SCR week 2: exercises

Exercises part 1

1.1 `error... error`

What do you think it is that produces the warnings in the following lines of code? How would you fix it?

```
mydlist <- vector(mode = 'list', length = 2)
mydlist[1] <- 97:122
```

```
## Warning in mydlist[1] <- 97:122: number of items to replace is not a
## multiple of replacement length
```

```
mydlist[2] <- letters
```

```
## Warning in mydlist[2] <- letters: number of items to replace is not a
## multiple of replacement length
```

1.2 Creating a list

a.

Create a list object with the following entries:

- An entry called `name` with value "my list"
- An entry called `normal_values` with 50 draws from a standard normal distribution
- An entry without a name that contains 50 samples from the numbers `1:10` using `sample`
- An entry called `id` that contains the numbers 1 through 50, but shuffled around using `sample` (try e.g. `sample(1:5)` a number of times.
- An entry called `my_sampler` that is a `function` that takes an integer argument and returns as many random samples from a standard normal distribution as the integer argument.

Answer:

b.

Try to access the `my_sampler` element from your list. Is it possible to run the function?

Answer:

c.

Convert the list to a data.frame, by using `as.data.frame`. Does it work?

Answer:

d.

Use negative indices to remove the element that gives a problem in **c** and feed the reduced list to `as.data.frame`. Does the result contain any strange variables? Do you notice any vector recycling?

Answer:

e.

Use three different ways of accessing list elements, to access `normal_values`, the samples from `1:10`, and `id`, and feed them to the `data.frame` function to create a data.frame (and store it as an object called `my_data_frame`). Add some nice tags for the variables you create.

Answer:

1.3 Writing a bivariate summary function

a.

Create two vectors, each containing a 100 draws from a standard normal distribution.

Answer:

b.

Create a list that contains the following elements:

- A matrix of which its first 10 rows are the first 10 elements of both variables side by side
- A correlation matrix
- A covariance matrix
- A list with two elements:
- A vector with a few univariate statistics for variable 1 (i.e. `min`, `max`, `mean`, `range`)
- A vector with a few univariate statistics for variable 2

Make sure you set nice tags (labels) for all entries.

Take a look at the `cov()` and `cor()` functions to see whether it is more convenient to give these functions two vectors (with the two normal variables) or a matrix containing the two variables (given that you want a covariance *matrix* and a correlation *matrix*).

Answer:

c.

Try to `unlist` your list, with the argument `recursive = FALSE`.

Answer:

d.

Try to `unlist()` your list, with the argument `recursive = TRUE`. Do you notice the difference with the previous question?

Answer:

e.

Now use the code you've written in **b** to write a function that takes as argument two vectors, and produces the summary provided in the list you've created in **b**.

Try out your function use as arguments the two vectors you've created in **a** and see if you get the same result as in **b**.

Answer:

f.

Instead of returning a `list` object, do you think there is another way to all of the information you've computed in your bivariate summary function?

Answer:

Write down your answer here in text...

1.4 Choices, choices...

Often in programming, and thus in `R`, there are multiple ways to do things. The most important thing is that whatever you do: first get as fast as possible towards a correct solution with probably a "very ugly" script of code.

Then, there are other things to consider such as: readability of your code (by yourself and by others), the efficiency of your code (in terms of time and use of computer memory), maintainability of your code, and generic applicability of your code. Perhaps even aesthetics (the Art of R Programming).

Let us take a look at a simple example in which there are choices to be made. We've seen multiple ways to access a list. We can access an element by its name, or by its position in the list.

As Matloff notes, using names (and tags) is more convenient, because if the order of the elements changes, using numbers for positions might no longer be correct. This may happen easily if at some later point, when you decide the list needs another extra component.

For example:

```
my_personal_details <- list(species = "human", age = "75")

# Use age to print:
paste("I am", my_personal_details[[2]], "years old.")
```

```
## [1] "I am 75 years old."
```

```r
# Oh oops, forgot to add that my name is...

my_personal_details <- c(name = "Mr. Miyagi", my_personal_details)

# Let's print my age again:
paste("I am", my_personal_details[[2]], "years old.")
```

```
## [1] "I am human years old."
```

```r
# hmm....
```

a.

Use `as.list()` to create a list object, with the sequence 1 through 260000.

Answer:

b.

