Quantitative Understanding in Biology
Introduction to R

Luce Skrabanek, Jason Banfelder

September 3rd, 2019

1 Prologue

1.1 What is R?

R is a free software environment for statistical computing and graphics (`www.r-project.org`). It can effectively analyze large-scale datasets, such as those resulting from high-throughput sequencing experiments. It promotes automated and reproducible analyses of scientific data, creates a wide spectrum of publication quality figures, and has an extensive library of add-on packages to facilitate many complex statistical analyses. Because it is free and ubiquitously available (it runs on Windows, Mac, and Linux computers), your investment in learning R will pay dividends for years to come.

1.2 What is RStudio?

While R is very powerful, it is essentially a command line program and is thus not the friendliest thing to use. Especially when learning R, a friendlier environment is helpful, and RStudio provides this, giving you things you expect in a modern interface like integrated file editing, syntax highlighting, code completion, smart indentation, tools to manage plots, browse files and directories, visualize object structures, etc.

From your computer, choose the RStudio application. This will start R under the hood for you.

2 Introduction

2.1 R is a calculator

1. The Console panel (lower left panel) is where you type commands to be run immediately. When R is waiting for a new command, you will see a prompt character, `>`.
2. To add two numbers, type after the prompt:

   ```
   1 + 2
   ```

   When you hit return, you should see ...

   ```
   [1] 3
   ```

3. The answer is of course 3, but what is the `[1]` before it? It turns out that all numbers in R are vectors, and the `[1]` is a hint about how the vector is indexed. To see a long vector of random numbers, type:

   ```
   rnorm(100)
   ```

   For now we can ignore the vector indexing; we will learn more about vectors and indexing shortly.

4. R understands basic math. Try typing:

   ```
   3 - 4
   5 * 6
   7 / 8
   ```

5. The order of operations is kept (PEMDAS). Note the difference between ...

   ```
   1 + 2 * 3
   ```

   and ...

   ```
   (1 + 2) * 3
   ```

6. You can force R to do integer division using the `%/%` operator (division symbol inside two percent signs):

   ```
   17 %/% 4
   ```

   and to get the remainder:

   ```
   17 %% 4
   ```

7. You can also compute powers:

   ```
   2 ^ 4
   ```

   even with fractional exponents.

   ```
   2 ^ 4.3
   ```

8. R comes with an extensive library of built-in functions.

   ```
   log(4)       # natural log
   log10(4)     # log in base 10
   log(4, 10)   # same as above
   sqrt(9)      # square root
   abs(3-4)     # absolute value
   exp(1)       # exponential
   ```


9. Note in the examples above, we have used comments (preceded by the `#` character). You can type them if you want but they do not add anything to the work that R does. Comments are not usually used when interactively typing commands into the Console, but are essential when writing scripts - stay tuned!

2.2 R has variables

1. It can be really useful to assign values to variables, so they can be referred to later. This is done using the assignment operator (`<-`).

   ```
   us.population <- 3.24e8 # From Wolfram|Alpha
   us.area <- 3719000      # From Wolfram|Alpha
   us.pop.density <- us.population / us.area
   us.pop.density
   ( us.pop.density <- us.population / us.area )
   ```

   Some notes:

   (a) Once a variable is defined, you will see it show up in the environment panel in RStudio.

   (b) R will not automatically print out the value of an assigned variable. Type the name of the variable by itself to see it. Alternatively, wrapping the assignment in parentheses executes the assignment and prints the result.

   (c) Case matters: `US.area` is not the same as `us.area`.

   (d) Word separation in R is traditionally done with periods, but this is slowly losing favor. Other options include snake_case (separated by underscores) or camelCase (capitalize each new word).

2. Often, "quick and dirty" variable names that you will be using often in the Console are named with single letter variables, whereas variables in a script are long enough to be self-explanatory.

   Tip: Note that in RStudio, the `Tab` key will attempt to autocomplete the variable or function name that your cursor is currently on.

3. Use the `rm()` function to get rid of a variable from your environment.

   ```
   rm(us.pop.density) # gets rid of the us.pop.density variable
   ```

   Note that removing variables from your environment can help reduce clutter and is essential when dealing with large objects.

