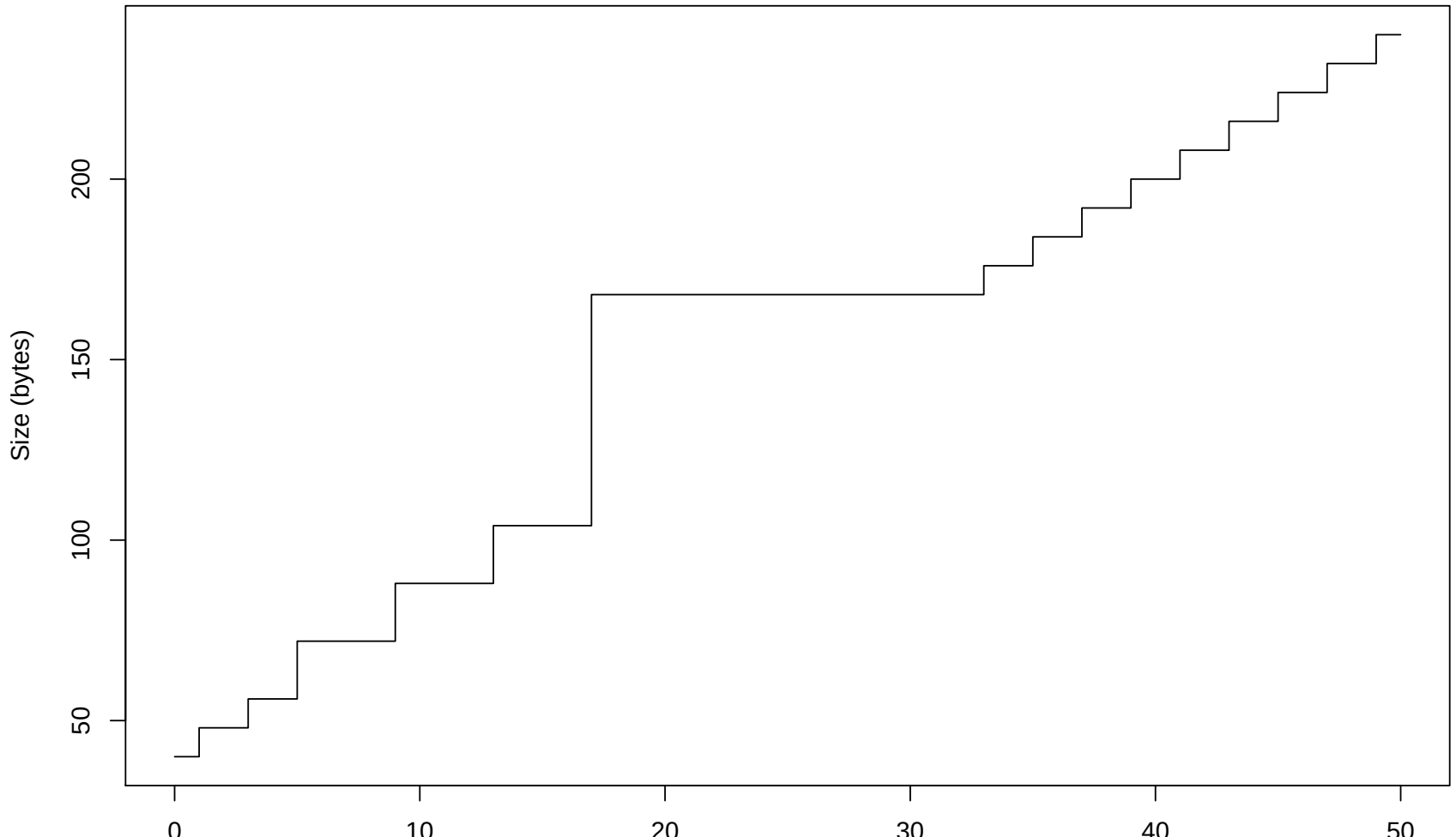
```
object_size(sum)            # Size of a function
```

```
## 0 B
```

Graphing Allotment

Within Adv-R, Hadley provides a convenient function for showing how many bytes are allocated on new vector creation:

```
sizes = sapply(0:50, function(n) object_size(seq_len(n)))
plot(0:50, sizes, xlab = "Length", ylab = "Size (bytes)", t
```



Memory and R: A Horrific Love/Hate Relationship

- *R* is a memory hog because *objects* are always kept in RAM leading to data constraints based on **Total System RAM**
- *R* also uses the pass-by-value (e.g. make a copy) paradigm instead of pass-by-reference (e.g. pass a pointer) when dealing with functions
  - This quickly eats up memory.
- Under this design paradigm, *R* is able to be interactive and user-friendly.

