OPTIMIZING R CODE

Jacalyn Huband

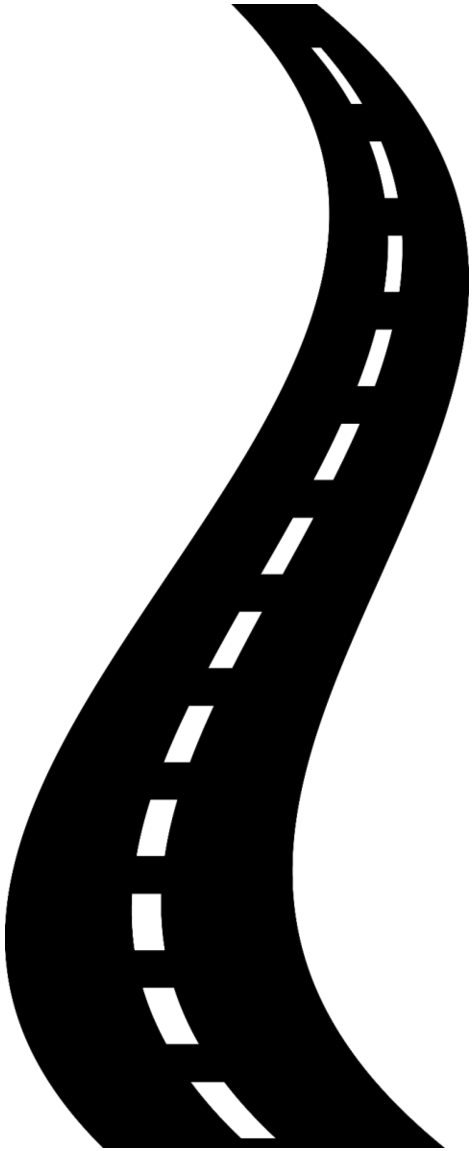
UVA Research Computing

- To get a copy of today's slides and examples:
  - You can clone it from https://github.com/jhuband/Optimizing_R_Workshop.git
  - Or, you can log into Rivanna, open a terminal window, and type:

    ```
    get_Opt
    ```

- Either of these methods will create a folder call Optimizing_R_Workshop.

Roadmap for this Workshop



- Talk about code optimization
- Look at general techniques for optimization
- Show how to find bottlenecks in your code with profiling

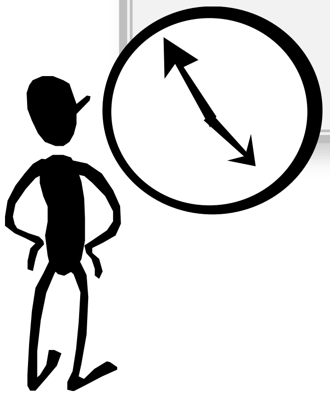
OVERVIEW OF CODE OPTIMIZATION

Your Software Project

It’s About Time

- After getting your code to work, you may find that the code does not *scale* for larger data sets.

> - Suppose it takes only 40 sec for a simple data set.
> - But, when you run it on a data set that is twice the size of the first, the time takes 400 secs!
>
> $$400\ secs = \ \ 6.7\ mins$$

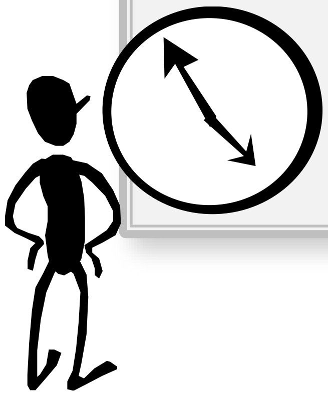


Research Computing

It’s About Time

- Or, you may find that repeating the algorithm multiple times takes too long.

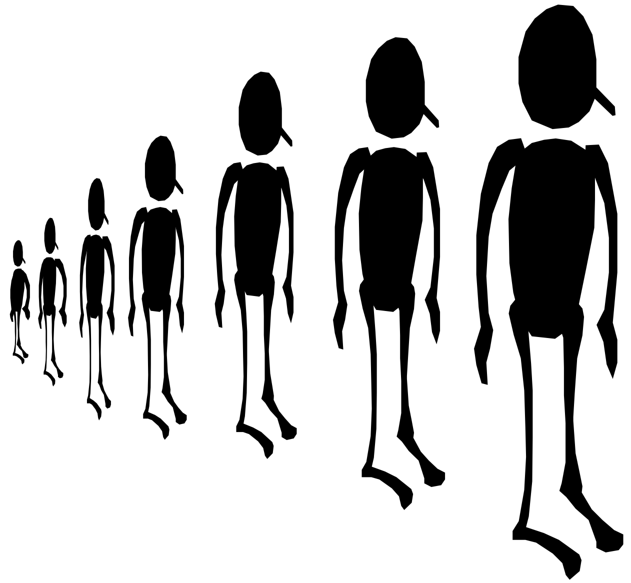
  - Suppose it takes only 40 sec for one iteration (or for one data set).
  - But, you need to run it for 5000 iterations (or on 5000 data sets).



