SC306 Homework 3: R Intro Part 2

Othar Zaldastani II

February 27, 2026

Tutorial 2 - More on Vectors, Data Frames, and Functions David M. Goehring 2004 (creator) Juliet R.C. Pulliam 2008,2009 Steve Bellan 2010, 2012 Jim Scott 2014

```r
#test comment
```

SECTION A. Accessing Vector Elements

By the end of this tutorial you should:

- Be able to retrieve useful subsets of your data
- Understand more about data frames
- Know the methods and uses of logical values in R
- Be able to generate and use factors
- Know how to write your own generic functions

Beyond Numbers: Relational and Logical Operations in R

So far everything you have done in R has involved numbers or vectors of numbers. To properly exploit R's complexity, you need to become familiar with relational and logical operations in R.

Relational operations work just like numerical operations, in terms of how they are processed. Return for a moment to our first calculation from the last tutorial, an addition problem:

3 + 4

The analogous calculation of a single relational operation is something like

5 > 4

"Is 5 greater than 4?" Yes. And R tells you that this is a TRUE statement. Or,

1 + 1 < 1

Makes sense, right?

The greater-than, >, and less-than, <, symbols are straightforward. Similarly, R has greater-than-or-equal-to and less-than-or-equal-to symbols, >= and <=, respectively.

Slightly less intuitive are the relational operators for equality, ==, and inequality, !=. Try

x <- 4

x == 1 + 3

y <- x != 4

y This last example demonstrates that variables can hold logical values. These relational operators also operate on logical values, as in,

```
y == FALSE
```

Logical operations are operations that only make sense when performed on TRUE and FALSE values. These will likely be familiar to you, the central operations being AND (&), OR(|), and NOT(!).

The operators used in R are standard: &, |, and !, respectively. Let's see them in action:

```
!TRUE
```

```
to.be <- FALSE
```

```
to.be | !to.be
```

```
FALSE & (TRUE | FALSE)
```

By combining logical and relational operations, we can make complex inquiries about values.

R has special values for infinity (Inf), not-a-number (NaN), and not-applicable (NA). These generally behave very sensibly - a mathematical operation on not-a-number is obviously not a number as so is returned as NaN. Things are less simple when using logical and relational operators. Consider 4 != NaN. In one respect, the answer perhaps should be TRUE; that is, 4 definitely isn't equal to not-a-number. But, striving for consistency, R returns NA, much as it would for a mathematical operation. Even worse is the situation

```
x <- NaN x == NaN
```

You might think that this is a reasonable test for whether x has a numerical value, but it won't work for the same reason mentioned above. In general, keep this trickiness in mind and remember there is a special function is.na() for determining whether x is a valid number: is.na(x)

It is worth noting that information about these operations can be pulled up at any time by typing help("&") or help(">") or the using help() function with any of the other symbols used in these operations.

Vectors of Logical Values

As a shorthand, TRUE and FALSE can be entered as T and F. This allows for rapid entry of vectors of logical values, for example,

```
logical.vec <- c(T,T,F,T)
```

```
logical.vec
```

Unfortunately, and rather inexplicable, T and F cab be reassigned to any arbitrary values. This will render most code utterly unpredictable. So, never, never, never do this:

```
T <- 4 REALLY BAD,BUT NO ERROR PRODUCED
```

And, if you ever do something like this (though you shouldn't!), make sure you quickly do this:

rm(T) which will set T (or F) back to its default logical value.

Relational or logical operations also act on vectors to produce vectors of logical values, as in,

```
x <- rnorm(10)
```

```
x < 0
```

```
y <- (x > -.5) & (x < .5)
```

```
!y
```

(make sure you understand what's happening above - if you don't - ASK!) This will be especially handy when we look at the concept of indexing, below.

Generating Sequences

There are many occasions in R when you need a patterned sequence of numbers. Most counting can be accomplished by use of the seq() function. If you haven't already done so, it is worth taking a look at the help-file on seq() because it has a few arguments that can make your life easier.

?seq

For example, seq() can generate a vector of a certain length between certain endpoints by typing

x <- seq(0,1,length.out=20)

giving you a vector of length 20 between 0 and 1, confirmable by typing

length(x)

A very common need in R is to generate vectors with an interval of 1 between each element. R has a shorthand for this using the colon notation, as follows,

y <- 5:10

generating a vector that counts from 5 to 10, inclusive. Note that is generally treated first in the order of operations.

