Introduction to R

Code

Brian D. Davison

26 January 2022

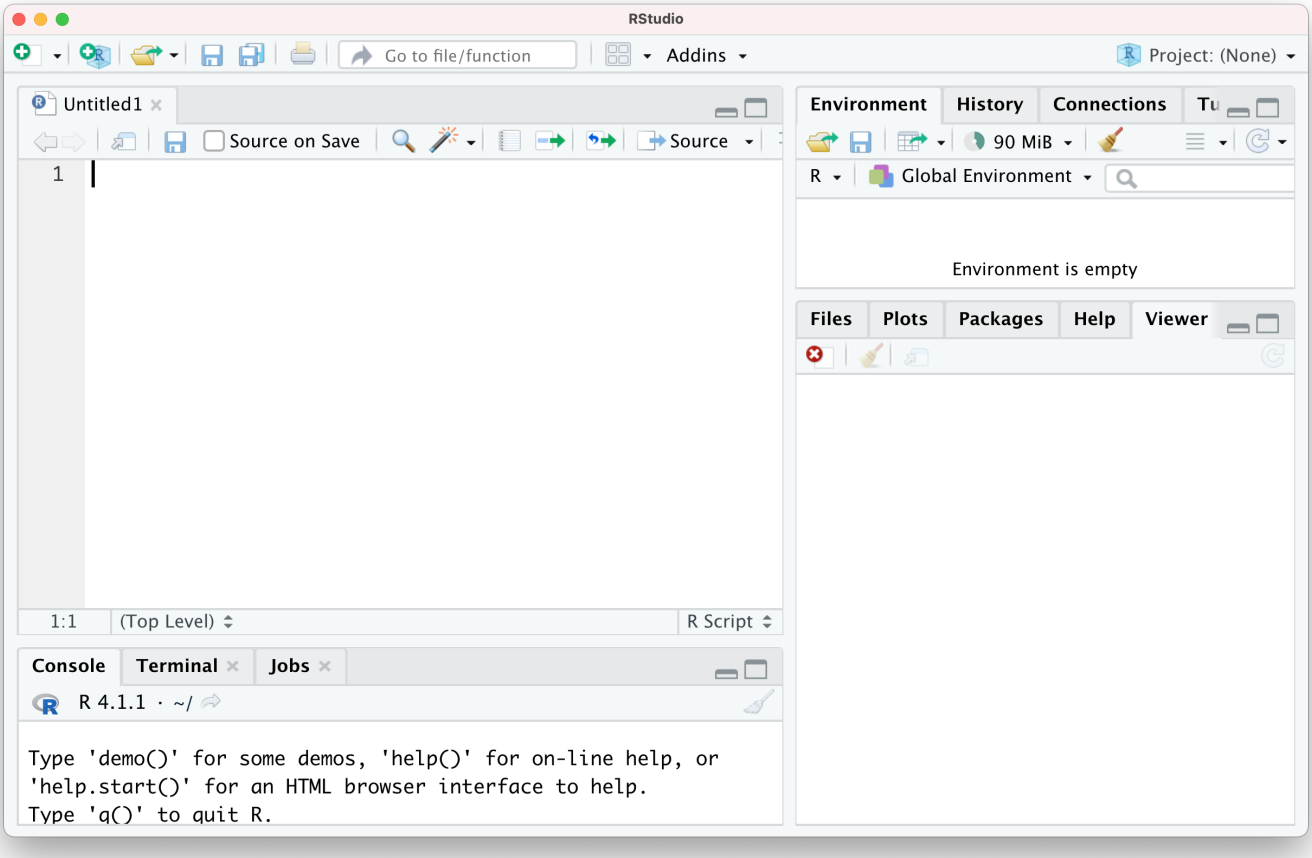
0. Labs in this course

Most labs in this course are virtual, self-paced exercises. You'll have a document (like this one) that will guide you through the material and exercises and when complete, you'll have something that you'll need to submit in Coursesite to get credit. Note that in labs you are encouraged to talk to classmates and post to Piazza with questions or comments.

In general, I recommend going through the lab in sequence, **making sure to read everything**, as later parts typically depend on earlier parts.

1. Getting Started with R

At this point, I assume that you have already installed R and RStudio. RStudio is an integrated development environment (IDE) for R. It isn't strictly required, but it does make working with R easier and we will use it throughout the semester. Start RStudio, and it should open to a display something like this:



We are going to start by spending time in the *Console* window at the bottom left hand side. This window is where R lives, and responds to commands. So let's make it bigger — you can click on the window maximization icon at the far right of the console window, or double click on the gray area to the right of the last tab before the icons at the right. Either way, that will maximize the window vertically.

