Stat 243: Problem Set 4, Due Friday Oct-21

October 06, 2016

Instructions

Please turn in (1) a copy on paper, as this makes it easier for us to handle AND (2) an electronic copy through bCourses so we can run your code if needed.

Problem: Function Arguments

Refer to the section about **Function Arguments** in Chapter Functions (from Advanced R by Hadley Wickham):

http://adv-r.had.co.nz/Functions.html#function-arguments

1. What does this function return? Why? Which principle does it illustrate?

```
f1 <- function(x = {y <- 1; 2}, y = 0) {
  x + y
}

f1()
```

2. What does this function return? Why? Which principle does it illustrate?

```
f2 <- function(x = z) {
  z <- 100
  x
}

f2()
```

Problem: Infix Function

R has the function `%in%` which is a binary operator for *value matching*. This function returns a logical vector indicating if there is a match or not for its left operand:

```
a <- c(1, 2, 3, 4, 5)

# is 1 in vector a?
1 %in% a
```

```
## [1] TRUE
```

```r
# is 6 in vector a?
6 %in% a
```

```
## [1] FALSE
```

Write a complementary function to `%in%` using an infix function `"%nin%"` (*not in*), such that it returns a logical vector indicating if there is not a match for its left operand:

You should be able to call it like this:

```r
a <- c(1, 2, 3, 4, 5)

1 %nin% a          # FALSE
6 %nin% a          # TRUE
c(6, 7) %nin% a    # TRUE TRUE
```

Problem: Function `random_sum()`

Consider the following code—saved in an `R` script file `random.R`:

```r
# clear the workspace
rm(list = ls())

random_sum <- function(n) {
  # sum of n random numbers
  x[1:n] <- ceiling(10 * runif(n))
  cat("x:", x[1:n], "\n")
  return(sum(x))
}

x <- rep(100, 10)

show(random_sum(10))
show(random_sum(5))
```

Assuming that you have the file `random.R` in your working directory, you can load the code in your R session with `source()`:

```r
source("random.R")
```

When I load it in my computer I get the following results:

```
x: 6 2 10 1 9 7 9 5 5 6
[1] 60
x: 9 6 9 6 5
[1] 535
```

Explain what is going wrong and how you would fix it.

Problem: Scoping

This problem will have you thinking about scoping in R. Consider the following code:

```
# initialize an empty list
myFuns <- vector(mode = "list", length = 3)

for (i in 1:length(myFuns)) {
  myFuns[[i]] <- function() {
    return(i)
  }
}

# First evaluation
for (j in 1:length(myFuns)) {
  print(myFuns[[j]]())
}
```

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

```
# Second evaluation
for (i in 1:length(myFuns)) {
  print(myFuns[[i]]())
}
```

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

a. Explain what is the result of the first evaluation `for` loop (with `"j"`). Why is the result 3 every time?
b. Explain the result of the second `for` loop. In particular, where is the value of `"i"` in the three `MyFuns` functions being found?
c. Now consider the following code where the three functions are generated within another function. Why do both loops now give the same result and where is `"i"` being found?

```
funGenerator <- function(len) {
  f <- vector(mode = "list", length = len)
  for (i in seq_len(len)) {
    f[[i]] <- function() {
      i
    }
  }
  return(f)
```

```
}


myFuns <- funGenerator(3)

# Third evaluation
for (j in 1:length(myFuns)) {
  print(myFuns[[j]]())
}
```

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

```
# Fourth evaluation
for (i in 1:length(myFuns)) {
  print(myFuns[[i]]())
}
```

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

Problem: Transforming Data with For Loops

The data set for this problem has to do with weekly gasoline prices in California (source: *U.S. Energy Information Administration*):

https://www.eia.gov/dnav/pet/hist/LeafHandler.ashx?n=PET&s=EMM_EPMR_PTE_SCA_DPG&f=W.

