STAT S400: Statistical Computing with R

Assignment #3 – An Introduction to Data Structures

Here you will learn about the main data structures you are likely to encounter, and use, when performing data analyses in R. Also, some more tips on how to generate samples to play with are given. As with the previous assignment, make good use of the script file for these notes, it is named `Assignment3.R`.

Relevant Resources for this Assignment

The R Book: Chapters 2, 3, 4 and 6.

An Introduction to R: Chapters 3, 4, and 6.

R Language Definition: Chapter 3.

Your favorite elementary statistics text: Relevant topics/chapters.

Preliminaries

As indicated in the previous assignment, there are a variety of structures (more commonly called *objects*) in which data can be stored in R. The the *classes* of such objects that you will be introduced to in this course will mostly include *vectors*, *factors*, *lists*, and *data frames*. Another point to note is that up to now the only way data have been imported into R (and stored in these data structures) has been through coding. Additional methods by which this can be done will be introduced along the way in this and later assignments.

Vectors

In R a vector is the most common, and basic, data structure that will be encountered. Vectors can contain a single data value, or any number of data values. You have already been introduced to vectors, for example,

```
> (numbers <- seq(from = 1, to = 10, by = 2.5))
[1] 1.0 3.5 6.0 8.5
```

creates and outputs the *numeric* vector `numbers` – enclosing the assignment statement within parentheses instructs R to output the "computation." Similarly,

```
> (words <- rep(x = c("male", "female"), times = c(3, 2)))
[1] "male" "male" "male" "female" "female"
```

creates and outputs the *character* vector `words`, and

```
> set.seed(seed = 4)
> (boolean <- sample(x = c(TRUE, FALSE), size = 5, replace = TRUE))
[1] FALSE TRUE TRUE TRUE FALSE
```

creates and outputs the *logical* vector `boolean`.

The `is.vector` function can be used to test an object to determine if it is a vector. For example, running the code

```
> is.vector(numbers); is.vector(words); is.vector(boolean)
[1] TRUE
[1] TRUE
[1] TRUE
```

reveals that all three objects are vectors. Then, the class (more literally, the *type* of data contained in the vectors) to which these vectors belong can be determined by running

```
> class(numbers); class(words); class(boolean)
[1] "numeric"
[1] "character"
[1] "logical"
```

Finally, to determine what *type* of data the actual contents of the vectors are, run

```
> typeof(numbers); typeof(words); typeof(boolean)
[1] "double"
[1] "character"
[1] "logical"
```

So, the *objects* `numbers`, `words`, and `boolean` are *vectors* belonging to the classes *numeric*, *character*, and *logical*, respectively. The type *double* indicates `numbers` contains *real numbers* (or, decimal approximations at least) as opposed to *integers*.

A quick summary of the *structure* of an object can be obtained using the `str` function. This function is useful in that it identifies the class of the object (note: *numeric*, *character*, and *logical* are vector classes), gives an indication of its length/dimensions, and lists the first few entries. For example,

```
> str(numbers); str(words); str(boolean)
 num [1:4] 1 3.5 6 8.5
 chr [1:5] "male" "male" "male" "female" "female"
 logi [1:5] FALSE TRUE TRUE TRUE FALSE
```

summarizes the information gleaned from using all of the previous functions.

Factors

Typically, character vectors such as the previously created `words` are converted into what is called a *factor* if the contents are to be involved in any form of statistical analysis. For example,

```
> words <- as.factor(words)
```

redefines `words` as a factor. Then,

```
> class(words); typeof(words)
[1] "factor"
[1] "integer"
```

and

```
> levels(words)
[1] "female" "male"
```

provides a list of the labels for the *levels* of the factor `words`. Again, the `str` function can be used,

```
> str(words)
 Factor w/ 2 levels "female","male":  2 2 2 1 1
```

So `words` is a factor with two levels having the indicated labels.

Curiously, the *type* of the contents of `words` is now *integer*. Think of this as representing the arrangement of the levels in `words` – here `"female"` is identified as the first level, and `"male"` the second – run `fix(words)` to get a better feel for how the object `words` is coded. Close the R Editor window for `words` when you are done.

We will come to (loosely) refer to *vectors* and *factors* (such as `numbers`, `words`, and `boolean`) as *variables* or *samples* depending on the context. Recall that the types of variables seen in elementary statistics courses were *numeric* (typically a vector in R), or *categorical* (typically a factor in R). The object `boolean` is an example of a *logical* vector.