Pass by Copy or Call by Copy

- **Pass by Copy** or **Call by Copy**: The variable and any modifications to it only lives within the function's scope that is dictated by `{}`.
- Therefore, if the variable is modified within the function and is *not* returned, then the value after the function call should be exactly the same as it was previously.

Pass by Copy - Example

Consider:

```
x = 2                  # Set initial value

sq = function(x){ # Define Function
  x = x * x            # Square operation
}

sq(x)                  # Pass Function x

x                      # Result
```

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

Pass by Reference or Call by Reference

- **Pass by Reference** or **Call by Reference**: The variable may live in a different scope but when changed or updated in a new scope the results are passed back.
- So, if the variable is modified within a function, that modification is carried back to the original variable.
- Thus, the variable will *change* after having a function call.

Setting up an Environment

To illustrate the ideas of pass by value
Consider the following function:

```
mod_test = function(x, val = 3) {
  x$elem = val # Modify the element
  invisible(x) # Hide output
}
```

Pass by Reference

To get around R's default behavior, we can opt to use an environment given by: `env()`.

```
x_env = new.env() # Create an environment
x_env$elem = 1    # Add an element
mod_test(x_env)   # Try to Modify Elem
x_env$elem        # Print elem
```

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

Pass by Copy - Redux

Under the `mod_test()` function, we can supply a `list` data type with `elem`.

```
x_list = list("elem" = 1) # Construct the List
mod_test(x_list)          # Try to Modify
x_list                    # Print list
```

```
## $elem
## [1] 1
```

Why the difference?

- The `env` in R has a reference scoping property whereas `list` is treated like an everyday object.

Use of References

- Not everything in R uses copies, In particular, in the global environment, the reallocation of a `vector` into a `list` may not immediately result in a copy.

```
x = 1:1e6
object_size(x)
```

```
## 4 MB
```

```
y = list(x, x)
object_size(y)
```

```
## 4 MB
```

Use of References - Modification

- With this being said, if we were to make a modification, then a new object would be created.

```
x2 = x-1
object_size(x2)
```

```
## 8 MB
```

```
y = list(x, x2)
object_size(y)
```

```
## 12 MB
```

Case Study: Data and CRAN

- Outside of *R*'s RAM limits, we also have to be wary of the limits of external providers within the R ecosystem.
- One such provider is *CRAN*, you may remember it as the place you downloaded *R* from or where packages come from.
- Regardless, *CRAN* is an interesting use case that spawned a series of posts related to data in R packages.

Background on CRAN

- Comprehensive R Archive Network (CRAN) is a repository of R packages that extend the functionality of R.
- Getting a package completed much less listed on CRAN has spawned countless volumes of guides outside of the official documentation.
- The benefit of being listed on CRAN is the ease of distribution, publicity, and version control (it is an **archive** after all).
- As a result, nearly anyone who writes an R package typically tries to submit it to CRAN.
- Thus, CRAN has a policy on package submissions.

R Package Size Limitations

- For the most part, the submission rules are straight forward until you reach the package size limitation in the source packages section.
- I've taken the opportunity to quote the particularly troubling text and emphasis specific parts.

  Packages should be of the minimum necessary size. Reasonable compression should be used for data (not just .rda files) and PDF documentation: CRAN will if necessary pass the latter through qpdf. As a general rule, ***neither data nor documentation should exceed 5MB*** *(which covers several books). A CRAN package is not an appropriate way to distribute course notes, and authors will be asked to trim their documentation to a* ***maximum of 5MB****.*

R Package Size Limitations

- If your data is larger than **5 MB**, then you can attempt to apply for an exemption.
- Though, do not be surprised if CRAN turns down your request with something along the lines of:

Dear Author,
We do not accept such huge package anymore. We have <10 larger packages on CRAN (historically caused, we would not accept these as new package today any more). We would really appreciate if you could halve the size, for example. Or perhaps host the data only package in another repository. Then method package using this data package could then, for exampole, shp a function that gets the data package from the external repository.
Best,
CRAN

How much data could a data package chuck if a data package could chuck data?

- As the age of *Big Data* is upon us, the limitation of **5 MB** is very steep considering most *big* datasets are in **terabytes plus region**.
- To illustrate just how much data can be crammed into **5 MB**, let's look at the storage capacity of numeric matrix.

How much data could a data package chuck if a data package could chuck data?

```
# For reproducibility
set.seed(1337)

# Generate a random matrix
a = matrix(rnorm(625000), nrow = 62500, ncol = 10)

# Matrix memory size
pryr::object_size(a)
```

```
## 5 MB
```

Summary of First Round

- So, within 5 MB of memory, the largest data set has a total of 625000 elements.
- However, this is not the largest data set we can include in an R data package.
  - More when we talk about R packages!

Questions? Comments? Concerns?

- Any questions on **Memory** or **Advanced Data Structures?**