2.3 Working with environments and history

1. You can save your environment (the set of variables you have defined). To do so, click the Save icon in the Environment tab (top right). Once you have saved your environment, the actual R command that was run pops up in your Console. Note that RStudio automatically adds the traditional file extension of `.RData`.

2. To clear your current environment, click the broom icon on the Environment panel. You can also achieve this by typing:

   ```
   rm(list = ls(all = TRUE))
   ```

   Note that all the variables you just defined have disappeared!

3. To load an environment, click the Load icon and select the `.RData` file that you saved earlier. Again, you'll see the corresponding R command in the Console panel. Note that loading an environment does not empty your existing environment, but it will overwrite any existing variables.

4. It is good practice to have a separate directory for each project or analysis that you are working on. If you tell R about this directory, it will, by default, load and save files from it. We call this the working directory. You can browse files and directories from the Files tab of the lower right panel. Set the working directory using the gear icon (the More button). Alternatively, you can use the `ctrl-shift-H` shortcut. Once run, the R command to set the working directory is also shown in the Console tab.

5. When you quit R, you will be asked if you want to save your workspace image (meaning your environment) in your working directory in a file called `.RData`.

6. RStudio always saves your history in your working directory. This can be a problem when restarting RStudio just by clicking on the Dock or the Start menu as your working directory will be your home directory and you will not see the history saved from your last session. On Macs, an easy way to specify your project directory when starting R is to drag the folder you want onto the RStudio icon. For Windows, the icon in your Toolbar does not work; you will have to use an alias on your Desktop. This will also load any saved `.Rdata` and .Rhistory files from that directory.

7. An even better practice is to create a new RStudio project for every analysis. This is RStudio's way of supporting and streamlining the common practice of keeping all the input data, R scripts, and other files associated with that analysis together. This will create a .Rproj file in your project directory. Now, whenever you want to work on that analysis again, opening that .Rproj file will bring you back to exactly where you left off (the same working directory, the same files open, the same command history), although you'll have to re-populate your environment.

8. Note that you can easily copy a line from your history to the Console by double-clicking it, or using the To Console icon.

2.4 Getting help

1. Much work has gone into making R self-documenting. There are extensive built-in help pages for all R commands, which are accessible with the `help()` function. Thus, to see how `sqrt()` works, type:

   ```
   help(sqrt)
   ```

   The help page will show up in the Help section of the RStudio window. In case typing `help` is too long, there is a shortcut.

   ```
   ?sqrt
   ```

2. It should be noted that some special characters or reserved words have to be quoted when using either of the above help functions.

   ```
   ?"+"
   ```

   will show the help page for the arithmetic operators. Note that since the `+` function is just one of a group of similar operators, the help page explains all of them in a single page, rather than having separate pages for `+`, `-`, `*`, `/` etc. The help pages quite often will group similar functions together this way (e.g., the related functions `log()` and `exp()` are found on the same page).

3. Another very useful command is the `example()` function. Almost all R commands will include a series of examples on their help pages (accessible using the `help()` or `?` functions). You can run these

examples directly from your console by using the `example()` function. To see the examples for the `sqrt()` function, type:

```
example(sqrt)
```

This runs the set of examples that are listed at the bottom of the help page, exactly as if you had typed them out yourself.

4. The R help is not always as transparent as one would like and StackOverflow (`stackoverflow.com`) may be a better bet for answering your questions.