Two Useful Bookkeeping Functions

It is a good idea not to let the workspace get too cluttered with (temporary) objects. So, you can list all objects in the workspace using

```
> ls()
[1] "boolean" "numbers" "words"
```

You can remove all objects from the workspace

```
> rm(list = ls(all = TRUE))
```

then

```
> ls()
character(0)
```

indicates the workspace now contains no objects. As will be shown a little later, the `rm` function can also be used to remove individual or collections of specific objects from the workspace.

Lists

Begin by generating two random samples from normal distributions – use the `set.seed` function to ensure the samples used here are duplicated every time this code is run.

```
> set.seed(seed = 5)
> sample1 <- round(rnorm(n = 5, mean = 5, sd = 1.25), 2)
> set.seed(seed = 13)
> sample2 <- round(rnorm(n = 7, mean = 9, sd = 0.79), 2)
```

Now, to place these in a *list*, run

```
> sampleFormat <- list(males = sample1, females = sample2)
```

Then, to determine the class of `sampleFormat`, run

```
> class(sampleFormat)
[1] "list"
```

`and,` to print `sampleFormat` to the console, run

```
> sampleFormat
$males
[1] 3.95 6.73 3.43 5.09 7.14
$females
[1] 9.44 8.78 10.40 9.15 9.90 9.33 9.97
```

Now, list all objects in the workspace

```
> ls()
[1] "sample1" "sample2" "sampleFormat"
```

then remove `sample1` and `sample2` from the workspace

```
> rm(sample1, sample2)
```

Finally, to see what's left, run

```
> ls()
[1] "sampleFormat"
```

The `names` function can be used to list the names of the objects contained in a list

```
> names(sampleFormat)
[1] "males" "females"
```

and the `str` function can again be used to determine the structure.

```
> str(sampleFormat)
List of 2
 $ males :  num [1:5] 3.95 6.73 3.43 5.09 7.14
 $ females:  num [1:7] 9.44 8.78 10.4 9.15 9.9 9.33 9.97
```

Now try to display contents of, for example, `males`.

```
> males
Error:  object 'males' not found
```

So, the vectors `males` and `females` (*inside* `sampleFormat`) are no longer available for direct access.

Accessing the contents of a list

Here are four ways by which the contents of any list can be accessed. We can use the `$` operator, for example,

```
> sampleFormat$males
[1] 3.95 6.73 3.43 5.09 7.14
```

accesses the contents of `males` from within `sampleFormat`. We can use *indices* to do the same,

```
> sampleFormat[1]
$males
[1] 3.95 6.73 3.43 5.09 7.14
```

The `with` function provides another often convenient method to access the contents of a list

```
> with(data = sampleFormat, expr = males)
[1] 3.95 6.73 3.43 5.09 7.14
```

And, finally, the `attach` function provides a means of unrestricted access.

```
> attach(sampleFormat)
> males
[1] 3.95 6.73 3.43 5.09 7.14
```

If an object, such as a list, is "attached," it is important to "detach" it once it is no longer in use. To do this, run

```
> detach(sampleFormat)
```

The choice of which approach to use will typically depend on the tasks to be performed. The determining factor is very often (but not always) identifying which approach results in the simplest, and clearest code.

Another Helpful Bookkeeping Function

The `search` function can be used to list all objects that are attached (to the workspace). Run

```
> search()
[1] ".GlobalEnv"         "package:stats"  "package:graphics"
[4] "package:grDevices"  "package:utils"  "package:datasets"
[7] "package:methods"    "Autoloads"     "package:base"
```

Now run

```
> Attach(sampleFormat)
> search()
 [1] ".GlobalEnv"        "sampleFormat"       "package:stats"
 [4] "package:graphics"  "package:grDevices"  "package:utils"
 [7] "package:datasets"  "package:methods"    "Autoloads"
[10] "package:base"
```

Notice that `sampleFormat` appears in the outputted list. Then,

```
> detach(sampleFormat)
> search()
[1] ".GlobalEnv"         "package:stats"  "package:graphics"
[4] "package:grDevices"  "package:utils"  "package:datasets"
[7] "package:methods"    "Autoloads"     "package:base"
```

brings things back to as before.

You will encounter issues that can occur if an object, such as a list or a data frame (coming up next) is left attached. More on this when, and as, cases arise.

Data Frames

A *data frame* can be thought of as a special kind of list, one in which the included vectors and/or factors have the same length *and* whose entries are bound to each other – data frames are associated with multivariate data.