Give each entry in this list of 260000 elements a unique name by setting the `names` attribute. Use `paste()` and for example a combination of numbers and letters to automatically generate these unique names. You could e.g. call the first 26 entries `a1`, `b1`, `c1` etc, and the second 26 entries `a2`, `b2`, and so on.

Here's a small snippet of code to get you started:

```r
paste(letters[1:5], rep(c(1, 2), each=5), sep="") # this uses vector recycling!
```

```
##  [1] "a1" "b1" "c1" "d1" "e1" "a2" "b2" "c2" "d2" "e2"
```

Answer:

c.

Access the 130001th element and read its name. Then also access the elements via its name to double-check you get the same entry.

Answer:

d.

Let's introduce two new functions: `replicate()` and `system.time()`. These can be nicely used together to measure the amount of time it takes for the computer to do something. Many operations on the computer are done unmeasureably fast. So to measure how long a particular computation takes, is to repeat it many times, and to look at the total time it takes to do all the replications. This is where `replicate()` comes in.

Two examples of `system.time()` are: `system.time({ 1 + 1 })`, and `system.time(mean(1:10))` (`R` will provided you with three timings, in this course `user` is the most important one to look at). An example of `replicate()` is: `replicate(n = 100, mean(1:10))`, where `n` is the number of times you replicate `mean(1:10)`. Combine what you see from these examples to compare the speed of accessing an element from a list, using the name of the element, and the position. Note: you may have to use many replicates. Which is faster?

Answer:

1.5 Using `lapply()`

`lapply` is an `R` function that allow us to repeatedly 'apply' a function to the elements of a vector. Basically, `lapply` takes the first element from the vector, uses that element as the argument for a function you provide, and returns the result from that function. It then moves on to the next element and repeats the process until it has used all elements in the vector.

a.

Create a vector variable, called `my_vector`, with the numbers 1 through 26, and shuffle them with the function `sample`.

Answer:

b.

Use `lapply` to take the `sqrt` of all the elements of `my_vector` with the following code: `lapply(my_vector, sqrt)`. Look at the result. What does the result look like? How many elements do you have?

Answer:

c.

Use `lapply()` to translate the numbers 1 through 26, to the corresponding `letter` in the alphabet. Write a function to do this: i.e. write a function that takes a number, and returns the corresponding letter, call this function `NumberToLetter`.

Answer:

d.

Suppose you'd want the result of applying your functions to all the elements of the vector to be a vector itself. How could you do this? *Hint: we've already seen this function during class and in some of the previous exercises.*

Answer:

e.

So far, all of the tasks in this exercises could have been done, simply by exploiting `R`'s vectorized functions. We can feed a vector to the `sqrt()` function, and it will return, as a vector, all the square roots of the elements in that vector. We could have used the 26 numbers as index for the `letter` object: `letters[my_vector]`. However, this is not necessarily always possible.

Use the following code to create a new list object: `my_list <- replicate(10, list(rnorm(5)))`. Look at the results to see what the `replicate` function has done.

Answer:

f.

Suppose now that we wish to calculate the mean of the 10 entries. Try feeding `my_list` to the function mean. Does it work?

Answer:

g.

Use `lapply()` to take the mean of each of the entries in `my_list`.

Answer:

Outro

Beyond simple examples, `lapply` can become tricky to use. For example, instead of using the `mean` function we may need to use a function that requires multiple arguments, and if so, how would we need to write this? For now, get comfortable with `lapply()` and the concept of repeatedly applying a function on elements of a vector. We'll get into more detail of the `*apply` family in a later lecture.

Exercises part 2

2.1 Row and column names

a.

Like during the lecture, create a dataset of your own with some variables containing random normal values.

Answer:

b.

Write the file to your computer using `write.table`. Use all the defaults.

Answer:

c.

Look at the contents of the file with Notepad or TextEdit or similar. Do you see *just* the variables that you've created, or does the file contain more information?

Answer:

Write down your answer in text here...

d.

`R` automatically also writes the row and columnnames of the data.frame you write. Look at the helpfile of `write.table` and see how to change the defaults to **not** write rownames.

Answer:

e.

Try to read in this new file, that does not have rownames, using `read.table` and its defaults. Anything wrong with the data.frame you get from reading this file?