$$40 \frac{sec}{iteration} * 5000\ iter. = 200{,}000\ sec \ \text{ or } \ 55.6\ hours$$

Should you parallelize it?

- Before requesting time on the cluster and parallelizing your code, you may want to **optimize** it.



What is Code Optimization?

- The process of making code more efficient (either with time or memory)
- Important caveats:

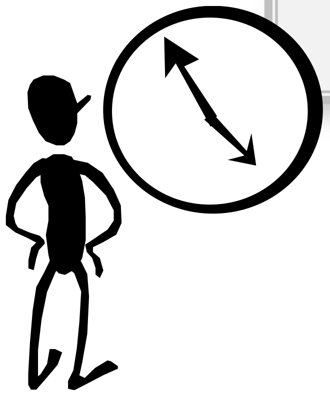
> - The correctness of the code must be preserved.
> - The code should run faster “on average”.
> - There is a tradeoff between your efforts and the computer’s efforts. (**Should you spend a week trying to make a program faster by 1 sec?**)

Going Back to Time

- Returning to our timing example with multiple iterations, suppose you were able to reduce the time of one iteration to 36 seconds.

$$36\,\frac{sec}{iteration} * 5000\ iter. = 180{,}000\ sec \ \text{ or } \ 50.0\ hours$$

You just saved 5.6 hours!



How can we reduce the time for R code to run?

General Advice

1. Do not re-compute values that are used frequently.
2. Avoid using loops.
3. Avoid referencing a single value of an array (e.g., A[i] or M[i,j]) multiple times.
4. Pre-allocate memory whenever possible.
5. I/O is slow – try to consolidate it.
6. Avoid overuse of parentheses.
7. Avoid ifelse for vectors.
8. Use tidyverse to manipulate data

General Advice #1

- **Do not re-compute values that are used frequently.**

  Compute once and store in a variable. This is especially critical in loops. Re-computing a single value thousands or millions of times will increase the time it takes for your code to run.

Before:

```
variance = 0


 for (x in values) {
   distToMean = x - mean(values)
   variance = variance +
           (distToMean^2)/length(values)
 }
```

After:

```
variance = 0
meanOfValues = mean(values)
N = length(values)

for (x in values){
   distToMean = x - meanOfValues
   variance = variance + (distToMean^2)/N
}
```

General Advice #2

- **Avoid using loops.**

  Take advantage of vectorizations.

Before:

```
variance = 0
meanOfValues = mean(values)
N = length(values)

for (x in values){
  diffMean = x - meanOfValues
  variance = variance + (diffMean^2)/N
}
```

After:

```
diffMean = values - mean(values)
variance = sum((diffMean)^2 /length(values))
```

General Advice #3

- **Avoid referencing a single value of an array (e.g., A[i] or M[i,j]) multiple times.**

  If you know that you will be using the value frequently in a block of code, save it to a variable.

Before:

```
y = 0


 for (x in values) {
    y = y + ( sin(M[i, j] * x) * exp(M[i, j]) )/M[i, j]
 }
```

After:

```
y = 0
 mij = M[i, j]

for (x in values) {
    y = y + ( sin(mij * x) * exp(mij) )/mij
 }
```

General Advice #4

- **Pre-allocate memory whenever possible.**

  If you know the size and type of the list or array, reserve memory for the entire list/array before doing calculations.

Before:

```
newValues = c( )


for (x in values){
   newValues = c(newValues, sqrt(x))

}
```

After:

```
N <- length(values)
 newValues = rep(0.0, N)

 for (i in seq(1, N)){
   newValues[i] = sqrt(values[i])

 }
```

General Advice #5

- **I/O is slow – try to consolidate it.**

  When writing results to a file, don’t write each result separately. Instead, create a list of output and write the whole list with one command.