One can easily convert data in *sample format* from a list to *multivariate format* to place in a data frame. Using the list `sampleFormat` as an example, begin by extracting the contents of `sampleFormat` and placing them in a single vector, first for all the males then the females.

```
> scores <- unlist(x = sampleFormat, use.names = FALSE)
```

Now get the lengths of the vectors in `sampleFormat`

```
> (n <- sapply(X = sampleFormat, FUN = length))
 males females
     5       7
```

and then identify the labels for the levels: `"male"` for sample 1 data and `"female"` for sample 2 data.

```
> categories <- c("male", "female")
```

Next, create a factor corresponding to `scores`

```
> genders <- as.factor(rep(x = categories, times = n))
```

And, finally, store the two in a data frame

```
> bivarFormat <- data.frame(score = scores, gender = genders)
```

As before, we can run checks. For example,

```
> class(bivarFormat)
[1] "data.frame"
```

or

```
> str(bivarFormat)
'data.frame':  12 obs.  of 2 variables:
 $ score :  num 3.95 6.73 3.43 5.09 7.14 9.44 8.78 10.4 9.15 9.9 ...
 $ gender:  Factor w/ 2 levels "female","male":  2 2 2 2 2 1 1 1 1 1 ...
```

The nature of the contents of a data frame can also be examined using the `head` and `tail` functions. For example, to look at the first three rows and last three rows of `bivarFormat`,

```
> head(x = bivarFormat, n = 3)
  score  gender
1  3.95    male
2  6.73    male
3  3.43    male
> tail(x = bivarFormat, n = 3)
    score  gender
10  9.90   female
11  9.33   female
12  9.97   female
```

So the data frame `bivarFormat` contains the sample data from the list `sampleFormat` reorganized in bivariate format.

Now tidy the workspace up a bit. First check the workspace

```
> ls()
[1] "bivarFormat" "categories" "genders" "n" "sampleFormat"
[6] "scores"
```

then remove all objects except `sampleFormat` and `bivarFormat`

```
> rm(categories, genders, scores, n)
```

Accessing the contents of data frames

The four methods used to access the contents of `sampleFormat` can also be used to access the contents of `bivarFormat` in exactly the same manner. However, observe that the dimensions of a data frame differ from those of a list.

So, for example, while the dimensions of `sampleFormat`

```
> dim(sampleFormat)
NULL
```

show nothing, the dimensions of `bivarFormat` are

```
> dim(bivarFormat)
[1] 12 2
```

indicating that `bivarFormat` has 12 rows and 2 columns. This makes the use of indices a little more specialized, at least at the fundamental level.

For example, to output all columns in the 3rd and 9th rows, run

```
> bivarFormat[c(3, 9), ]
  score  gender
3  3.43    male
9  9.15  female
```

To output all columns for `females` run

```
> with(data = bivarFormat, expr = bivarFormat[gender == "female", ])
   score  gender
6   9.44  female
7   8.78  female
8  10.40  female
9   9.15  female
10  9.90  female
11  9.33  female
12  9.97  female
```

and to output all rows of the first column

```
> bivarFormat[ , 1]
 [1] 3.95 6.73 3.43 5.09 7.14 9.44 8.78 10.40 9.15 9.90 9.33 9.97
```

So, when accessing the contents of a data frame by indices it is important to pay attention to the indices for rows *as well* as for the columns.

Saving and Loading R Data Structures

Data structures such as `sampleFormat` and `bivarFormat` (think of these as parts of a *workspace*) can be saved for recall in a later session. For example, run

```
> save(list = c("sampleFormat", "bivarFormat"),
+     file = file.choose())
```

A *Select File* window will pop up in the working directory, for example

```
z:\My Documents\STAT400\Assign3
```

Next, assign a file name (for example, `trial.RData`), and click on the "Open" button. Finally, click on the "Yes" button in response to the resulting pop up window containing the question "Create the file?"

If you check the directory in question you should see the file `trial.RData`. Now remove all objects in the workspace. To recall the data structures contained in `trial.RData`, run

```
> load(file = file.choose())
```

and select the desired directory and file, for example,

```
z:\My Document\STAT400\Assign3\trial.RData
```

It will be found that the two objects `sampleFormat` and `bivarFormat` are back.

Another way to load a workspace (i.e., a file having the extenstion "`.RData`") is to activate the console by clicking on it, then use the R Gui menu sequence **File→Load Workspace...** and navigate to the appropriate folder to access the file of interest.