Inside the Console window, you'll see a prompt ("**>**") where the cursor is. Here we can type R commands or expressions, such as `3 + 4`. Try it! You'll see that R will evaluate the expression and show you the result, and you can type any arithmetic expression using operators +, -, *, /, and ^ for powers.

---

2. Getting and Using R Packages

R has tens of thousands of packages that contain libraries and data that make your life easier – no need to reinvent the wheel!

Generally, you will need to install a library once (which puts it on your machine) and then whenever you want to use it in your code, you just load it.

The R command `install.packages("XYZ")` will automatically retrieve and install package called XYZ from a CRAN server. (*No, XYZ is not a real package name and you'll get an error if you try to install it.*) You only need to do this *once* per version of R. For this reason, we type such commands in the console, but never as part of a script or program since you don't want to re-install a package every time you run the code.

The R command `library(XYZ)` will load it from your installation. (Note that quotes around the package name were not required here, but you can add them if you want.) This is needed every time you start R, and doesn't hurt anything if you do it more than once.

Packages that we load and use in this way are open source and have code and documentation which you can view at https://cran.r-project.org/ (https://cran.r-project.org/).

---

3. A taste of what R can do

Some libraries are already installed by default in R. One of them is the MASS library, so let's load it so that we have access to a dataset called *cats*: `library("MASS")`

Now that the library is loaded, let's use the dataset to actually do something. Copy or type the following two lines into the console:

```
with(cats, plot(Bwt, Hwt))
title(main="Heart Weight (g) vs. Body Weight (kg)\nof Domestic Cats")
```

You'll see that R has generated a scatter plot of the data in the cats dataset, either in the Viewer panel to the right or in a new window. And from that plot, we can easily see that body weight and heart weight are positively correlated (since the points generally go up and to the right). You can effectively read the first line as *using the cats dataset, plot a scatterplot from the Bwt and Hwt fields*. R automatically labeled the axes with the field names, and the second line added a title to the plot. R makes it easy to plot data. We will see much more about plotting later.

Now how about a more informative plot:

```
# using the cats dataset, plot a scatterplot of Body Weight vs Heart Weight
# with points differentiated by sex (color and symbol)
with(cats, plot(Bwt, Hwt, type="n", xlab="Body Weight in kg", ylab="Heart
Weight in g", main="Heart Weight vs. Body Weight of Cats"))
with(cats,points(Bwt[Sex=="F"], Hwt[Sex=="F"], pch=16, col="red"))
with(cats,points(Bwt[Sex=="M"], Hwt[Sex=="M"], pch=17, col="blue"))
```

A few things to notice about this code. First, we added a few lines of comments. Anything following the # symbol on that line is ignored. Second, R often doesn't mind if you start a command on one line and finish it on another. Finally, this is a more complex plot. The first line doesn't actually plot any points (`type="n"`) but prepares an empty plot, with specified axes names. The second line adds the female hearts (only plotting the points for which the Sex was F), and uses a distinct symbol (`pch=16`) and color (`col="red"`). Feel free to try other symbol numbers and color names. (We will work to understand the syntax here in a future lab.) This more informative plot makes it clear that generally female cat hearts are smaller than male hearts.

Now, we've been using the cats dataset, and you haven't seen what it contains. You can see it directly, simply by typing the name (`cats`) at a console prompt. You'll see all the rows have row numbers, and if you scroll back to the beginning you'll see the names of the columns (fields). This dataset is pretty small—a large one would make you wait minutes before it had finished printing its contents, and you wouldn't want to scroll backwards! So instead of viewing datasets directly, we might want to see some information about the dataset. Use the command `summary(cats)` instead, and you'll see information about each field, such as how many of each type of heart are present, and statistical summaries of the real-valued fields. This is much shorter, and often tells us useful information, like the mean body weight.

---

4. Better demonstrations for graphics

While showing a scatterplot is quite functional, R is capable of much more interesting graphics. Run each of these commands (one line at a time), and hit return to move from image to image.

```
demo(graphics)
demo(image)
demo(persp)
```

---

5. R Basics

R is command-line driven

- but it can produce graphical output
- it is case sensitive
- commands are either expressions or assignments
- commands are separated by a newline or the semi-colon (;)
- comments start with a hash (#)
- command history is available with arrow keys (no need to retype everything from the previous line!)
- quit R with `quit()` or `q()` or use the menu item

In R, `source("commands.R")` will execute recorded commands from the file specified. If you wanted to run the same R actions from the command line (e.g., a shell), you would type `R --no-save < commands.R` .