2.5 Data types

1. So far, we have only been dealing with numerical data, but in the real world, data takes many forms. R has several basic data types that you can use.

```
has.diabetes    <- TRUE         # logical  (note case!)
patient.name    <- "Jane Doe"   # character
moms.age        <- NA           # used to represent an unknown ("missing") value
NY.socialite.iq <- NULL         # used to represent something that does not exist
```

2. When working with truth values, you can use the logical operators:

```
AND (&)
OR (|)
NOT (!)
```

```
is.enrolled  <- FALSE
is.candidate <- has.diabetes & ! is.enrolled
```

3. R uses tri-state logic when working with truth values.

```
TRUE & FALSE
T | F
T & NA
F & NA
TRUE | NA
FALSE | NA
TRUE & ! FALSE & NA
```

4. R can convert among datatypes using a series of `as.()` methods.

```
as.numeric(has.diabetes)
as.numeric(is.enrolled)
as.character(us.population)
as.character(moms.age) # still NA - we still don't know!
```


3 Data Structures

3.1 Overview

R has several different types of data structures and knowing what each is used for and when they are appropriate is fundamental to the efficient use of R. The ones that we are going to examine in detail here are: vectors, matrices, lists and data frames.

A quick summary of the four main data structures:

Vectors are ordered collections of elements, where each of the objects must be of the same data type or mode, but can be any mode.

A **matrix** is a rectangular array, having some number of columns and some number of rows. Matrices can only comprise one data type (if you want multiple data types in a single structure, use a data frame).

Lists are like vectors, but whereas elements in vectors must all be of the same type, a single list can include elements from any data type. Elements in lists can be named. A common use of lists is to combine multiple values into a single object that can then be passed to, or returned by, a function.

Data frames are similar to matrices, in that they can have multiple rows and multiple columns, but in a data frame, each of the columns can be of a different data type; within a column, all elements must be of the same data type. You can think of a data frame as being like a list, where each element corresponds to a complete vector, and all elements are the same length.

3.2 Vectors

1. We've already seen a vector when we ran the `rnorm()` command. Let's run that again, but this time assigning the result to a variable.

   ```
   x <- rnorm(100)
   ```

2. Many commands in R take a vector as input.

   ```
   sum(x)
   max(x)
   summary(x)
   plot(x)
   hist(x)
   ```

3. There are many ways of creating vectors. The most common way is using the `c()` function, where `c` stands for concatenation. Here we assign a vector of characters (character strings must be quoted).

   ```
   colors <- c("red", "orange", "yellow", "green", "blue", "indigo", "violet")
   ```

4. The `c()` function can combine vectors.

   ```
   colors <- c("infrared", colors, "ultraviolet")
   # remember that "infrared" is a one-element vector
   ```

   By assigning the result back to the `colors` variable, we are updating its value. The net effect is to both prepend and append new colors to the original `colors` vector.

5. You can get the length of a vector using the `length()` function.

   ```
   length(colors)
   ```


6. You can access an individual element of a vector by its position (or "index"). In R, the first element has an index of 1.

   ```
   colors[7]
   ```

7. You can also change the elements of a vector using the same notation as you use to access them.

   ```
   colors[7] <- "purple"
   ```

   Tip: Appending an element is a slow operation because it actually creates a new vector. If you do this a limited number of times, this is fine, but if you are doing this 1000s of times, it is more efficient to create an empty vector of a pre-determined size, and then change the elements. You can create a blank vector using the `vector()` function.

   ```
   a.numeric.vector <- vector(mode="numeric", length=1000)
   a.numeric.vector[50] <- 5
   a.numeric.vector[750] <- 10
   plot(a.numeric.vector)
   ```

8. You can access multiple elements of a vector by specifying a vector of element indices.

9. R has many built-in datasets for us to play with. You can view these datasets using the `data()` function. Two examples of vector datasets are `state.name` and `state.area`.

10. We can get the last ten states (alphabetically) by using R's convenient way of making a vector of sequential numbers, with the ":" operator

    ```
    indices <- 41:50
    indices[1]
    indices[2]
    length(indices)
    state.name[indices]
    ```

    Exercise:

    a. We've seen how to list the last 10 states (alphabetically). How would you list the first 10 states?

    b. How would you list the first 10 **and** last 10 states (alphabetically)?

    c. Can you generalize this so that it works for any arbitrary length vector?

11. We can test all the elements of a vector at once using logical expressions. Let's use this to get a list of small states. First, how do we determine what a small state is?

    ```
    summary(state.area)
    ```

    Next, figure out which states are in the bottom quartile.