Don't underestimate the value of the colon notation. Even for typing a vector of length 2, like (1,2) or (2,1) - using the c function to generate the vector is pretty tedious (e.g., c(1,2)). These vectors can be generated in three quick characters by typing 1:2 or 2:1, respectively. You may want to check out the rep() function, for repeating sequences, which can also save time.

Indexing

R has an incredibly useful way of accessing items from a dataset. Each item in a dataset has it's own index, or numbered location, in the object's structure. Square brackets are used to extract an item or items from a dataset, but it is crucial to understand that there are two completely distinct ways in which brackets are used to access items. I will consider the two methods for accessing a vector of length n in turn below.

The first option: Logical vector of length n Use it for: Finding a subset of data based on a rule

Logical indexing works as if you've asked your indexing vector the question, "Do you want this item?" for each of the items in the vector.

x<-1:5

x[c(T,F,F,F,T)]

If we combine this logical indexing with the relational and logical operators you learned above, we have an exceptionally powerful tool to retrieve data that meet any set of criteria.

y<-rnorm(10000)

hist(y[!((y>-2)&(y<0))])

I will give more insight below when I discuss indexing data frames. Stay tuned.

In any operation in R, vectors will be automatically repeated until they reach the necessary length for the operation to make sense. For example, note the results of

1:6 + 1:2

The same repetition holds for logical vectors.

The second option: Value or vector of any length with values (1 to n) OR (-n to 1) Use it for: Single item retrieval or shuffling, sorting, and repeating

Accessing single items with brackets and a single index should be straightforward

x<-3*0:5

x[4] show the 4th element of x

One tedious way of creating a new vector of values from a vector's elements would be c(x[2],x[3],x[4]) TEDIOUS

So R makes it much easier by allowing a vector of indices to generate a vector. Thereby, the command above becomes

x[2:4]

There is nothing preventing you from accessing any element any number of times.

x[c(2,2,2,5,5,5)]

Additionally, R allows you to use negative indices, indicating which items you want to exclude, as in,

x[c(-1,-6)]

This is fine and productive as long as you remember never to mix negative and positive indices - R will not know what you want it to do:

x[c(-1,4)] BAD

Sorting

Now that you have been introduced to indexing, you may have an inkling of how much more powerful the sorting functions of R can become.

As an introduction, let's say you have a 4-element vector,

my.vector <- 5:8

Using numerical indexing, we can manually re-order this vector by calling each of its indices once in our preferred order, for example

my.vector[c(2,3,4,1)]

or, for a quick reversal

my.vector[4:1]

Now, manually generating the vector of indices is not monumentally useful, which is where the function order() comes in. As demonstration, imagine we have a vector of student names and a corresponding vector of student heights (in meters).

stud.names <- c("Carol", "Walter", "Rachael", "Petunia", "Clark","Justin")

stud.heights <- rnorm(6,1.7,.12)

What we definitely don't want to do is to perform sort() on each of these vectors independently. This will eliminate the pairing of the name to the height. So how can we sort one vector and have the other vector align correctly? Try order() on the names,

order(stud.names)

Note that it returns the indices in the right order (alphbetically), not the names themselves.

From what you learned above, you know it is now an easy matter to sort both of our vectors, as follows,

stud.names[order(stud.names)] same effect as sort()

And, obviously, sorting the names by the heights is exactly analogous, one way to plot the result is:

barplot(stud.heights[order(stud.heights)], names.arg=stud.names[order(stud.heights)], ylab="Height (m)", main= "Student Heights")

I have conveniently skipped over an important concept, because R handles it fairly intuitively, but I want to mention the terminology. The variable stud.names and the results of ls(), for example, are called vectors of strings or character arrays. R handles them conveniently, so we don't need to worry too much about them, but knowing the terminology will improve your understanding of R's in-line help documents.

SECTION B. More on Data Frames

Before we cover advanced topics of data frames, I wanted to point out the function data.frame() which puts data together to form data frames.

First I want to generate a vector of student class-years to correspond to the stud.names before creating a data frame (Freshmen as 1, Sophomores as 2, etc.).

stud.years <- c(4,2,2,3,1,3)

Now making a data frame is easy (each argument will just add more columns to the data), the only trick being that we have to assign the constructed data frame to a variable, as follows,

student.data <- data.frame(stud.heights,stud.years)

student.data

Voila! Your own data frame. But, wait, where our the student names? And can we have better column headings than our redundant variable names?

