Lesson 04 - Getting Started with R

Last Updated 06-29-19

Introduction

This lesson is designed to explain the basics of how R works as a programming language.

Learning Objectives

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
- Assign values to objects in R.
- Learn how to *name* objects
- Solve simple arithmetic operations in R.
- Call functions and use arguments to change their default options.
- Inspect the content of vectors and manipulate their content.
- Subset and extract values from vectors.
- Write logical statements that resolve as TRUE and FALSE.
- Describe what a data frame is.
- Summarize the contents of a data frame.
- Extract vectors out of data frames using variable names.

Writing Scripts

In lesson 02 we saw that we could write R code in the console and get immediate results. There are two main ways of interacting with R: by using the console or by using script files (plain text files that contain your code). We will be working in R markdown files exclusively in this class, but it is important to be aware that there are also script files that have an extension of `.R`. These can contain code and comments only, not normal text sentences like this.

Because we want our code and workflow to be reproducible, and often your code may span several lines at a time, it is better to type the commands we want in a script, and save the script. This way, there is a complete record of what we did, and anyone (including our future selves!) can easily replicate the results on their computer. It's also easier to fix mistakes this way, without having to retype in the entire command.

Start a new Rmarkdown file

- Go to File –> New File –> R Markdown to open a new R markdown window.
- Give this file a name such as "Lesson 04 notes", and put your name as the author.
- Delete all the template language below line 11.

Now let's go back to that long expression from lesson 2 (corrected), but this time type it into a new code chunk. Recall we can make a new code chunk by pressing `CTRL`+`ALT`+`I`, or by clicking on *Insert* then *R*. Also recall that we submit this code by pressing `Ctrl`+`Enter` or clicking the green play arrow in the top right corner of the code chunk.

```
2 + 5*(8^3)- 3*log(10)
```

Make a change to the above expression and run the command again.

For the rest of this lesson, when asked to try out some code type it into a new code chunk.

Creating objects in R

To do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`.

- You want your object names to be explicit and not too long.
- They cannot start with a number (`2x` is not valid, but `x2` is).
- R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`).
- There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list).
- It's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`) because these already tend to be in use by different parts of R.
- See Google's style guide for more information.

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
```

```
## [1] 55
```

```
weight_kg          # and so does typing the name of the object
```

```
## [1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the animal's weight in pounds in a new object, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

R executes code in top-down order. So what happens on line 10 occurs before line 11. What do you think is the current content of the object `weight_lb`? 126.5 or 220?

Comments. *The comment character in R is* `#`*, anything to the right of a* `#` *in a script will be ignored by R. It is useful to leave notes, and explanations in your scripts as demonstrated earlier.*

Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R *packages* (lesson 02).

A function usually takes one or more inputs called *arguments*, and often (but not always) return a *value*.

A typical example would be the function `sqrt()`. The input is the number `4`, and the return value (the output) is the square root of 4, namely 2. Executing a function ('running it') is called *calling* the function.

```
sqrt(4)
```

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

Let's look into the `round` function.

```
round(3.14159)
```

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

We can learn more about this function by typing `?round`. The **Usage** section of the help documentation shows you what the default values for each argument are. This is a very important piece to pay attention. Sometimes the default behaviors are not what you want to happen.

```
round(x, digits=0)
```

In the **Arguments** section the help file defines what each argument does.

- `x` is the object that you want to round. It must be a *numeric vector*.
- `digits` is an integer indicating the number of decimal places to round to.

Above, we called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. We see that if we want a different number of digits, we can type `digits = 2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

This is a simple function with only one argument. Functions are the backbone of how R does it's thing. You will get lots of practice with functions, and quickly encounter functions that require many arguments.

Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g`:

```
(weight_g <- c(50, 60, 65, 82))
```

```
## [1] 50 60 65 82
```

A vector can also contain characters:

```
(animals <- c("mouse", "rat", "dog"))
```

```
## [1] "mouse" "rat"   "dog"
```

The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume objects have been created called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(weight_g)
```

```
## [1] "numeric"
```

```
class(animals)
```

```
## [1] "character"
```

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

Data Types

R objects come in different data types. You just saw vectors comprised of numbers (**numeric**), of words (**character** or **string**). The other common data type that we will be encountering in this class is a `"logical"` (sometimes called Boolean). These are created by writing a logical statement where the answer is either TRUE or FALSE. Silly examples include "Is 3 greater than 4?" and "Is the square root of 4 equal to 2?"

```
3>4
```

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

```
sqrt(4)==2
```

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

We will see how to use these logical statements to do things such as subsetting data and creating new variables.

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`). We will only talk about `data.frames` and `factors` in this class.

Doing math on vectors

You can perform math operations on the elements of a vector such as

```
weight_KG <- weight_g/1000
weight_KG
```

```
## [1] 0.050 0.060 0.065 0.082
```

When adding two vectors together, the elements in the same position are added to each other. So element 1 in the vector `a` is added to element 1 in vector `b`.

```
a <- c(1,2,3)
b <- c(6,7,8)
a+b
```

```
## [1]  7  9 11
```

More complex calculations can be performed on multiple vectors.