    ```
    cutoff <- 37317
    state.area < cutoff
    ```

    Note that this returns a vector of logical elements. We have seen that we can access vector elements using their indices, but we can also access them using logical vectors.

    ```
    small.states <- state.name[state.area < cutoff]
    ```

12. We can test for membership in a vector using the `%in%` operator. To see if a state is among the smallest:

    ```
    "New York" %in% state.name[state.area < cutoff]
    "Rhode Island" %in% state.name[state.area < cutoff]
    ```


13. You can also get the positions of elements that meet your criteria using the `which()` function.

    ```
    which(state.area < cutoff)
    state.name[which(state.area < cutoff)]
    ```

    Techniques like this can be useful for removing outliers from your data.

14. Let's get the area of Wyoming:

    ```
    state.area[state.name == "Wyoming"]
    ```

    Notes:

    (a) The `==` is a test for equality. This is different from assignment.

    (b) The indexing vector here is a logical vector.

15. While this works, it can be a little long-winded. Luckily, R lets us name every element of a vector using the `names()` function.

    ```
    names(state.area) <- state.name
    ```

16. And now we can access Wyoming directly:

    ```
    state.area["Wyoming"]
    ```

17. Here the indexing vector we are using to access elements is a character vector.

    ```
    state.area[c("Wyoming", "Alaska")]
    ```

18. Now we can see all the small states and their areas in one shot:

    ```
    state.area[small.states]
    ```

19. Sadly, not all functions that fetch an element from a vector keep the associated name.

    ```
    min(state.area)
    ```

    But you can find the index at which the minimum occurs, and use that.

    ```
    state.area[which.min(state.area)]
    ```

20. In addition to using the `:` notation to create vectors of sequential numbers, there are a handful of useful functions for generating vectors with systematically created elements.

    ```
    seq(1, 10)     # same as 1:10
    seq(1, 4, 0.5) # shows all numbers from 1 to 4, incrementing by 0.5 each time
    ```

    Let's look carefully at the help page for the `seq()` function.

    ```
    ?seq
    ```

21. The `seq()` function can take five different arguments, but not all of them make sense at the same time. In particular, it would not make sense to give the `from`, `to`, `by`, and `length` arguments, since you can figure out the length given `from`, `to`, and `by`. You can pass argments by name rather than position; this is helpful for skipping arguments.

    ```
    seq(0, 1, length.out = 10)
    ```

    Tip: In scripts, it is often good form to use named arguments, even when not necessary, as it makes your intent clearer.

    ```
    seq(from = 1, to = 4, by = 0.5)
    seq(from = 0, to = 1, length.out = 10)
    ```


22. Take a look at the help again: note that all of the arguments have default values, which will be used if you don't specify them.

```
seq(to = 99)
```

23. Another commonly used function for making regular vectors is `rep()`. This repeats the values in the argument vector as many times as specified. This can be used with character and logical vectors as well as numeric.

```
rep(colors, 2)
rep(colors, times = 2) # same as above
rep(colors, each = 2)
rep(colors, each = 2, times = 2)
```

24. When using the `length.out` argument, you may not get a full cycle of repetition.

```
rep(colors, length.out = 10)
```

25. In many cases, R will implicitly "recycle" vector elements as needed to get operations on vectors to make sense. When vector operations align, results are as you would expect:

```
x <- 0:9
y <- seq(from = 0, to = 90, by = 10)
x + y
```

Here, the first element of `x` has the first element of `y` added to it, the second element of `x` has the second element of `y` added to it, etc. What happens, though, when the vectors are not the same length?

```
(1:5) + y
```

26. Here the elements of the first vector were recycled (your linear algebra professor would be horrified). When one vector is shorter than the other, the elements of that entire vector get recycled, starting from the first element and getting repeated as often as necessary. Note that if this mechanism does not use a complete cycle, you'll get a warning.

```
(1:4) + y
```

27. Finally, note that using a single value (i.e., a scalar) is just a special case of recycling the same value over and over.

```
y * 2
```

Exercise:

a. `0:10 / 10` yields the same result as `seq(from = 0, to = 1, by = 0.1)`. Can you understand why? Which do you think is more efficient?