Before:

```
filename = "outputTest1.txt"
outFile = file(filename, "w")

for (x in values){
  line = paste("The square root of",
                    x, "is", sqrt(x))
  write(line, outFile)
}
close(outFile)
```

After:

```
N <- length(values);  lines = rep(" ", N)
for (i in seq(1, N){
  x <- values[i]
  lines[i] = paste("The square root of“,
                   x , "is", sqrt(x))
}
filename = "outputTest2.txt"
outFile = file(filename, "w")
write(lines, outFile)
close(outFile)
```

General Advice #6

- **Avoid overuse of parentheses.**

  The contents inside parentheses are evaluated and stored in a special list structure. There is overhead for allocating the list, storing, and retrieving the results.

Before:

```
y = ( 1 + (values) ) / ( (1) + (values)^2 )
```

After:

```
y = ( 1 + values ) / ( 1 + values^2 )
```

General Advice #7

- **Avoid ifelse for vectors.**

  Although it is expected to be more concise, the ifelse statement can be less efficient than a more detailed approach.

Before:

```
y <- ifelse(values > 4.0, 1, -1)
```

After:

```
x <- runif(numValues, min=1, max=20)

y <- rep(-1, length(x))
y[x > 10] <- 1
```

General Advice #8

- **Use *tidyverse* to manipulate data.**

  tidyverse is a collection of packages designed specifically for ease of manipulating data. Designed by Hadley Wickham, it has a different approach to its syntax.

Before:

```
myData <- read.csv(filename)
quality <- sort(unique(myData$quality))

N <- length(quality) ;   avg_chl <- rep(0.0, N)
for (i in 1:N) {
  qual <- quality[i]
  ndx <- which(myData$quality == qual)
  avg_chl[i] <- mean(myData$chlorides[ndx])
}
df <- data.frame(quality, avg_chl)
```

After:

```
library(tidyverse)
 myData <- read.csv(filename)
 myData %>% select(quality) %>%
                 group_by(quality) %>%
                        summarize(avg_chl = mean(chlorides))
```

HANDS-ON ACTIVITY #1

Can you improve the code?

- Suppose I want to count how many rows in a data frame have a NA entry.   How would you optimize the code below?

```
data <- read.csv("Data/test_data_1.csv")
print(head(data))

numRows <- dim(data)[1]
count <- 0
for (i in 1:numRows){
   rowData <- data[i, ]
   if (any(is.na(rowData))){
     count = count + 1
   }
}
print(count)
```

This code is available in
Optimizing_R_Workshop/activity_1_Opt.R

If using Rstudio, make sure that you click on
Session >Set Working Directory >To Source File Location

Hint: `which(is.na, arr.ind=TRUE)` will give you the row and column for each NA entry.

Question: How can you test that your changes are better?

Time the Code Snippet

- Enclose your code with the timing commands and compare.

```
data <- read.csv("Data/test_data_1.csv")
print(head(data))
startTime = proc.time()
numRows <- dim(data)[1]
count <- 0
for (i in 1:numRows){
   rowData <- data[i, ]
   if (any(is.na(rowData))){
     count = count + 1
   }
}
print(count)
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

This code is available in Optimizing_R_Workshop/activity_2_Opt.R

How does your code compare with the original code?

Run it a couple more times. Are you getting the exact same times when running the same code?

Test with replicate

- To get a better timing estimate, we want to run the code several times and average the results.

```
data <- read.csv("Data/test_data_1.csv")
print(head(data))
numReps = 1000
elapsedTime = system.time(
               replicate(numReps,
                         count_rows_1(data))
             )
print(elapsedTime/numReps)

elapsedTime = system.time(
               replicate(numReps,
                         count_rows_2(data))
             )
print(elapsedTime/numReps)
```

This code is available in Optimizing_R_Workshop/activity_3_Opt.R

How does your code compare with the replicate?

Is it consistently faster?

Activity

- Also in the Optimizing_R folder, there are two scripts that add graphics to visualize the timing differences using some of the General Advice for optimizing R.

1. Open up one (or both) examples and make sure that you can run them on your system. Notice that the size of the input for the functions keeps increasing.

2. Choose another one of the General Advice options and test if the optimized version is indeed faster.

PROFILING

Profiling

- Profiling is a way of determining where your complete program is using the most resources.
  - Profiling is most useful when your code is modularized (i.e., the code has been broken down into tasks and each task is performed in a function).

  - It identifies where the code is spending the most time.

Rprof Function

- The Rprof function is available in R and should work on most platforms.
  - Simply include a call to Rprof at the beginning and end of your program.
  - At the beginning, provide a filename for the data to be stored.
  - At the end, provide NULL – this tells Rprof that it can stop the analysis.