	Week 1		Week 2		Week 3		Week 4		Week 5	
Year-Month	End Date	Value	End Date	Value	End Date	Value	End Date	Value	End Date	Value
2015-Jan	01/05	2.671	01/12	2.594	01/19	2.484	01/26	2.440		
2015-Feb	02/02	2.441	02/09	2.627	02/16	2.798	02/23	2.959		
2015-Mar	03/02	3.418	03/09	3.439	03/16	3.356	03/23	3.267	03/30	3.209
2015-Apr	04/06	3.147	04/13	3.102	04/20	3.158	04/27	3.433		
2015-May	05/04	3.711	05/11	3.732	05/18	3.807	05/25	3.757		
2015-Jun	06/01	3.693	06/08	3.591	06/15	3.511	06/22	3.480	06/29	3.450
2015-Jul	07/06	3.432	07/13	3.880	07/20	3.897	07/27	3.812		
2015-Aug	08/03	3.724	08/10	3.565	08/17	3.584	08/24	3.483	08/31	3.342
2015-Sep	09/07	3.266	09/14	3.155	09/21	3.072	09/28	2.994		
2015-Oct	10/05	2.949	10/12	2.914	10/19	2.861	10/26	2.847		
2015-Nov	11/02	2.817	11/09	2.824	11/16	2.780	11/23	2.716	11/30	2.691
2015-Dec	12/07	2.679	12/14	2.654	12/21	2.736	12/28	2.825		

Figure 1: Weekly CA Gasoline prices 2015

The image above is a screen-capture showing the data set as it appears in the EIA website: weekly California retail gasoline prices from January till December 2015 (source: *U.S. Energy Information Administration*)

I've scrapped the data from 2015 and saved it in a csv file available in the github repository:

https://github.com/ucb-stat243/stat243-fall-2016/raw/master/data/raw-gas-prices-2015.csv

The data table in `raw-gas-prices-2015.csv` has 11 columns:

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11
2015-Jan	01/05	2.67	01/12	2.59	01/19	2.48	01/26	2.44		
2015-Feb	02/02	2.44	02/09	2.63	02/16	2.80	02/23	2.96		
2015-Mar	03/02	3.42	03/09	3.44	03/16	3.36	03/23	3.27	03/30	3.21
2015-Apr	04/06	3.15	04/13	3.10	04/20	3.16	04/27	3.43		
2015-May	05/04	3.71	05/11	3.73	05/18	3.81	05/25	3.76		
2015-Jun	06/01	3.69	06/08	3.59	06/15	3.51	06/22	3.48	06/29	3.45

Table 1: First six rows of weekly gas prices

- `V1` corresponds to the month name
- `V2`, `V4`, ..., `V10` contain the starting day of the week (some months have 4 weeks, and others have 5 weeks)
- `V3`, `V5`, ..., `V11` contain the weekly gas prices

Transforming the raw data

The goal of this problem is to "reshape" the raw data set and create a new table `clean-gas-prices-2015.csv` with a simpler structure having the following form:

week	date	price
1	01/05	2.67
2	01/12	2.59
3	01/19	2.48
4	01/26	2.44
5	02/02	2.44

Table 2: First five rows of weekly gas prices

- `week` has the number of weeks (52 in total)
- `date` corresponds to the starting dates of the week
- `price` corresponds to the price for the associated date

Basic For Loops: Write code using one or more `for` loops to extract the end dates that will produce the vector `date`, and to extract the price values that will produce the vector `price`. When writing this code, do not worry about speed, memory management or efficiency. I just want you to write code that gets the job done.

Profiling your Loops: Examine the loops that you just used to get the end dates and values. Use the functions `Rprof()` and `summaryRprof()` to inspect which operations are consuming most resources. Here are some resources about profiling your code:

- https://tgmstat.wordpress.com/2013/09/25/profiling-r-code/
- http://www.stat.berkeley.edu/~nolan/stat133/Fall05/lectures/profilingEx.html

Better loops: Knowing where time is spent (in which functions and calls), try to rewrite the loops to make them more efficient. Maybe you need to initialize vectors allocating enough memory; or you may need to use vectorized code.

Use the function `proc.time()` or `system.time()` to time your initial and second implementations.

Extra Credit

No For Loops: Try to write code avoiding using R loops to get the vectors `date` and `price`.

To get extra credit, your code should be faster than the time spent using one or more `for` loops.