R Tutorial 1

William Bell

2018-11-28

We are going to cover the basics today of using the R language. This means the calculator, comparison, assignment, types of vectors, subsetting, and writing functions for R. Don't worry if you don't recognize all of those words, that is part of what we'll be learning about!

R as Calculator

The simplest thing that R can do is act like any calculator does:

```
2+2
```

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

```
6 %% 2
```

```
## [1] 0
```

```
0:10+3
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12 13
```

```
0:10-3
```

```
##  [1] -3 -2 -1  0  1  2  3  4  5  6  7
```

```
0:10*3
```

```
##  [1]  0  3  6  9 12 15 18 21 24 27 30
```

```
0:10/3
```

```
##  [1] 0.0000000 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000
##  [8] 2.3333333 2.6666667 3.0000000 3.3333333
```

```
0:10%%3
```

```
##  [1] 0 1 2 0 1 2 0 1 2 0 1
```

```
0:10%/%2
```

```
##  [1] 0 0 1 1 2 2 3 3 4 4 5
```

The first four probably look familiar, addition, subtraction, multiplication, and divison are very ordinary operations. However on top of these more common operations, there are two more, **modulo** and **integer division**. For those who remember long division (this would be a good time to remind yourself about that!), modulo gives the **remainder** of the first term divided by the second, and integer division gives the **quotient** of the first term divided by the second.

These operations are very important and you will find many surprising uses for instance for modulo and integer division.

One useful feature of R which many other languages do not have is **vectorized operations**, that is, when I add 2 to (2, 3, 4, 51), I add 2 to each of them. Similarly I can add (2, 1, 5, 6) to (2, 3, 4, 51), and it will add together the first, second, third, etc terms.

Try that for yourself in your console:

```
c(2, 1, 5, 6) + c(2, 3, 4, 51)
```

Think about what will happen in the next two cases, and then see what happens:

```
c(1, 1, 1, 1) + c(1, 2)

c(1, 1, 1) + c(3, 1)
```

Fool around with this as long as you need, but before we finish I will point out one last thing. We introduced two ways here of making a vector: `c()` combines all the elements given into one vector and the `:` function makes a vector of integers going from the integer on the left to the integer on the right. You can learn more about these with the following help commands:

```
?'c'

?':'
```

These functions are useful for creating vectors that you can play with.

Comparison (Boolean Expressions) and Assignment

At some point, you're going to want to check if something is bigger than something else, or equal to another thing, or some other sort of comparison. When you do, you're performing a comparison. R offers the following basic comparison operators:

```
2 == 2
```

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

```
3 == 2
```

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

```
2 < 3
```

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

```
2 <= 3
```

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

```
2 <= 2
```

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

```
4 > 2
```

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

```
2 != 2
```

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

```
1 != 0
```

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

```
1:10 > 5
```

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

If you look at these, most of them will look familiar from highschool algebra (greater than, greater than or equal to, equal to, not equal to). As you can see, comparison is also vectorized.

You might wonder, why is it two equals signs instead of one? Let's try that:

```
2 = 3
```

```
## Error in 2 = 3: invalid (do_set) left-hand side to assignment
```

```
3 = 3
```

```
## Error in 3 = 3: invalid (do_set) left-hand side to assignment
```

This error message is quite confusing if you weren't expecting it. But read it carefully, you're being told you can't assign something to the left-hand side, the 2, assignment is when you give something a name, and that starts to make sense of it. You're trying to say that 2 is what we should call 3, which the language doesn't like.

If instead we do:

```
a = 3

a
```

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

We create an object, named a, which is the vector of length one with 3 as its only element. There are two basic ways of doing (local) assignment in R (and numerous sillier ways):

```
a = 3

a <- 3
```

These all do the same thing. That doesn't stop people from thinking one or the other is better, and whereas = is more frequently used in other languages and so possibly good practice if you plan to use other languages, it also is one fewer characters, <- is the preference of most serious R users (and once you use it for awhile = looks ugly). No need to get caught up in these religious wars between programmers, but now you know! I will use <- since I like it and nobody can stop me!