```
Rprof("profile.out")

#Entire program in here
         .
         .
         .

Rprof(NULL)
```

Rprof Results

- To get the results in a human-readable format, use the summaryRprof function, along with the filename where the data were stored, in R or Rstudio.
  - The results will be displayed in a table format.

```
summaryRprof("profile.out")
```

Example Summary

```
> summaryRprof("profile.out")
$by.self
             self.time self.pct total.time total.pct
"computeE"        4.86    75.23       6.40     99.07
"-"               0.54     8.36       0.54      8.36
"<"               0.32     4.95       0.32      4.95
"^"               0.22     3.41       0.22      3.41
"+"               0.20     3.10       0.20      3.10
"<="              0.10     1.55       0.10      1.55
"("               0.08     1.24       0.08      1.24
"sqrt"            0.08     1.24       0.08      1.24
"unlist"          0.02     0.31       0.04      0.62
"rbind"           0.02     0.31       0.02      0.31
"strsplit"        0.02     0.31       0.02      0.31
```

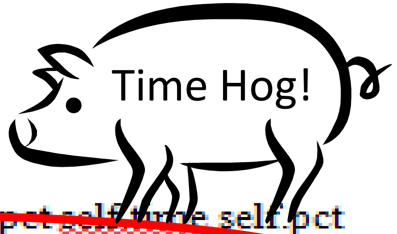
```
$by.total
             total.time total.pct self.time self.pct
"computeE"         6.40     99.07      4.86    75.23
"-"                0.54      8.36      0.54     8.36
"<"                0.32      4.95      0.32     4.95
"^"                0.22      3.41      0.22     3.41
"+"                0.20      3.10      0.20     3.10
"<="               0.10      1.55      0.10     1.55
"("                0.08      1.24      0.08     1.24
"sqrt"             0.08      1.24      0.08     1.24
"readData"         0.06      0.93      0.00     0.00
"unlist"           0.04      0.62      0.02     0.31
"rbind"            0.02      0.31      0.02     0.31
"strsplit"         0.02      0.31      0.02     0.31

$sample.interval
[1] 0.02

$sampling.time
[1] 6.46
```

Example Summary

```
> summaryRprof("profile.out")
$by.self
           self.time self.pct total.time total.pct
"computeE"      4.86    75.23       6.40     99.07
"-"             0.54     8.36       0.54      8.36
"<"             0.32     4.95       0.32      4.95
"^"             0.22     3.41       0.22      3.41
"+"             0.20     3.10       0.20      3.10
"<="            0.10     1.55       0.10      1.55
"("             0.08     1.24       0.08      1.24
"sqrt"          0.08     1.24       0.08      1.24
"unlist"        0.02     0.31       0.04      0.62
"rbind"         0.02     0.31       0.02      0.31
"strsplit"      0.02     0.31       0.02      0.31
```




```
$by.total
           total.time total.pct self.time self.pct
"computeE"       6.40     99.07      4.86    75.23
"-"              0.54      8.36      0.54     8.36
"<"              0.32      4.95      0.32     4.95
"^"              0.22      3.41      0.22     3.41
"+"              0.20      3.10      0.20     3.10
"<="             0.10      1.55      0.10     1.55
"("              0.08      1.24      0.08     1.24
"sqrt"           0.08      1.24      0.08     1.24
"readData"       0.06      0.93      0.00     0.00
"unlist"         0.04      0.62      0.02     0.31
"rbind"          0.02      0.31      0.02     0.31
"strsplit"       0.02      0.31      0.02     0.31