The answers lie in two new functions that we will use with assignment notation, names() and row.names(). Let's take a look:

names(student.data)

row.names(student.data)

What we see is vectors of strings corresponding to the columns and rows, respectively. We can change these by assigning replacement strings to the indexed values or by substituting our own vector of strings.

names(student.data)[1] <- "heights"

names(student.data)[2] <- "class.years"

row.names(student.data) <- stud.names

The result is downright beautiful:

student.data

The assignments above are the first of many examples in R that seem to defy logic: it seems as though we're assigning something to a function, which shouldn't make sense because a function isn't a variable. In fact, you can think of the functions names() and row.names() as "access functions" - they do not perform an action, but merely grant access to a property of the argument variable, and this is why we can make assignments of the sort seen above.

Indexing data frames: brackets, logical, or numerical vectors are still the way to access data frames, but with a slight complication, because data frames are multidimensional. The solution (which also holds for matrices, etc.) is to separate the two dimensions with a comma. R treats the first entry as the row number and the second entry as the column number; thus, to access the second column of the fourth row, type

student.data[4,2]

Or the last three rows of the second column,

student.data[4:6,2]

Not too tricky? There are two further complications.

To access an entire row or entire column, leave the index blank, as in,

student.data[,1] FIRST COLUMN

student.data[3,] THIRD ROW

student.data[,] ENTIRE FRAME, equivalent to "student.data"

The only other complication is the ability to enter the names() or row.names() as indices:

student.data[“Justin”,]

Putting all of this together, we can quickly generate subsets of our data:

tall.students <- student.data[student.data*height* > *mean*(*student.data*height),]

Or sort our data by various aspects:

student.data[order(student.data$class.years),]

Introduction to factors When performing statistical analyses - we often want R to look at a set of data and compare groups within the data to one another. For example, you have our data frame containing data on students in a course. There are two columns of data, height and class.year. How can you look at the means of height by class.year?

Or, another example, you have sampled a number of rabbits and have a column for weights before a diet treatment and a column for weights after a diet treatment and a third column stating the diet treatment (e.g, “none”, “grain diet”, and “grapefruit diet”). How can you evaluate the change in weight as affected by diet?

The answer to these questions is to use factors. You can think of factors as categorical variables.

Many of the datasets that come with R already have their data interpreted as factors. Let’s take a look at a dataset with factors:

Note that you may have to install and load the DAAG package in order to load these data. To do so, in Rstudio click on the packages tab then click on Install Packages and type in DAAG in the window that appears and click Install. Verify that the DAAG package now appears in your list of installed packages. Check it’s box to load it.

data(moths, package=“DAAG”)

help(moths, package=“DAAG”)

moths

The help file tells us that our last column, habitat, is a factor. What does this mean?

See what happens when we pull up this column by itself:

moths$habitat

It looks pretty standard, at first, but then we notice that it is more than just a list of habitat names - it has another component, levels.

Factors have levels. Levels are editable, independent of the data itself. To see the levels alone, you can type

levels(moths$habitat)

When called that way, it has the identity of a vector of strings.

The levels() function behaves just like the names() and row.names() functions (i.e., weird), and you can make assignments or reassignments to the levels

levels(moths$habitat)[1] <- “NEBank”

Factors will be exceptionally handy when performing statistical tests, but the various plot functions can give you an idea of uses of a factored variable, such as,

boxplot(moths*meters moths*habitat)

The tilde, ~, used in a number of contexts in R, can generally be read as “by” which gives a general explanation of its use here - visualizing meters by habitat.

Making a factor: Now you know how to employ a factored variable, and the next step is to know how to make a factor out of a variable. The general syntax is:

x <- factor(c(“A”,“B”,“A”,“A”,“A”,“B”))

For vectors of strings, like that one. The results are usually fine as is.

But let’s go back to our student.data data frame. We listed class.years as a number 1 through 4, but these are discreet categories with well-defined names. A more elegant solution is to factor the column of the data frame, much like is seen with moths.

student.data$class.years < -factor(student.data$class.years)

levels(student.data$class.years)

Not ideal, but we can use reassignment to change the names of the years.

levels(student.data$class.years) <- c(“Freshman”, “Sophomore”,“Junior”, “Senior”)

With satisfying (preliminary) results available with:

student.data

boxplot(student.data$heights~student.data$class.years)

So far, we have been using data contained within R’s packages; however, when working on your own data/research you will most likely want to read in a dataset of your own. The read.table() function, and a number of related functions designed for reading in data in a variety of formats, are essential for importing your own data. I suggest trying this out at some point, and I wanted to mention a convenient GUI tool for retrieving the data from your drive - incorporating file.choose() into the command, as follows.

roo <-read.table(file.choose(),header=TRUE, sep=“ “)

(note: you may have to minimize your current window to see the file.choose window that pops up!)

