R Session #2: Vectors and Using Functions

Contents

1 Companion material

DataCamp Introduction to R, Chapter 3 (Vectors) DataCamp Intermediate R, Chapter 3 (Functions): "Introduction to Functions" through "Required, or optional?"

2 Recap from session #1

After the first session, you should:

1) be familiar with the Source, Console, Environment, and Files panes of Rstudio, and what each is for.
2) know how to send commands from the Source to Console, and how to save your work (code as .R files, and any variables you want to keep for a later session as .Rdata files).

3) know what numeric, character, and logical variables are, and how arithmetic and/or logical operators can be applied to each of them.

Review these if you're unclear on them!

3 What is a vector?

In R, a "vector" is a group of values that are all of the same type. We can combine individual values into a vector with the `c()` (for "combine" or "concatentate") function.

```
cities = c("San Francisco","London","Marrakesh") # a character vector
population = c(8.37e5,8.31e6,9.29e5) # a numeric vector (using scientific notation)
have_visited = c(TRUE,TRUE,FALSE) # a logical vector
```

4 Accessing individual vector elements (aka "indexing")

We need a way to access individual elements within a vector. Each element's position in the vector is called its "index". To retrieve the values at particular indices in a vector, we use brackets `[]`.

```
> cities[2] # 2nd element of the "cities" vector
[1] "London"
> population[c(1,3)] # 1st and 3rd elements of the "population" vector
[1] 837000 929000
```

4.1 Re-assigning individual elements

With indexing, we can also change individual elements in a vector, rather than recreating the whole thing from scratch. Say you won a trip to Marrakesh, so you get to change its value in `have_visted` to `TRUE`.

```
> have_visited # before
[1]  TRUE  TRUE FALSE
> have_visited[3] = TRUE # read: "assign TRUE to the third element of have_visited"
> have_visited # after
[1] TRUE TRUE TRUE
```

4.2 Indexing with logical vectors

We can also use a logical vector as a set indices. As long as the vector you're indexing and the logical vector are the same length, the logical vector acts as what's called a "mask", and only the corresponding TRUE elements are taken.

```
> have_visited
[1] TRUE TRUE TRUE
> # The elements of have_visited that are TRUE are c(1,2), so indexing cities
> # with have_visited is the same as indexing cities with c(1,2).
> cities[c(1,2)]
[1] "San Francisco" "London"
> cities[have_visited]
[1] "San Francisco" "London"        "Marrakesh"
```

Exercise: In one line of code, get the names of any visited cities with population greater than 1 million.

4.3 Named indices

We can refer to a specific element not only by its integer index, but also by a descriptive name, if it has one. Right now, none of our vector has named indices. You can both set and check the names of a vector (or other data structures, as we'll see later) with the `names()` function.

```
names(have_visited) # no names yet
NULL
names(have_visited) = cities
names(have_visited) # now the values of "cities" are indices of "have_visited"
[1] "San Francisco" "London"        "Marrakesh"
have_visited["London"] # and we can use them just like integer indices
London
  TRUE
```

5 Operations on vectors

The same arithmetic and logical operations you've done with single numbers, characters, and logicals will all work with vectors, too.

5.1 Operations between vectors and single values

Easy. The operator is applied between the single value and each element of the vector.

```
> c(1,2,3)*2  # multiply each element of c(1,2,3) by 2
```

```
[1] 2 4 6
```

```
> c(1,2,3)>=2 # evaluate whether each element of c(1,2,3) is greater than or equal to 2
```

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

5.2 Operations between vectors of equal length

Also easy: the operator applies to each pair of corresponding elements between the two vectors.

```
c(2,3,4) + c(10,20,30) # means c(2+3, 3+3, 4+3)
```

```
[1] 12 23 34
```

```
c(2,3,4) < c(10,20,30) # means c(2<3, 3<3, 4<3)
```

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

```r
c(3<4, TRUE) & c(TRUE, "one"==1) # means c(TRUE & TRUE, TRUE & FALSE)
```

```
[1]  TRUE FALSE
```

5.3 Operations between vectors of unequal length (optional)

Here's where things get slightly tricky. The reason that `c(1,2,3)*2` gives `c(2,4,6)` is that the value 2 is "recycled" until it matches the vector `c(1,2,3)`'s length. In other words, R effectively converts `2` to `c(2,2,2)` so that it can do vector addition element-by-element. With unequal vectors, the shorter vector gets recycled until its length matches the longer vector. This leads to slightly different behavior for shorter vectors whose lengths divide evenly into the longer vector's length, and those that don't.

```r
> # These ...
> c(1,2,3,4) + c(1,2)
[1] 2 4 4 6
> c(1,2,3,4) + c(1,2,3)
Warning in c(1, 2, 3, 4) + c(1, 2, 3): longer object length is not a
multiple of shorter object length
[1] 2 4 6 5
>
> # are equivalent to these ...
> c(1,2,3,4) + c(1,2,1,2) # c(1,2) gets recycled twice to have length 4
[1] 2 4 4 6
> c(1,2,3,4) + c(1,2,3,1) # c(1,2,3) starts to be recycled, but stops at length 4
[1] 2 4 6 5
```

We get a warning in the second case because 3 does not divide evenly into 4, which means the shorter vector's recycling gets cut short. This is a common error message, and it might be no big deal, or (more likely) it might mean that the vectors you're trying to operate on aren't what you think they are!