R will evaluate any expression you give it, including assignments and functions.

```
3 + 4 # prints 7 (as we already saw)
x <- 3 # assigns 3 to x
x*x + 4 # prints 13

citation() # calls function to show a message on how to give the creators credit for wor
k using R
```

Notice that we use the `<-` symbol for most assignments in R. (While `=` would also work in this case, the equal symbol is used in a few other cases which would be confusing, and the R community has standardized on `<-` .)

---

6. Objects in R

Entities that R creates and manipulates are called **objects**.

- this includes variables, arrays, strings, functions, or structures built from them
- E.g., we can easily create an array: `x <- c(10.4, 6.5, 3.1, 5.4, 21.7)` (**do this!**)

We can see the names of objects in the current workspace using `objects()` [or `ls()`].

- Omitting the () will show you the contents (code) for the object (when that object contains a function)

`rm(x)` will remove object x from workspace

Try these! In particular, make sure x is assigned the contents of the array specified above.

---

7. Getting Help

If you want to know what an R function does, ask for help!

- `help()` provides documentation: e.g., `help(mean)` or `?mean`

Sometimes seeing examples is more useful

- `example(mean)`

If you don't know the name of the function, you can search for it

- `help.search("median")`

If you know part of the name, you can just search function names

- `apropos("mean")`

---

8. Simple vector manipulations

Typing `1/x` (**try it!**) produces `[1] 0.09615385 0.15384615 0.32258065 0.18518519 0.04608295`. `1/x` is a form of vector arithmetic.

As we've already seen, the `c()` function concatenates its arguments to produce a vector. Those arguments can also contain vectors: `y <- c(x, 0, x)`.

Typing the expression `y` tells R to show you its contents, and produces `[1] 10.4 6.5 3.1 5.4 21.7 0.0 10.4 6.5 3.1 5.4 21.7`. A vector can hold only one type of value. We will see other kinds of values below. (For the CS students, note that when we created y, the type of the 0 value argument was converted from an integer to a real to match the other real values.)

---

9. Vector arithmetic

Arithmetic operations (as seen above) are performed element by element.

Vectors need not be the same length; shorter vectors are recycled as needed (even fractionally) to match the length of the longest vector. Constants are simply repeated.

Consider `v <- 2*x + y + 1` (**put it into your console, assuming you still have `x` and `y` defined**)

- This expression produces a new vector `v` of length 11 (which was the length of y) and a *warning* (which is not an error, as the computation still completed).
- You can think through the evaluation of this expression as follows:
  - `2*x` is performed first, generating a new vector of length 5 (since x was a vector of five elements)
  - The result of `2*x` is added to y. However, since y is length 11, the first vector will need to be repeated a little more than twice to generate a length 11 vector. Once both vectors are the same length, then the first elements can be added together, then the second elements, then the third, etc. This process results in a new length 11 vector.
  - The resulting length 11 vector is then added to 1; 1 is a constant, but in vector arithmetic a constant is repeated as many times as needed to be the same length. Thus, the value 1 is added to each of the 11 elements, producing the overall resulting 11 element vector, which is then assigned to v.

Other functions with a single argument can be applied similarly to vectors. log(), exp(), sin(), cos(), tan(), sqrt() all have usual meanings.

---

10. More working with vectors

We saw above that we could easily perform element-by-element arithmetic operations on vectors. Some operations can compute results from the contents of vectors

Let's assume that you have run `x <- c(10.4, 6.5, 3.1, 5.4, 21.7)`.

Then `max(x)`, `min(x)`, `length(x)`, `sum(x)` all have obvious meanings. A little more interesting is that `range(x)` returns a vector with two values, equivalent to `c(min(x), max(x))`, e.g., `[1] 3.1 21.7`. Other obvious functions: `mean(x)` produces `9.42`; `median(x)` produces `6.5`; `prod(x)` produces `24556.24`. There are many other functions that calculate something over the entire vector.

---

11. Regular sequences

In R, we can abbreviate simple sequences of numbers. For example, `1:30` is the same as the vector `c(1, 2, 3, ..., 29, 30)`, as is `seq(1,30)`. You can try all of them in the console, and they will print out the same vector.

The colon operator has high precedence so `2*1:15` means multiply the vector of values `1:15` by `2`, resulting in the vector `c(2, 4, ..., 28, 30)`. And naturally, `30:1` generates the decreasing number sequence `c(30, 29, ..., 2, 1)`.

The `seq()` function is a more general mechanism. If you look at the documentation for it, you'll find that it has five arguments, but fortunately they are not all needed at once. Moreover, *arguments can be named* so that the order you pass arguments in doesn't matter! This is quite different from languages like C or Java.

