Lab: Vector Manipulation

36-600

Fall 2022

If you can see this message: congratulations! It means that you have sucessfully installed `R` and `RStudio` on your computer and that you are ready to dive into building those basic skills that will lead you to being able to manipulate data frames, tables of data where each row represents an observation (e.g., a person) and each column represents a measurement (e.g., the height or weight of that person).

Before we start, note that when you insert code into each "code chunk" below (the sections delimited with "'{r} and "'), you can test it in a number of ways:

- at the upper-right corner of a code chunk, there is a green triangle... clicking on that runs all the code in the chunk
- you can highlight code and, at least on a Mac, press Command-Return, and the highlighted code will execute (you can do this with Windows too, but we'll have to collectively remember the stroke sequence... Control-Return?)
- you can click on "Knit," which will run *all code in the file* and output an html file with the results

Question 1

Each column of a data frame is itself a vector. So let's start with some basic vector manipulation. Use the `c()` function to define a vector `x` with four values: 1, 4, 1, and 9. (*You should replace the "# FILL ME IN" statement below with your answer.*) Note that vectors are homogeneous (all of the same type), with the most common types being `double` (or `numeric`), `character`, and `logical`. The vector you define has type `double`. Check this by typing `typeof(x)` and noting the output. Then type `x[4]`. What do you see? (Indicate the answer by, on a new line in the code chunk, typing one number symbol # [which denotes a comment] and then afterwards typing your answer in words [and/or numbers].)

```
x <- c(1,4,1,9)
typeof(x)  # it is "double"
```

```
## [1] "double"
```

```
x[4]       # you see the 4th element of x, the value 9
```

```
## [1] 9
```

If the value(s) inside the square bracket is/are numeric, then that/those elements of the vector are displayed. (Note: `R` counts from 1, not 0.) If the value(s) are logical, then only those elements with value `TRUE` are displayed. This will make more sense below.

Question 2

Now define a vector `y` with four values: 2, 2, 5, 8. Then add `x` and `y`, and multiply `x` and `y`. Note that the following operators are using to carry out basic math in `R`:

Operation	Description
+	addition

Operation	Description
-	subtraction
*	multiplication
/	division
^	exponentiation
%%	modulus (i.e., remainder)
%/%	division with (floored) integer round-off

```
y <- c(2,2,5,8)
x + y
```

```
## [1]  3  6  6 17
```

```
x * y
```

```
## [1]  2  8  5 72
```

What you should observe are vectors with four numbers each. Note that `R` did not require you to loop over the vector indices, i.e., `R` did not make you add `x[1]` and `y[1]` first, then `x[2]` and `y[2]`, etc. `R` made things easy, by utilizing *vectorization*: it takes care of entire vectors at once, without explicit loops needing to be defined by you. Now, define a vector `z` with three values: 1, 2, 3. Add `x` and `z`. Does it work?

```
z <- c(1,2,3)
x + z
```

```
## Warning in x + z: longer object length is not a multiple of shorter object
## length
```

```
## [1]  2  6  4 10
```

```
# OK, this is an R "weirdism": when the vector lengths are not the same, the smaller
# one is extended via "recycling": z -> z' = c(1,2,3,1). Why? Why not just throw
# an exception? That's a question with no known answer, at least to me.
```

Question 3

Now redefine the vector `x` to be of length 500, with all the elements being 0. You don't want to do this using the `c()` function! (Look back at the notes for alternatives.)

```
x <- rep(0,500)
```

Question 4

Coming up with ad hoc vectors of integers is a bit of a pain, and you wouldn't want to type them out if they are long. One way to come up with a random vector is with the `sample()` function. The information that you pass into a function is called an argument, and `R` functions sometimes can have many arguments. Let's look at the help page for `sample()`, which you can bring up by typing `?sample` in the console, or going to the `Help` pane and typing `sample` in the search bar.

```
Usage
```

```
sample(x, size, replace = FALSE, prob = NULL)
```

(What I just typed is an example of a verbatim block: it doesn't execute as `R` code.) What we see is that `sample()` has four arguments. Two of them, `replace` and `prob`, have *default values*...so if you are happy with the defaults, you need not refer to these arguments at all. So you just need to specify, at a minimum, two arguments: `x`, which is either a number or a vector from which to sample data, and `size`, which is the number of data to sample. If you do this

`x <- sample(10,5)`

you are telling `R` to sample five numbers between 1 and 10 (inclusive), with all the numbers being different (because `replace=FALSE`), and to save the numbers as the vector `x`. If you do this

`x <- sample(40:50,5)`

you are telling `R` to sample five different numbers between 40 and 50 (inclusive). And if you do this