$sample.interval
[1] 0.02

$sampling.time
[1] 6.46
```

Result of Profiling

- Once you know where your code is spending most of its time, you can focus on the portions that need optimization.
  - In the previous example, it was spending most of its time in a function called computeE, where it spent 6.40 seconds out of the total 6.46 seconds.
  - Note: We can only optimize what we have written. We cannot optimize the rbind function, but we can try to reduce the number of calls to it!

FINAL COMMENTS

Optimization Strategy

- During optimization, your goal is to minimize the amount of work the computer is required to do. The strategy we recommend is a two-step approach:

  1. Write code that is efficient from the start (e.g., use vectors instead of loops)

  2. After your code is debugged and working, try more aggressive optimization techniques (e.g., manipulating the mathematical formulas to reduce calls to built-in math functions).

Wallclock Time is Important!

- The only metric that ultimately matters is the Wallclock time.
  - Wallclock is how long you have to wait for your program to run.
  - Wallclock is (sometimes) how much you get charged for.
  - Wallclock is how long your code is blocking other users from using the machine.

Disadvantages?

- **Optimizing code is time consuming**

  Do not waste weeks optimizing code that will run once for 1 hour.

- **Some optimizations can make the code harder to read and debug.**

  Which is more readable:

  y = a3*x^3 + a2* x^2 + a1*x + a0

  or

  y = a0 + x*(a1 + x*(a2 + x*a3))

- **Be aware that different architectures can respond in different ways.**

  Just because code is optimized on your laptop does not necessarily mean that it is optimized on your colleague's computer.

- **Some optimizations can adversely affect parallel scaling.**

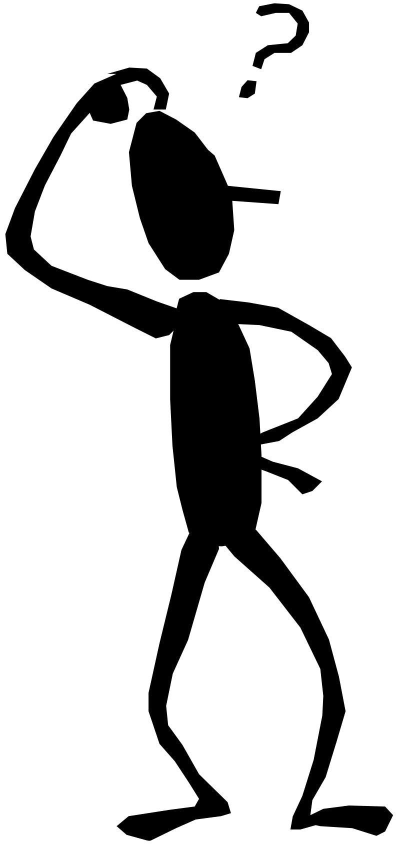
When to optimize?

- Code optimization is an iterative process requiring time, energy and thought. It is recommended for:
  - Codes that will be widely distributed and used often by the research community.
  - Projects that have limited allocation, so that you can maximize the available time on the compute resources.

When optimization isn't enough

- When you have done everything possible to optimize your code, and it still isn't fast enough, you can
  - Find a better algorithm (if one exists).
  - Look into parallelizing your code.

Questions?



TIMING

Quantifying

- To determine if a change in your code improves the efficiency, you need a way to time your code.
    - You can use the time command in Unix; or
    - You can use built-in timing functions in R to measure the time for specific blocks of code.

We'll look at the timing functions in R.

Measuring Time

- proc.time( ) can be used to capture the times before and after several lines of code. The elapsed time can be computed and printed. The print function will display three times (generally in millisecs):
  - Total user time – the time to execute the user instructions (code);
  - Total system time – the time for CPU tasks; and
  - Real elapsed time – how long you had to wait overall.

```
maxValue = 1000000
startTime = proc.time()

for (i in 1:maxValue){
  tmp = 2*i
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

user	system	elapsed
0.42	0.02	0.44

Comparing two codes

- Suppose I want to compare two blocks of code to see which is faster.

```
startTime = proc.time()
for (i in 1:maxValue){
  tmp = 2*i
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

user	system	elapsed
0.42	0.02	0.44

```
startTime = proc.time()
for (i in 1:maxValue){
  tmp = i+i
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

user	system	elapsed
0.41	0.00	0.41

How accurate is the time?

- What happens if we run them again?

```
startTime = proc.time()
for (i in 1:maxValue){
  tmp = 2*i
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

First run

user	system	elapsed
0.42	0.02	0.44

Second run

user	system	elapsed
0.36	0.02	0.37

```
startTime = proc.time()
for (i in 1:maxValue){
  tmp = i+i
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

First run

user	system	elapsed
0.41	0.00	0.41

Second run

user	system	elapsed
0.40	0.00	0.42

What happened to the time?