b. Can you predict what this command does?

```
10 ^ (0:5)
```

28. R supports sorting, using the `sort()` and `order()` functions.

```
sort(state.area)               # sorts the areas of the states from smallest to largest
order(state.area)              # returns a vector of the positions of the sorted elements
state.name[order(state.area)]  # sort the state names by state size
state.name[order(state.area, decreasing = TRUE)]
                               # sort the state names by state size
```


29. We can also randomly sample elements from a vector, using `sample()`.

```
sample(state.name, 4)              # randomly picks four states
sample(state.name)                 # randomly permute the entire vector of state names
sample(state.name, replace = TRUE) # selection with replacement
```

    This is frequently used in bootstrapping techniques.

30. Other miscellaneous useful commands on vectors include

```
rev(x)    # reverses the vector
sum(x)    # sums all the elements in a numeric or logical vector
cumsum(x) # returns a vector of cumulative sums (or a running total)
diff(x)   # returns a vector of differences between adjacent elements
max(x)    # returns the largest element
min(x)    # returns the smallest element
range(x)  # returns a vector of the smallest and largest elements
mean(x)   # returns the arithmetic mean
```

Summary: Vector elements are accessed using indexing vectors, which can be numeric, character or logical vectors.

Summary: List of logical expression functions:
`<` `>` `<=` `>=` `!=` `==` `%in%`

Summary: Methods of generating regular vectors:

1. Numeric vector, from scratch, shortcut:
   `from:to`
2. Numeric vector, from scratch:
   `seq(from, to, by, length.out, along.with)`
3. Any type of vector, derived from an existing one (x):
   `rep(x, times, length.out, each)`

3.3 Factors

Factors are similar to vectors, but they have another tier of information. A factor keeps track of all the distinct values in that vector, and notes the positions in the vector where each distinct value can be found. Factors are R's preferred way of storing categorical data.

The set of distinct values are called levels. To see (and set) the levels of a factor, you can use the `levels()` function, which will return the levels as a vector.

1. R has an example factor built in:

```
state.division
levels(state.division)
```

2. To get a hint about how R stores factors (or any other object), we can use the `str()` function to view the structure of that object. You can also use the `class()` function to learn the class of an object, without having to see all the details.

```
str(state.division)
class(state.division)
```

Note the list of integers corresponds to the level at each position. While factors may behave like character vectors in many ways, they are much more efficient because they are internally represented as integers and computers are good at working with integers.

3. You can convert a vector to a factor using the `factor()` function. Let's wish for some ponies.

```
pony.colors <- sample(colors, size = 500, replace = TRUE)
str(pony.colors)
```

Note that we are storing each color as a character string. This is not ideal. Let's convert this vector to a factor.

```
pony.colors.f <- factor(pony.colors)
str(pony.colors.f)
```

4. You can plot a factor to see how frequently each level appears.

```
plot(pony.colors.f)
```

The levels are plotted in the order they are returned by `levels()`. But you can control the order of the levels when you create the factor.

```
pony.colors.f <- factor(pony.colors, levels=colors)
str(pony.colors.f)
plot(pony.colors.f)
```

5. You can make a factor from a factor, reordering its levels as you go.

```
plot(state.division)
state.division <- factor(state.division, levels = sort(levels(state.division)))
plot(state.division)
```

6. You can rename the levels in a factor by assignment to its `levels()`. This only changes the labels, not the underlying integer representation. In this case, the labels we have are quite long; let's abbreviate them.

```
levels(state.division)
levels(state.division) <- c("ENC", "ESC", "MA", "MT", "NE", "PAC", "SA", "WNC", "WSC")
plot(state.division)
```

7. In most cases, you can treat a factor as a character vector, and R will do the appropriate conversions. Here we list the states in the North East, and then compare the sizes of various groups of states.

```
state.name[state.division == "NE"]
mean(state.area[state.division == "NE"]) / mean(state.area[state.division == "WSC"])
t.test(state.area[state.division == "SA"], state.area[state.division == "MT"])
```