For example, consider this expression: `seq(-5, 5, by=.2) == seq(length=51, from=-5, by=.2)`. What do you think it does? Think through what each part of this is doing before trying to determine the overall result of the expression. (Remember you can ask questions on Piazza if something doesn't make sense!) Note that we've used a mixture of named and unnamed parameters when `seq()` is called. The unnamed parameters are matched to the function arguments in the original sequence (i.e., the first unnamed parameter is mapped to the "from" argument).

---

12. Character vectors

A character vector is denoted as a sequence of characters delimited by double quotes ("), but you are permitted to use either single or double quotes.

A useful function is `paste()`, as it takes an arbitrary number of arguments, interprets them as strings, and concatenates them one by one into character strings.

- First try: `paste("A", "hello", 4, "you")`
- Then try: `labs <- paste(c("X", "Y"), 1:10, sep="")`

Do you understand what the `sep` argument does? You might need to try some variations of this to fully understand why it works this way.

13. Index vectors

Vectors are much like arrays in other languages, and we can we can extract a single value by specifying the index position in square brackets. Note however, that **R counts positions starting at 1**. Thus, if `y <- c(1, 3, 5, 7, 9)`, then `y[2]` produces the value `3`.

However, unlike other languages, we can also select a subset of a vector (not just a single value) if we provide an index vector rather than a single index value. So, `y[1:2]` produces a vector containing the first two elements of `y`, i.e., equivalent to `c(1, 3)`. Alternatively, we can specify the positions of items to exclude from the results, by negating them. For example, `y[-2]` produces all values, in original order, except for the value in position 2, i.e., `[1] 1 5 7 9`.

We can also use a logical index vector instead. For example, consider the vector `y>6`. This is a five-valued logical vector that produces `[1] FALSE FALSE FALSE  TRUE  TRUE`. If we use it as an index vector, e.g., `y[y>6]` we see that the only values in the result are those that are greater than 6. Note that `y>6` generates a vector that is exactly the same length as `y`. You should try using an index vector that is shorter, and see what happens. (If you can't think of how to create an index vector that is shorter, consider creating y1 <- y[1:4] which only contains four elements. Now use it in a comparison expression to generate a logical vector that you can use as a logical index vector.) Similarly, try one that is longer. Both are valid operations in R (in that they do not cause an error and stop the computation).

14. Index vectors with named positions

We saw previously that a function can have named arguments. The same is true for positions of vectors. We can use a vector of character strings to name positions. Imagine `fruit` was a vector of prices.

Hide

```
fruit <- c(5, 10, 1, 20)
names(fruit) <- c("orange", "banana", "apple", "peach")
lunch <- fruit[c("apple","orange")]
lunch
```

In the above, we see that `fruit` and `lunch` are numeric vectors (that is, they are vectors containing numbers). Thus, we could do arithmetic (e.g., `15*lunch` to learn how much it would cost per kind of fruit for 15 lunches). But we can access elements of those vectors using the names as indexes. There are some obvious questions you might ask (like what happens if you have the same name in the character vector, or too few or too many names). Try such situations out yourself.

15. R control structures

There are three kinds of control structures supported by R that should sound familiar: **if-else**, **for loops**, and **while loops**. If you don't know what these are from a prior course, you should look online for a relevant tutorial before continuing.

```
# if-else
if (fruit[1] > fruit[2]) {
   cat(names(fruit)[1], "is larger")
} else {
   cat(names(fruit)[1], "is not larger")
}
```

Here we see a number of things. First, the `if` logical condition goes inside of parentheses. Second, we've used curly braces `{}` to hold blocks of statements (here just one statement), which is just like other languages you may have seen. We've used a new function `cat()` which concatenates its arguments and outputs them to the screen. (The indentation of this line is for readability, not a requirement of R.) And finally, `names(fruit)[1]` might look strange, but if you think about it, it can make sense. We know that `names(fruit)` is valid, and generates a vector of character strings, and so the `[1]` is being applied to that, resulting in a single string (the name of the first fruit) to be concatenated by `cat()`. Note that the `else` component is optional and can be omitted.

```
cat("\nTry a for loop\n")
# for loop - output each fruit
for (f in fruit) print(f)
```

This is an example of a for loop, which is a bit different in R than other languages. Here, inside the parentheses after the `for`, we specify a loop variable (`f` in this case), which takes on the values of the elements of the vector that follows the keyword `in`. Thus, we **must** have a vector of values to iterate over (i.e., the loop is performed as many times as there are values in the vector, and each time the loop variable is assigned the value of the next element in the vector). Finally, the loop itself is simply the call to the `print()` function, which is another way to output to a screen, but it automatically generates a newline after printing its content, unlike `cat()` which requires that you specify newlines manually, which we do in this example.

```
print("Try a while loop")
# while loop - output the fruit that are in numerically growing order but otherwise stop
i <- 1
while (fruit[i] < fruit[i+1]) {
   print(fruit[i])
   i <- i + 1
}
print(fruit[i])
```

So, much like the `if()` statement, the `while()` loop requires the condition to be in parentheses, and we see that there is a block of code that is executed repeatedly as long as the condition is TRUE.