```
wt_lb <- c(155, 135, 90)
ht_in <- c(72, 64, 50)
bmi <- 703*wt_lb / ht_in^2
bmi
```

```
## [1] 21.01948 23.17017 25.30800
```

All these operations on vectors behave the same way when dealing with variables in a data set (data.frame).

If you want to add the values *within* a vector, you use functions such as `sum()`, `max()` and `mean()`

```
sum(a)
```

```
## [1] 6
```

```
max(b)
```

```
## [1] 8
```

```
mean(a+b)
```

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

Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
```

```
## [1] "rat"
```

```
animals[c(2, 3)]
```

```
## [1] "rat" "dog"
```

The number in the indices indicates which element to extract. For example we can extract the 3rd element in `weight_g` by typing

```
weight_g[3]
```

```
## [1] 65
```

Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not. Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests such as:

```
weight_g > 50  # returns TRUE or FALSE depending on which elements that meet the condition
```

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

We can use this output to select elements in a *different* vector where the value of that logical statement is TRUE. For instance, if you wanted to select only the values where weight in grams is above 50 we would type:

```
weight_g[weight_g > 50]
```

```
## [1] 60 65 82
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

Weight is less than 30g or greater than 60g

```
weight_g[weight_g < 30 | weight_g > 60]
```

```
## [1] 65 82
```

Weight is between 60 and 80lbs

```
weight_g[weight_g >= 60 & weight_g <= 80]
```

```
## [1] 60 65
```

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

A common task is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

```
## [1] "rat" "cat"
```

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

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

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

```
## [1] "rat" "dog" "cat"
```

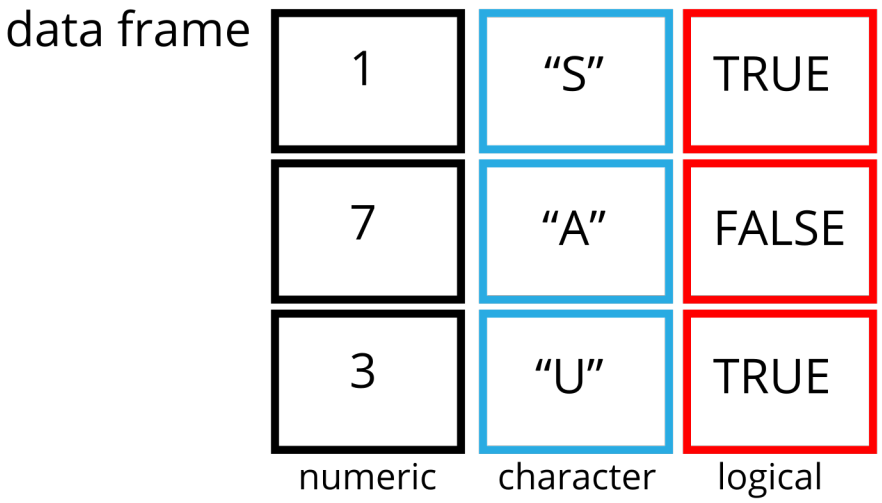

Figure 1: figure depicting a data frame

Order matters.

When considering string or character vectors or data elements, R treats everything in alphabetical order. Thus

```
"four" > "five"
```

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

This will come back to bug you when dealing with categorical data types called `factors` in a later lesson. Don't worry, we'll show you how to be the boss of your factors and not let R tell you that "one" is greater than "four".

Data Frames

Data frames are like spreadsheet data, rectangular with rows and columns. Ideally each row represents data on a single observation and each column contains data on a single variable, or characteristic, of the observation. This is called `tidy data`. We will start to learn some tools to look at the data, and for getting data from an external file into R for analysis.

We will use a data set called `iris` that comes with R for now, but in the next lesson learn how to import data from an external file into R. We can open a data viewer window to see the contents of this data frame by typing

```
View(iris)
```

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors). For example, here is a figure depicting a data frame comprising a numeric, a character, and a logical vector.

When data sets are very large, it may be difficult to see all columns or all rows. We can get an idea of the structure of the data frame including variable names and types by using the `str` function,

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

The iris data set has 5 variables, `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width` which are all numeric variables, and `Species` which is a factor.

Inspecting data.frame objects

We already saw how the functions view() and str() can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

- Size:
  - `dim(iris)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow(iris)` - returns the number of rows
  - `ncol(iris)` - returns the number of columns
- Content:
  - `head(iris)` - shows the first 6 rows
  - `tail(iris)` - shows the last 6 rows
- Names:
  - `names(iris)` - returns the column names (synonym of colnames() for data.frame objects)
  - `rownames(iris)` - returns the row names
- Summary:
  - `str(iris)` - structure of the object and information about the class, length and content of each column
  - `summary(iris)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides a data.frame

Identifying variables

Data frames can be subset by calling indices (as shown previously), but also by calling their column names directly:

```
iris[, "Petal.Length"]
iris[, 3]
iris$Petal.Length
```

The `$` notation has the format `data$variable` and so can be thought of as specifying which data set the variable is in. It is easy to imagine a situation where two different data sets have the same name.

This allows us to perform calculations on an individual variable. Below is an example of finding the mean Sepal length for all irises in the data set.

```
mean(iris$Petal.Length)
```

```
## [1] 3.758
```

You can also subset a variable based on the value of a secondary variable. Here is an example of finding the average Sepal Length for the `setosa` species only.

```
mean(iris$Petal.Length[iris$Species=="setosa"])
```

```
## [1] 1.462
```

Note that the $ is used in both locations where we want to identify a variable.