In order to learn more about assignment or comparison, check their respective help pages with the following commands:

```
?'<'

?'='
```

This covers assignment but there is more to be said for comparison. Sometimes you don't want to just check one comparison, you might want to check more than one at once, or you might want to check than one of two conditions is satisfied, or a more complex combination. This requires the other Boolean operators:

```
!(1:10 == 4)
```

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

```
(1:10 == 4) | (2:11 > 7)
```

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

```
c(TRUE, FALSE, TRUE, FALSE) & c(FALSE, TRUE, TRUE, TRUE)
```

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

This introduces three new logical operators, '!' can be read as 'not'. So if you pass it a logical vector, it will change every true to a false and every false to a true. '&' takes in two logical vectors, and unless there is a true and a true on both sides, then it will output false for a given spot. '|' can be read as 'or' (or for the more pedantic, inclusive or). The 'or' takes in values, and outputs true if and only if at least one value is true for a given spot.

Types of Vector

We've already seen three kinds of vector, but maybe you didn't notice.

```
b <- 0:10

c <- c(0, 1.2894893843838439, 2, 3, 4, 5, 6, 7, 8, 9, 10)

d <- c(TRUE, FALSE, T, F)
```

We can see the types of each of these vectors with the `typeof` function:

```
typeof(b)
```

```
## [1] "integer"
```

```
typeof(c)
```

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

```
typeof(d)
```

```
## [1] "logical"
```

The first vector is an **integer** vector, which if you remember your highschool mathematics, is a whole number, negative or positive or zero (. . .-2, -1, 0, 1, 2, . . .). The computer can't give you every integer, which we'll return to below. The second is called a **double** vector, a **double** vector is composed of numbers that can be whole or not, i.e. numbers with decimal places. But be wary, doubles are accurate to only a finite number of decimal places, and this can cause havoc for certain operations:

```
10000 + 0.001
```

```
## [1] 10000
```

Usually it only matters if you're interested in **very** small numbers, or differences between big numbers, so for instance:

```
10000000000 + 1000
```

```
## [1] 1e+10
```

(This would be a good time to refamiliarize yourself with scientific notation)

Is certainly not good, neither is:

```
1/(10^308)
```

```
## [1] 1e-308
```

```
1/(10^309)
```

```
## [1] 0
```

```
1e-320
```

```
## [1] 9.999889e-321
```

The second case is called a **numeric underflow**, which means the number is too small for double arithmetic to distinguish it from zero. In the previous code chunk where we added together a really big number and a fairly big number, and the third line in this chunk where instead of 1 x 10^{-320} it gives you a number slightly smaller than that, it is simply trying to find the closest number it knows.

The opposite of a numeric underflow is **numeric overflow**, which applies to both doubles and integers:

```
1e1000
```

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

```
1e308
```

```
## [1] 1e+308
```

```
1e154*1e155
```

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

When a number gets sufficiently big in magnitude, we call it `Inf` (for infinity) or `-Inf` depending, shockingly, on whether it is positive or negative.

If you're familiar with Python, this might sound familiar from Floating Point Numbers, the difference is that whereas Floating Point Numbers are guaranteed accurate to roughly 16 decimal places (before addition, subtraction which of course increase the error), Double Numbers are accurate to 32 decimal places. For most applications the difference doesn't matter.

The final vector type we've already looked at is the logical vector:

```
d
```

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

A logical vector is a vector of trues and falses, usually created by a comparison or recorded in data. We will see more uses for logical vectors when we cover subsetting, but one useful thing to know about logical vectors is that underneath the surface, they're secretly integers:

```
1*d
```

```
## [1] 1 0 1 0
```

```
2*d
```

```
## [1] 2 0 2 0
```

The trues are 1's and the falses are 0's. This comes in handy for instance, if you want to know how many responses are 'True' in a survey:

```
sum(d)
```

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

Or the proportion of responses that are 'true':

```
mean(d)
```

```
## [1] 0.5
```

Think about why it would be the case that the sum of a logical vector where True = 1 would be the number of Trues in that vector, and why the mean would be the proportion of Trues in the vector.

Before we continue, I would like to introduce one more basic type of vector, there are others but this is good enough for our purposes. The last type is a **character** vector:

```
e <- c("This", 'is', 'a', "charactR", "vector.") ## No difference between quotes and apostrophes,
## just don't mix them though.

e
```

```
## [1] "This"     "is"       "a"        "charactR" "vector."
```

```
typeof(e)
```

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

This holds information in the form of text. For instance, if you have a short answer question on a survey, you might record it in a spreadsheet, and then try to examine the text for patterns in R.