---

16. R functions

We've used a number of functions already, such as `cat()`, `print()`, `paste()`, `seq()`, and `median()`. Creating a function in R is pretty straightforward:

```
myfunction <- function(arg1, arg2, ...) {
   statements
   return(object)
}
```

The above isn't a real function definition, but it shows the format. To create a function, we assign an object the result of the function `function()` which defines the code. `function()` takes an arbitrary number of parameters to define the arguments, and allows one or more statements in the block within curly braces. Finally, it can return an object using the `return()` function, which can be a simple value or a complex object like a vector (as we saw with the `range()` function).

Here is a real example, only slightly more complex.

Hide

```
mysummary <- function(x, show=TRUE) {
  center <- mean(x); spread <- sd(x)

  if (show & center > 100) {
    cat("Big Mean=", center, "; SD=", spread, "\n", sep="")
  } else if (show) {
    cat("Small Mean=", center, "; SD=", spread, "\n", sep="")
  }

  return( c(center, spread))
}
```

This has created a function called `mysummary()` which calculates and optionally prints the mean and standard deviation of a vector of numbers. Once defined, we can use it:

Hide

```
mysummary(y, FALSE)

ss <- mysummary(2*y)
```

As we can see, the first time we asked `mysummary()` to not print anything, and so we only see the result (what was returned from the function). The second time, we did not override the default value of show and so it printed it nicely. Since we captured the results into ss, there is nothing else generated as output. If you try calling `mysummary()` with large enough values, you'll get the other output.

---

17. Creating R programs

An R program is simply a series of R commands that is executed in sequence. At the beginning of this lab we maximized the console, which minimized the open file window above (probably called "Untitled1"). If there is no such window, you'll need to create it (File->New File->R Script) which will show above the console. This space is an editor that holds code and can be saved to a file. You'll need to resize things so you can get into that window. You can type commands into that window, typically one per line. The **Run** button (just above and mostly to the right side of your code) will run the code line where your cursor is (or if you highlight a bunch of lines, it will run all of them). And by "run" I mean that RStudio will copy them to the console for execution. (Look at your console

history if needed to see what it did for you.) RStudio has many other options for running code; look in the Code menu item and you'll see that there are short-cuts for running the selected lines and even for running everything in the editor (this one is found in **Code->Run Region->Run All**).

Please do the following in the editor, which you will submit as CW2 in Coursesite:

1. Create a comment with your name and the date
2. Show the result of computing the 32nd power of .8
3. Define a new variable, `j` with a vector of at least three real-valued numbers. Now create a line to output the `sin()` of all the values of `j` with a single expression. Add another line to output the result of adding 2 to the values of `j` with a single expression.
4. Output the exponent of the log of all the values of `j`. Do you get the same values as j had to start?
5. Output the code that implements the function `ls()`.
6. Write code that assigns `x` to be a vector containing the elements `15,63,41,57,91,29`. Add a second line that shows TRUE when the value in `x` is greater than 45 and FALSE otherwise.
7. Assign `a` to be a vector containing the integers from 2 through 5 (using the simplest shortcut we saw today). Set `b` to be a vector of two strings "X" and "Y". Show the content of a new vector using `c()` that contains `a` and `b`. Note that a and b are vectors holding very different kinds of values. Did you notice what happened to the values of a when you combined them together? This is because a vector can hold only one kind of value.
8. Assign `c` and `d` to each hold an integer value. Write code that if c and d have different values, assign `v` to be 5, otherwise assign `v` to be 10. Make sure to test your code by trying a variety of values for c and d.
9. Write a simple function (you can choose the name) that takes one integer argument. If that argument is missing, its default value is 10. If the provided value is less than 1, output an error message. Otherwise, print the letter X as many times as the argument value.
10. Make sure that everything you've written can be run as is by a grader (i.e., do a **Run All**). Save this R file as *yourlastname.R*. Submit this file as your answer to the IntroR Lab assignment.

That's all for today!