This will let you use yoursystem’s familiar file-selection window to locate the data on your drive. Note: the sep = “ ” argument specifies that white space deliniates your data. If this is not the case, you’ll
have to specify the correct deliniation. For example, if you have data stored in a spreadsheet (e.g. excel), you can save the spreadsheet as a csv file (comma separated values) and then specifiy sep = “,” to read it into R

Applying functions to data frames

Many functions you might like to apply to your data frames will produce unpredictable results.

A few work nicely: Here we access the built in airquality data in R (you can type ?airquality to get more details)

nyc.air <- airquality[,c(“Wind”,“Temp”)]

nyc.air

colMeans(nyc.air)

summary(nyc.air)

But others that you might try do not work as you want:

sum(nyc.air) sums wind and temperature together

var(nyc.air) gives covariance in a matrix (probably not what you want!)

The solution to these troubles is to use the function sapply(), which performs the function named in the second argument on the first argument - in a more predictable fashion than seen above.

sapply(nyc.air, sum) totals each column separately

sapply(nyc.air, var) computes variance of each column separately

SECTION C. Composing your own functions

A more advanced (and very important) topic - So far in R we have used the functions that come with R and its various packages. You have come up with methods for adjusting your data for visualization on your own, but you did this in many separate steps, each of which refer to the specific items you are manipulating. Since often you want to perform the same series of actions on different objects, R makes it relatively easy to compose your own generic functions and store them in R's memory.

Before you start writing a function you need to have your mind set on three things:

- 1) What you want to give the function as input, 2) What you want the
- function to do, 3) What you want the function to give as output

A trivial example

Imagine you need to repeatedly transform sets of data, but your transformation is "non-standard." For this example, I'm imagining that you want the natural logarithm of the data, plus one. We know how to perform these operations on a number we have stored in a variable, no problem,

x <- 1:10

log(x)+1

But what we would really like is a named function which will do this in one step, log.plus.one().

What we will do is make an assignment to log.plus.one, but rather than assigning a value (or vector, etc.), we assign a function which we define on the spot. We use the command function, which looks like a function but is not a function. What is a function? It's a control element of the R language - it isn't executed like a function, but rather it informs R to treat the code around it in a special way.

The command function has an interesting syntax. Its arguments are the names of variables which will serve as the arguments for your function (the first of three bullets, above). Then, after this parenthetical bit, comes the meat of the function - what you want it to do and what you want it to give back to you (the last two bullets, above). In our log.plus.one() case, what we want it to do and what we want it to give back happen to be the same thing, therefore we can define it very simply, as follows,

log.plus.one <- function(y) log(y)+1

Cool! Let's test it out:

log.plus.one(x)

It behaves just like we would want it to.

A separate little world

Wait a second. I used y in my function definition but called the function with my variable x as the argument. What happened to y?

y

The variable is untouched by the function.

In order to keep functions fully generic, when you give the function command, R generates a separate, untouchable variable space which has no interactions with your R workspace. This means that the names of your function arguments (and any variables assigned within your function) can be anything you find convenient - there is never any risk of a conflict with your active variables.

Longer functions

Either because the function is too complex to be executed on a single line or because you want to make the function's methods clearer, you will often generate functions longer than one line. For this purpose, R introduces another type of bracket, curly brackets, { }. These are control brackets, and indicate the contents should be treated as a unit.

As a final example,

```
(function(x,y){z <- x^2 + y^2 x+y+z })(0:7, 1)
```

Note that the function is written on two lines, but this isn't an issue because of the brackets. Note also that this function is anonymous. It is never assigned, but used in place.

A common tendency when first learning to program is to write code in a condensed form (such as the anonymous inline function defined above) but this can make it difficult to follow what is going on when you return to the code later on. While writing code in this way takes a certain amount of cleverness and demonstrates that you have understood the concepts, it is better practice to write out your code so that it is easy to follow. This includes using plenty of whitespace, to make your code easy to read, and thoroughly commenting your commands as you go. The example above is therefore better written as follows:

SUM.VALS.PLUS.SUM.SQS() - a function that takes two numerical values as input and returns the sum of the values plus the sum of their squares:

sum.vals.plus.sum.sqs <- function(x,y) { z <- x^2 + y^2 define z as the sum of the values's squares

return(x + y + z) add the values to the sum of their squares and return the result as output }

Perform the above function with x equal to the numbers from 0 to 7 and y equal to 1:

```
sum.vals.plus.sum.sqs(0:7,1)
```

That's the end of Rscript2! Happy function writing!