You can also check if a vector is a certain kind of vector with several different comparison functions:

```
is.double(a)
```

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

```
is.double(b)
```

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

```
is.double(c)
```

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

```
is.double(d)
```

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

```
is.double(e)
```

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

```
is.integer(b)
```

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

```
is.logical(d)
```

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

```
is.character(e)
```

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

There are also some categorical ones, for instance, if something is a double, or integer vector (along with a couple types not yet considered - not all of them predictable!), then when we use the function `is.numeric` it should output true:

```
is.numeric(a)
```

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

```
is.numeric(b)
```

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

There are also functions to convert between types:

```
d
```

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

```
as.integer(d)
```

```
## [1] 1 0 1 0
```

```
c
```

```
##  [1]  0.000000  1.289489  2.000000  3.000000  4.000000  5.000000  6.000000
##  [8]  7.000000  8.000000  9.000000 10.000000
```

```
as.integer(c)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

We see that sometimes some information is lost in these transitions, such as from double to integer we lop off all the extra decimal places. When we change something from one type to another, it is called **coercion**, many functions do it automatically (e.g. arithmetical operations convert everything to double unless you're working with two integer vectors).

Aside from vectors, we'll have to contend with the three other basic kinds of data structure in R: matrices, lists, and data frames. The final one on this list is perhaps one of the most lasting contributions of R, so much so that it has been adopted in other languages more or less by copy-paste (such as the Pandas package in Python). We will use lists when we use the strsplit function later, and the twitter information is stored as a data frame.

Next we must address subsetting:

Subsetting

Subsetting is one of the most basic tasks while working in a programming language, what it entails is taking the elements you want out of a vector. So for instance:

```
b
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Suppose I want the zero at the beginning of this vector, since it is the first element of the vector, I can pick it out as follows:

```
b[1]
```

```
## [1] 0
```

If you've already written in another language like Python, you may have encountered indexing starting with zero instead of one. If you prefer indexing from zero, you can keep it to yourself (enter the next reglious war between programmers - indexing from zero vs indexing from one!), but everything in R is indexed from 1.

We can see how this works fairly easily with more examples:

```
d[2]
```

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

```
b[c(5, 2)]
```

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

```
3:5
```

```
## [1] 3 4 5
```

```
e[3:5]
```

```
## [1] "a"        "charactR" "vector."
```

```
5:3
```

```
## [1] 5 4 3
```

```
e[5:3]
```

```
## [1] "vector."  "charactR" "a"
```

Subsetting returns a new vector starting from 1 again. What we see is that the positions taken correspond to the numbers given and the vector returned is in the order of the numbers given. This will become more complicated as we consider lists and matrices, which each have two types of subsetting.

One thing you might want to do is instead of picking out a particular number you want, you can subset according to some condition. This requires you to use comparison:

```
c
```

```
##  [1]  0.000000  1.289489  2.000000  3.000000  4.000000  5.000000  6.000000
##  [8]  7.000000  8.000000  9.000000 10.000000
```

```
wholeNumbers <- (c %% 1) == 0

wholeNumbers
```

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

```
c[wholeNumbers]
```

```
##  [1]  0  2  3  4  5  6  7  8  9 10
```

What we've done is created a logical vector, and then we put in the logical vector to the subsetting function and we're returned every element of our vector corresponding to a true in the logical vector. Here are a few more cases for you to run on your own:

```
c[c > 5]

f <- 5:15

c[f > 5]

g <- f > 5

c[g]
```

See if you can guess what they'll do ahead of time.

Logical subsetting is one of the most useful operations in your vocabulary, often a single carefully thought out logical statement can get you exactly the data you want out of a big wad of data with some thought.