`x <- sample(3,10,replace=TRUE)`

you are telling `R` to sample ten numbers between 1 and 3 (inclusive), and repetition is allowed. (We call this "sampling with replacement.") Etc. Now, sample 100 numbers between 1 and 100 (inclusive) with replacement, and save the output as the vector `x`. How many unique integers are there in `x`? Use handy vector functions to get a concise answer: do not print out `x` and count by eye! (If you need help, call a TA or me over, or come to office hours.)

```
x <- sample(100,100,replace=TRUE)
length(unique(x))
```

```
## [1] 67
```

Question 5

What is the smallest integer between 1 and 100 (inclusive) that does *not* appear in your vector `x`? Review the handy vector functions to see if you can come up with a way to do this... but also feel free to seek out help if necessary. Note that there is no one correct answer.

```
# This has no good solution given the notes and what I meant you to really do here?
# Unknown. (Meaning: forgotten.) So take the following with a grain of salt.

1:100 %in% x   # returns TRUE if vector element is in x, FALSE otherwise
```

```
##   [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
##  [13] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
##  [25]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
##  [37]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
##  [49]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
##  [61]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
##  [73]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
##  [85]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
##  [97] FALSE  TRUE  TRUE FALSE
```

```
              # we should find the smallest FALSE
i <- 1:100
g <- i %in% x
min(i[!g])    # !g flips the TRUEs to FALSEs...this gives the smallest number
```

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

```
              # in the vector i (where i is 1,2,3,...,100) that does not appear
              # in x
```

Question 6

Relational operators are binary operators of the form "variable operator value," e.g., `x < 0`. The six basic relational operators are `==`, `!=`, `<`, `>`, `<=`, and `>=` (for "equals," "not equals," "less than," "greater than," "less than or equals," and "greater than or equals.") Relational operators return a vector of logicals, meaning a vector of `TRUE` and `FALSE` values. Below, redefine `x` to be the vector with elements 1, 4, 1, and 9, and then display the output for `x == 1` and `x > 3`.

```
x <- c(1,4,1,9)
x == 1    # TRUE, FALSE, TRUE, FALSE
```

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

```
x > 3     # FALSE, TRUE, FALSE, TRUE
```

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

Question 7

Apply the `sum()` function with input `x == 1`. Does the output make sense to you?

```
sum(x==1) # TRUE = 1, FALSE = 0, so the sum is 2
```

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

Question 8

Relational operators may be combined with `&` (logical AND) or `|` (logical OR). Below, display the output for `x < 2 | x > 5`.

```
x < 2 | x > 5  # displays TRUE, FALSE, TRUE, TRUE
```

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

Question 9

A reason to learn relational operators is that they underpin the manipulation of vectors (and thus underpin the manipulation of, e.g., rows or columns of data frames). To display a subset of values of the vector `x`, you can for instance type `x[...]`, where you would replace `...` with a relational operator. What happens when you type `x[x==1]`?

```
x[x==1] # x == 1 -> TRUE FALSE TRUE FALSE, with the elements associated with TRUE being printed out
```

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

Question 10

Some last things to do for now: apply the `length()` function to `x`, apply the `sort()` function to `x`, apply the `sort()` function to `x` with the additional argument `decreasing=TRUE`, apply the `unique()` function to `x`, and apply the `table()` function to `x`. (You may have done some similar things above when we told you to solve certain problems with handy vector functions.) Build intuition about what each does. (Note that `table()` is a handy function for doing exploratory data analysis of categorical variables.)

```
length(x)  # 4
```

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

```
sort(x)    # 1 1 4 9
```

```
## [1] 1 1 4 9
```

```
sort(x,decreasing=TRUE)  # 9 4 1 1
```

```
## [1] 9 4 1 1
```

```
unique(x)  # 1 4 9
```

```
## [1] 1 4 9
```

```
table(x)   # see output: 1 appears twice, etc.
```

```
## x
## 1 4 9
## 2 1 1
```

Question 11

(Looking ahead.) A *list* in R is a collection of vectors. Define a list below using `list()`, with the first argument being a defined vector with name `x` and values 1 and 2, and the second argument being a defined vector with name `y` and values "a", "b", and "c". (Note: your arguments won't look like `z <- c(TRUE,FALSE)` but more like `"z"=c(TRUE,FALSE)`). Display the list.

```
list(x=c(1,2),y=c("a","b","c"))
```

```
## $x
## [1] 1 2
##
## $y
## [1] "a" "b" "c"
```

The individual entries of a list are vectors, which are homogeneous, but the entries may each be of different type. **A list whose entries are all of the same length is a data frame.**