6 The `%in%` operator for logical vectors

The basic logical operators like `>` and `&` work element-wise on vectors, as we've just seen, but there's another (and very useful) logical operator that works slightly differently. Instead of operating on each corresponding *pair* of elements in each vector, it operates on *all* combinations of elements by checking each element of vector1 to see if its value is found anywhere in vector2.

```r
c(1,3) %in% c(5,3,2)
```

```
[1] FALSE  TRUE
```

```r
# 1 is NOT in the vector c(5,3,2)
# 3 IS in the vector c(5,3,2)
```

Logically, this is equivalent to:

```r
1==5 | 1==3 | 1==2
```

```
[1] FALSE
```

```
3==5 | 3==2 | 3==2
```

```
[1] FALSE
```

7 Functions for creating vectors with repeating patterns

Very often you'll need to create a vector with some repeating pattern, like a vector with a thousand zeroes, or a sequence from 1 to 100. It would be most inconvenient if we had to type out `myVec = c(0,0,0, ...,0)` or `myVec=c(1,2,3, ..., 100)`. Two functions you'll frequenetly for this are `rep()` (for "repeat" or "replicate") and `seq()` ("sequence").

```
> seq(1,10)  # sequence from 1 to 10, in steps of 1 (the default)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,10, by=2)  # sequence from 1 to 10, in steps of 2
[1] 1 3 5 7 9
> seq(1,10, length=5) # sequence of 5 evenly-spaced values between 1 and 10
[1]  1.00  3.25  5.50  7.75 10.00
> 1:10 # shorthand for seq(1,10); assumes integer sequences with steps of 1
 [1]  1  2  3  4  5  6  7  8  9 10
> rep(1,10)  # repeat the value 1, 10 times
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(1:4, times=3)  # repeat the whole vector 1:4 three times
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
> rep(1:4, each=3) # repeat each successive element of 1:4 three times
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

Note: While convenient, you need to be careful when using `:` to create simple sequences. If either side of the `:` is an expression needing evaluation, be sure to wrap it in parentheses.

```
1:4-1 # First forms the vector 1:4, then subtracts 1 from every element
[1] 0 1 2 3
1:(4-1) # Maybe this is what you wanted instead?
[1] 1 2 3
```

8 Function arguments (and using R's help section)

Since we're using functions to create vectors, this is a good time to talk more generally about functions' arguments (i.e. their inputs, the things inside the parentheses). For example, when we execute the command `rep(1:4, times=3)`, we say that the function `rep()` has been "called" with arguments `1:4` and `times=3`. Arguments come in two flavors:

- *required* (ones you have to supply, otherwise the function won't work)
- *optional* (further arguments that control more specfic behavior, but that already have some default value in case you don't supply one)

In this case, `1:4` (the thing to be replicated) is essential, and `times=3` (how to do the replicating) is optional. It's optional because the function already has defaults that tell it what to do if you don't give it an `each` or `times` argument. Look up the help for the `rep()` function by typing `?rep` in the Console (or, by going to the "Help" tab and typing `rep` in the search box at the upper right).

In the help page under “Usage” you’ll see that to use `rep()`, its first, required argument is a vector (which can be just a single value; a single value is essentially a one-element vector), followed by `...`. The ellipses refer to other possible arguments. Further down you’ll see that these can include `times`, `length.out` (which we don’t care about right now), and `each`. Each argument’s role is described, and further down under “Details”, it explains that the default value of `times` and `each` is 1.

Exercise: Look up the help for the `seq()` function and explain why each of these lines does what it does.

1) `seq(5)`
2) `seq(from=5)`
3) `seq(to=5)`
4) `seq(to=1,from=5)`
5) `seq(5, by=-1)`

Some of these might seem counter intuitive. The moral of the story is that relying on default behavior can sometimes give you unexpected results. **If a function has optional arguments, make sure you either clearly understand what they are and do, or specify them explicitly!**

8.1 Argument order

If you don’t use the names of the arguments, then they will be taken in the order they appear in the help page: `seq(10,1)` means `seq(from=10,to=1)`, because `from` comes before `to` in the argument list. However, if you explicitly use the names, then you may put arguments in whatever order you want: `seq(to=1, from=10)` is exactly the same as `seq(from=10, to=1)`.

9 Helpful functions for inspecting vectors

The following functions are among the most-used in R’s library. Use them early and often! You can ward off many bugs and headaches by finding out sooner rather than later that a vector (or other data structure) doesn’t have the structure you assumed it did.

```
> x = population
> which(population < 1e6) # which elements are TRUE (applies only to logical vectors)
[1] 1 3
> length(x) # the length of the vector (i.e. how many elements it has)
[1] 3
> class(x) # what variable type (e.g. "numeric"")
[1] "numeric"
> str(x) # prints helpful information, including the type, length, and first few values
 num [1:3] 837000 8310000 929000
> any(have_visited) # are any elements TRUE?
[1] TRUE
> all(have_visited) # are all elements TRUE?
[1] TRUE
> sum(have_visited) # how many elements are TRUE? Know why this works!
[1] 3
```

Exercises: Some of these might seem difficult to do in one fell swoop, so break them up into smaller pieces!

1) Generate a vector of 50 random values, `x` (or whatever else you want to call it), using the function `rnorm()`. Look up the help file with the command `?rnorm`.

2) Use indexing and `seq()` to create a vector containing only the first 15 and last 15 elements of `x`. Then usse `head(x)` and `tail(x)` to confirm your answer. You'll have to look up the help on these functions to see how to make them give you the desired number of elements.
3) Make a duplicate of `x` (call it `x2`, or `y`, or anything that makes sense to you), and multiply any values greater than 2 or less than -2 by 1.1 (hint: while not necessary to solve this problem, take a look at the `abs()` function).
4) Make a duplicate of `x`, and replace every third element with its predecessor (i.e. the value of `x[3]` becomes equal to the value of `x[2]`, `x[6]` equal to `x[5]`, etc.). What are some ways to make sure you got it right?