Examples of using Lists and Data Frames

The list `sampleFormat` and data frame `bivarFormat` can be used to demonstrate how such structures may be used in a couple of elementary hypothesis tests.

Denote the population mean scores for males and females by μ_{male} and μ_{female}, respectively, and suppose that of interest is to test the hypotheses

$$\text{H}_0 : \mu_{\text{male}} = \mu_{\text{female}} \qquad \text{vs.} \qquad \text{H}_1 : \mu_{\text{male}} \neq \mu_{\text{female}}$$

This is an example of a situation where the t-test of the difference between two independent population means is appropriate. The first task is to determine whether there is evidence to conclude that the variances of the two populations differ.

Denote the population variances of male and female scores by by σ^2_{male} and σ^2_{female}, respectively. Then the hypotheses to be tested are

$$\text{H}_0 : \sigma^2_{\text{male}} = \sigma^2_{\text{female}} \qquad \text{vs.} \qquad \text{H}_1 : \sigma^2_{\text{male}} \neq \sigma^2_{\text{female}}$$

The function `var.test` can be used to test these hypotheses as follows. First, using `sampleFormat`,

```
> with(data = sampleFormat,
+     expr = var.test(x = males, y = females, alternative = "two.sided",
+         conf.level = 0.95))
```

The resulting output shows:

```
        F test to compare two variances


data:  males and females
F = 8.8301, num df = 4, denom df = 6, p-value = 0.02183
alternative hypothesis:  true ratio of variances is not equal to 1
95 percent confidence interval:
 1.417999 81.213263
sample estimates:
ratio of variances
          8.830109
```

Alternatively, you can test these hypotheses using `bivarFormat` by running (output not shown)

```
> var.test(formula = score ~ gender, data = bivarFormat,
+     alternative = "two.sided", conf.level = 0.95)
```

Convince yourself that the two output/results are equivalent. Hint: an equivalent way to state the above hypotheses is

$$\mathrm{H}_0 : \sigma^2_{\text{male}} / \sigma^2_{\text{female}} = 1 \qquad \text{vs.} \qquad \mathrm{H}_1 : \sigma^2_{\text{male}} / \sigma^2_{\text{female}} \neq 1.$$

From the above, it is evident that, at $\alpha = 0.05$, there is sufficient evidence to conclude that the variances of the two populations *are not equal* ($F = 8.8301$, $df_{\text{N}} = 4$, $df_{\text{D}} = 6$, and p-value $= 0.02183$).

Now, it may be assumed that $\sigma^2_{\text{male}} \neq \sigma^2_{\text{female}}$, so to test the hypotheses

$$\mathrm{H}_0 : \mu_{\text{male}} = \mu_{\text{female}} \qquad \text{vs.} \qquad \mathrm{H}_1 : \mu_{\text{male}} \neq \mu_{\text{female}}$$

using `sampleFormat`, run

```
> with(data = sampleFormat,
+     expr = t.test(x = males, y = females, alternative = "two.sided",
+         paired = FALSE, var.equal = FALSE, conf.level = 0.95))
```

to get

```
       Welch Two Sample t-test


data:  males and females
t = -5.6301, df = 4.653, p-value = 0.003079
alternative hypothesis:  true difference in means is not equal to 0
95 percent confidence interval:
 -6.306887 -2.291399
sample estimates:
mean of x  mean of y
 5.268000   9.567143
```

Alternatively, using `bivarFormat` run (output not shown)

```
> t.test(formula = score ~ gender, data = bivarFormat, alternative = "two.sided",
+     paired = FALSE, var.equal = FALSE, conf.level = 0.95)
```

Again, convince yourself that the two output/results are equivalent. Hint: an equivalent way to state the above hypotheses is

$$\mathrm{H}_0 : \mu_{\text{female}} = \mu_{\text{male}} \qquad \text{vs.} \qquad \mathrm{H}_1 : \mu_{\text{female}} \neq \mu_{\text{male}}.$$

In both cases the argument assignment `var.equal = FALSE` instructs the function `t.test` (and R) that $\sigma^2_{\text{male}} \neq \sigma^2_{\text{female}}$.

So, at $\alpha = 0.05$, there is sufficient evidence to conclude that the means of the two populations *differ* ($t = -5.6301$, $df = 4.653$, and p-value $= 0.003079$).