- Depending on what is running in the background (e.g., garbage collection), each run could have a slightly different time.
  - To get a better estimate, we would want to run the time test several times and average the results.

```
maxValue=1000000
numLoops = 100
startTime = proc.time()
for (j in loopCounts){
    for (i in 1:maxValue){
      tmp = 2*i
    }
}
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime/numLoops)
```

user	system	elapsed
0.3604	0.0017	0.3641

HANDS-ON ACTIVITY #2

Time your optimizations

- Enclose your code with the timing commands and compare.

```
data <- read.csv("Data/test_data_1.csv")
print(head(data))

startTime = proc.time()
numRows <- dim(data)[1]
count <- 0
for (i in 1:numRows){
   rowData <- data[i, ]
   if (any(is.na(rowData))){
     count = count + 1
   }
}
print(count)
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime)
```

This code is available in Optimizing_R/activity_2_Opt.R

How does your code compare with the original code?

Is it consistently faster?

Try again with test_data_2.csv.

Using replicate()

- An alternative to the outer loop is the replicate( ) function..
  - You can specify how many times you want to replicate the code to be tested.

```
numReps = 100
startTime = proc.time()
results <- replicate(numReps,

          for (i in 1:maxValue){
            tmp = 2*i
          }
)
stopTime = proc.time()
elapsedTime = (stopTime - startTime)
print(elapsedTime/numReps)
```

user	system	elapsed
0.3274	0.0016	0.3295

Another timing function

- The function ***system.time( )*** handles the capture of times and computes the elapsed time for you.
  - The statements to be tested must be passed to system.time. This includes the replicate function, too.

```
numReps = 100
elapsedTime = system.time(
   replicate(numReps,

          for (i in 1:maxValue){
            tmp = 2*i
          }
   )
)
print(elapsedTime/numReps)
```

user	system	elapsed
0.3262	0.0007	0.3273

Timing a function

- Of course, the code will be cleaner if the tasks being timed are placed in a function.
  - The statements to be tested must be passed to system.time. It is cleaner to put the statements in a function and pass the function call to system.time.

```
double_by_add = function(x){
  tmp = x + x
}
numReps = 100
maxValue = 1000000
values = 1:maxValue
elapsedTime = system.time(
                replicate(numReps,
                          double_by_add(values))
              )
print(elapsedTime/numReps)
```

user	system	elapsed
0.3262	0.0007	0.3273

HANDS-ON ACTIVITY #3

UNDERSTANDING R

Why is R so slow?

- There are three design issues that make R a slow language (which we cannot change).

- **Extreme dynamism** – Because the contents of a variable can change dramatically during runtime, the interpreter cannot optimize the code.

- **Name lookup with mutable environments** – R spends a lot of time looking up the contents of a variable due to lexical scoping.

- **Lazy evaluation of function arguments** – R stores arguments as a special “promise” object until it is ready to use them. Creation of these objects for each argument could slow it down.

Extreme Dynamism Example

```
x <- 0L

for (i in 1:1e6) {
   x <- x + 1
}
```

Because the contents of x could change at any time through the loop, the interpreter must decide (every time through) what x is and whether it can add 1 to it.

Mutable Environments

- R allows a function to access the memory of a parent routine.

```
a <- 1

f <- function() {
  g <- function(){
    print(a)
    assign("a", 2, envir=parent.frame())
    print(a)
    a <- 3
    print(a)
  }
  g()
  print(a)
}
f()
print(a)
```

Exercise: What is printed?

```
a <- 1

f <- function() {
  g <- function(){
      print(a)
      assign("a", 2, envir=parent.frame())
      print(a)
      a <- 3
      print(a)
    }
  g()
  print(a)
}

f()
print(a)
```

Answer to "What is printed?"

```
a <- 1

f <- function() {
  g <- function(){
      print(a)
      assign("a", 2, envir=parent.frame())
      print(a)
      a <- 3
      print(a)
   }
  g()
  print(a)
}

f()
print(a)
```

Result:

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

Lazy Evaluation of Arguments

- In lazy evaluation, an argument is not evaluated until it is used.
- So, what would be the output of the following crazy code?

```
f <- function(x = ls()) {
  randomVal = runif(1)
  if (randomVal > 0.5){
    y <- randomVal - 1.0
  }
  return (x)
}

#Use default argument
print(f())

#Pass an argument
print(f(ls()))
```

Answer to “crazy code”

- In lazy evaluation, an argument is not evaluated until it is used.
- So, what would be the output of the following crazy code?

```
f <- function(x = ls()) {
  randomVal = runif(1)
  if (randomVal > 0.5){
    y <- randomVal - 1.0
  }
  return (x)
}

#Use default argument
print(f())

#Pass an argument
print(f(ls()))
```

Result:

```
[1] "randomVal" "x" "y"
[1] "f"
         OR

[1] "randomVal" "x"
[1] "f"
```