Functions

Doing data analsis involves performing a lot of complex operations. It is tempting to expend too much effort on it. For instance, consider the following ways of calculating the sample mean of a vector (it is not important to understand all the details just yet):

```
set.seed(1)

x <- runif(1000)

x[1:50]
```

```
##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
##  [7] 0.94467527 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675
## [13] 0.68702285 0.38410372 0.76984142 0.49769924 0.71761851 0.99190609
## [19] 0.38003518 0.77744522 0.93470523 0.21214252 0.65167377 0.12555510
## [25] 0.26722067 0.38611409 0.01339033 0.38238796 0.86969085 0.34034900
## [31] 0.48208012 0.59956583 0.49354131 0.18621760 0.82737332 0.66846674
## [37] 0.79423986 0.10794363 0.72371095 0.41127443 0.82094629 0.64706019
## [43] 0.78293276 0.55303631 0.52971958 0.78935623 0.02333120 0.47723007
## [49] 0.73231374 0.69273156
```

```
## Method #1

sumX <- 0

for (i in 1:length(x)) {
  sumX <- sumX + x[i]
}

meanXmethod1 <- sumX/length(x)

meanXmethod1
```

```
## [1] 0.4996917
```

```
## Method #2

meanXmethod2 <- mean(x)

meanXmethod2
```

```
## [1] 0.4996917
```

These methods both work, but one took five lines and the other took one line. Generally speaking, it is advisable to use functions when they shorten your writing.

But the function you want might not always exist. When it doesn't you need to write your own functions, and we'll cover how to do that here. Suppose we wanted to write our own mean function, then this is how we would do it.

```
Mean <- function(x) {

  sumX <- 0

  for (i in 1:length(x)) {
    sumX <- sumX + x[i]
  }

  meanXmethod1 <- sumX/length(x)

  return(meanXmethod1) ## return statement not strictly necessary but we will discuss
  ## that in another tutorial
}

Mean(x)
```

```
## [1] 0.4996917
```

The first thing we give is the name of the function, `Mean`. Then, how it's written it is sayng that we're assigning something to that name. We're saying it's a `function`, and we're specifying what it takes in `x`. Then inside the curly brackets `{...}` we say what the function is going to do, in this case, it is going to calculate the mean.

And there we are, we've made a function to calculate the mean. That's not particularly useful because the function is doing something that there is a built in function for, but it is a good example to get you used to how function syntax works.

Functions are the bread and butter of R, so much so that there are functions meant to take in functions or output functions. For example, the `apply` family of functions, which are worth getting to know, are made to take in functions, sometimes custom functions, and output a result.

Aside from fancy cases like the `apply` family, it is common wisdom that if you find yourself writing the exact same lines of code more than once, then it might be better to write a function instead so that you can do the same thing in one line next time you need to write it out.

Practice

In order to get to know these basic operations and the way they interact, we can try making a few functions. I'll provide either some code that does some operation and you have to use it in order to build a function, or I'll provide a function, and you'll need to say what it does.

1. This code adds two to every second element of a vector. Design a function that does the same:

```r
length(x)
```

```
## [1] 1000
```

```r
x[(1:500)*2] <- x[(1:500)*2] + 2
```

Then design a function that adds any number to every second element of a vector. Make sure it works for vectors of odd length (Hint: Consider how integer division might help you here).

2. Here we have a line that prints "Hello World!" Make it into a function.

```r
print("Hello World!")
```

```
## [1] "Hello World!"
```

Now look at the help page for `paste` and then make a function that takes in a name and prints "Hello World! My name is <name>".

3. We want to make every tenth spot of a vector be equal to zero. This code does it for one vector, understand why, and show how to do it with a function:

```r
x[((1:1000) %% 10) == 0] <- 0
```

Consider the case of the empty double vector:

```r
a <- numeric()
```

Make sure that your function works correctly on this vector (it should output the empty double vector). Hint: Look at the help page for `seq_along` or `seq_len`.

4. What does this function do? Try to answer using the help page for `rep` and what you know about Boolean operators.

```r
mysteryFunc1 <- function(x) {
  everySecond <- everyThird <- rep(FALSE, length(x))

  everySecond[(1:(length(x) %/% 2))*2] <- TRUE

  everyThird[(1:(length(x)%/%3))*3] <- TRUE

  everySecondOrThird <- everySecond | everyThird

  x[everySecondOrThird] <- 0

  x
}
```

5. What does this function do? What kind of object does it output?

```
mysteryFunc2 <- function(n) {

  function(x) {

    x[((1:length(x)) %% n) == 0] <- 0

    x
  }
}
```

Hint: Try seeing what it outputs in one case.

Conclusion

Next we will consider control flow (`if`, `else`, `else if`, `for`, `while`, and `repeat`), some basic handy functions (`rep`, `seq`, `length`, `mean`, `median`, `sum`, etc), and joins from the `dplyr` package.