Exploratory Exercises

There are three functions, `sapply`, `tapply`, and `by`, that are quite useful for working with lists and data frames. Find out more about these functions and try your hand out on these with `sampleFormat` and `bivarFormat`. See if you can duplicate the following.

First, use the `sapply` function to calculate the means for `males` and `females` from the data in `sampleFormat`. You should get

```
   males   females
5.268000  9.567143
```

Now use the `tapply` function to calculate the mean `score` by `gender` for the data in `bivarFormat` – you will find the `with` function useful here. You should get

```
  female      male
9.567143  5.268000
```

Now try the `by` function to calculate the mean `score` by `gender` for the data in `bivarFormat` – again, you will find the `with` function useful. You should get

```
gender:  female
[1] 9.567143
------------------------------------------------------------
gender:  male
[1] 5.268
```

Explore using these functions to find other statistics, such as maximums, minimums, medians, variances, and standard deviations.

To be Submitted

The Story and the Data

Bubba, an elementary statistics instructor at a large university in the Midwest, is a strong believer of project-based elementary statistics courses. His colleagues are sceptical, and argue that students do not grasp the underlying concepts when they take project-based courses. Moreover, they feel the benefits of teaching a project-based course do not warrant the extra work required from instructors of such courses – as compared to the traditional lecture-homework-test format. While Bubba does not believe he can sway his colleagues (he believes they are fuddy-duddies who are very set in their ways), he decided to perform a formal (statistical) study to convince *himself* that the project-based approach really is the way to go.

Since he regularly teaches two sections of elementary statistics each semester, Bubba decided to teach one traditional course and one project-based course in the Fall of 2012. A total of 175 students were randomly assigned to the two courses, with 92 ending up in the traditional course and 83 in the project-based course. During the semester he paid close attention to making sure he did the best he could (as a teacher) in both courses, exposing students in both courses to the same core concepts. Then, at the end of the semester students from both sections took a common, comprehensive, concepts-focused final exam. His data were organized under the following variables.

score: Score earned on the final – possible scores 0-100%

course: Which course the student was in – possible entries include `proj` (project-based) or `trad` (traditional)

gender: Gender of the student – possible entries include `male` or `female`.

field: Field of the student – possible entries include `natsci` (Natural Sciences), `soccsi` (Social Sciences), `mathsci` (Mathematical Sciences), and `other` (any other field).

The data he obtained are contained in the file `Assignment3.RData`, *first* save this file to your STAT S400 directory, for example, in

```
z:\My Document\STAT400\Assign3\Assignment3.RData
```

then load the data using

```
load(file = file.choose())
```

and select the desired directory and file `Assignment3.RData`. These data are to be used for Problems 1 through 5. Before beginning, explore the contents of the data frame that is loaded using the methods seen earlier in these notes.

The Problem Statements

Make sure you prepare *well-documented*, and *clearly organized* code to perform the indicated tasks *completely* and *correctly*.

1. Denote the population mean scores for the traditional and project-based course by μ_{trad} and μ_{proj}, respectively. Provide Bubba the code that he can use to test the hypotheses

   $$\text{H}_0 : \mu_{\text{proj}} \leq \mu_{\text{trad}} \qquad \text{vs.} \qquad \text{H}_1 : \mu_{\text{proj}} > \mu_{\text{trad}}$$

   at $\alpha = 0.10$. Keep in mind that you will need to test the equality of variances first to determine what to use for the argument `var.equal` in the function `t.test`.

2. Denote the population mean scores for the males and females by μ_{male} and μ_{female}, respectively. Provide Bubba the code that he can use to test the following hypotheses at $\alpha = 0.05$.

   $$\text{H}_0 : \mu_{\text{male}} = \mu_{\text{female}} \qquad \text{vs.} \qquad \text{H}_1 : \mu_{\text{male}} \neq \mu_{\text{female}}$$

   Once again, be sure to test the equality of variances first.

3. Provide code that will output the *contingency table* (frequency table) for `course` by `gender`. Hint: Take a look at the functions `table` and `xtabs`.

4. Provide code that will output contingency tables for `course` *and* `gender` by `field`. Hint: The output you get will include four separate contingency tables, one for each field and each having the same appearance as in the table produced for Problem 4.

5. Provide code that will construct a table that contains mean scores by `course` *and* `gender`. Hint: You might find the `tapply` or `by` functions useful at first. The trick then will be to transfer the results into table form.

Remember to follow the format laid out in the *About the Assignments* document on the course website. Call your assignment file, for example, `Hay-Jahans_Assign3.R`.