Look for example at the class and mode of the columns of the data.frame. If you don't know what `factors` are in `R` yet (we'll cover these next lecture week), also specificy the argument `stringsasFactors = FALSE`.

Answer:

f.

From the `read.table` helpfile:

> header: a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: header is set to TRUE if and only if the first row contains one fewer field than the number of columns.

Fix your code to correctly read in the file without rownames, but with the variable names.

Answer:

Outro

`R`'s defaults in writing and reading data make it convenient if you are exchanging data from and to `R`. However, other programs, when writing files, might not use rownames, or might use rownames, but explicitly give these a name in the header of the file (the first line) such as `row_name`. In both these cases, `R` will mess up reading the file, if left to its defaults. Then, you will need to do some manual tweaking of the arguments `read.*()` functions to get things right.

2.2 Reading and writing some difficult data

a.

Take a look at the file `difficult.txt`. The file contains two variables, with numerical values with decimal numbers, and rownames.

b.

Use `read.table`, `read.csv` and `read.csv2` to read in the data, and store these as three seperate objects. Like in the previous exercise, use `stringsAsFactors = FALSE`.

Answer:

c.

Inspect the three different data sets you've read into `R`. Make sure you inspect the modes of the variables and see if you can, for example, calculate a mean.

d.

If done correctly, you'll notice that none of these three default `read.*` functions correctly reads the file. Look specifically at the `sep` and `dec` arguments of the functions, and the corresponding seperators and decimal indications used in the file to fix the problem and correctly read in the data.

Answer:

Outro

`R` needs precise instructions: sometimes you need to tell it *exactly* what to do. Results from reading in data may often seen confusing, or weirdly incorrect, or surprisingly correct: these usually have to do with `R`'s default choices (= default arguments in the function)! Know that you can simply set and change these as you want, using arguments.

Exercises part 3

3.1 Sorting our data

We've seen how to access the variables of a data.frame, for example using `my_data_frame$id`. We can of course, also access the rows using (for the first row) `my_data_frame[1, ]`. You can also select multiple rows at the same time, e.g. using `my_data_frame[c(1, 2), ]`. This basically returns a data.frame, containing the selected rows, and more specifically in the order that you asked for: `c(1, 2)` asked to see the first row first, then the second row.

a.

Read in "my_data_frame.csv", into an object called `my_data_frame`.

Answer:

b.

Try `my_data_frame[c(3, 2, 1), ]` and compare the entries with first three entries of the original data.frame.

Answer:

c.

If you look at the variable `id`, you'll see its elements are not nicely ordered. We can get `R` to sort this variable, using the function `sort`. Look at the helpfile if needed, and sort the `id` variable.

Answer:

d.

We can also sort `my_data_frame` according to the values of `id`. Look at the helpfile of the function `order`. An example application is:

```
order(c(3, 1, 2))
```

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

The result is: 2 3 1 e.g. the second element should come first, then the third element, then the first element. This gives us the ordering: 1 2 3

Combine `order()` with what you've learned in **b** to create a *new* data.frame, that has its rows order according to `id`.

Answer:

e.

Suppose however that we also had a `group` variable. Add this variable yourself to the original data you read in from `my_data_frame.csv`. Use `rep()` to create a variable containing 5 different subsequent values (1 through 5), and repeated as many times as needed to have as many `group` entries as rows in the dataset (e.g. `1, 2, 3, 4, 5, 1, 2,` etc). Use `cbind()` to combine the dataset and your `group` variable. Could you also come up with another way to add the `group` variable to the data set?

Answer:

e.

Now let's order the dataset according to the group each object belongs to, and within each group, according to the `id`. We can do this my giving multiple arguments to `order` (e.g. `order(var1, var2)`). Try this to order the data as asked. In this case you can easily verify yourself if the ordering is correct!

Answer:

Applying apply

a.

Read in the data file `jumble.txt`. Which `read.*()` function do you think you need to use? Take a look at the contents of the file to decide.

Answer:

b.

Use apply to repeatedly paste together only every *fourth* element of the rows and the columns. That is: use letters number 4, 8, 12, etc. The result should be a vector of pasted together things, with as many elements as rows, or columns (depending on which index you used). Also paste together the elements of this resulting vector. In both paste operations, use the option `collapse = ""`.

Which version of apply made more sense?

Answer: