Subsetting Vectors

In this lesson, we'll see how to extract elements from a vector based on some conditions that we specify. In other words, we want to select some of the numbers in a vector based either on their position in the vector or the value that each number has.

For example, we may only be interested in the first 20 elements of a vector, or only the elements that are not NA, or only those that are positive or correspond to a specific variable of interest. By the end of this lesson, you'll know how to handle each of these scenarios.

I've created for you a vector called x that contains a random ordering of 20 numbers (from a standard normal distribution) and 20 NAs. Type x now to see what it looks like.

```
x
```

```
##  [1]          NA          NA          NA  0.21137604 -0.35540395
##  [6]  1.05133675 -0.29708595          NA  0.16522869 -0.08953262
## [11] -0.94205430          NA          NA          NA -2.31152481
## [16]          NA  2.17574587          NA          NA          NA
## [21] -0.89579353  0.12405336          NA -0.82601445 -1.82833011
## [26]          NA -0.13039136 -0.71400891          NA -0.93170517
## [31] -1.05010940          NA          NA          NA -0.02434250
## [36] -1.19228099          NA          NA  0.49693406          NA
```

The way you tell R that you want to select some particular elements (i.e., a 'subset') from a vector is by placing an 'index vector' in square brackets immediately following the name of the vector.

For a simple example, try x[1:10] to view the first ten elements of x.

```
x[1:10]
```

```
##  [1]          NA          NA          NA  0.21137604 -0.35540395
##  [6]  1.05133675 -0.29708595          NA  0.16522869 -0.08953262
```

Index vectors come in four different flavors – logical vectors, vectors of positive integers, vectors of negative integers, and vectors of character strings – each of which we'll cover in this lesson.

Let's start by indexing with logical vectors. One common scenario when working with real-world data is that we want to extract all elements of a vector that are not NA (i.e., missing data). Recall that is.na(x) yields a vector of logical values the same length as x, with TRUEs corresponding to NA values in x and FALSEs corresponding to non-NA values in x.

What do you think x[is.na(x)] will give you?

1. A vector of all NAs
2. A vector with no NAs
3. A vector of TRUEs and FALSEs
4. A vector of length 0

A vector of all NAs

Prove it to yourself by typing x[is.na(x)].

```
x[is.na(x)]
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Recall that ! gives us the negation of a logical expression, so !is.na(x) can be read as 'is not NA'. Therefore, if we want to create a vector called y that contains all of the non-NA values from x, we can use y <- x[!is.na(x)]. Give it a try.

```
y <- x[!is.na(x)]
```

Print y to the console.

```
y
```

```
##  [1]  0.21137604 -0.35540395  1.05133675 -0.29708595  0.16522869
##  [6] -0.08953262 -0.94205430 -2.31152481  2.17574587 -0.89579353
## [11]  0.12405336 -0.82601445 -1.82833011 -0.13039136 -0.71400891
## [16] -0.93170517 -1.05010940 -0.02434250 -1.19228099  0.49693406
```

Now that we've isolated the non-missing values of x and put them in y, we can subset y as we please.

Recall that the expression y > 0 will give us a vector of logical values the same length as y, with TRUEs corresponding to values of y that are greater than zero and FALSEs corresponding to values of y that are less than or equal to zero. What do you think y[y > 0] will give you?

1. A vector of all the positive elements of y
2. A vector of all the negative elements of y
3. A vector of all NAs
4. A vector of length 0
5. A vector of TRUEs and FALSEs

A vector of all the positive elements of y

Type y[y > 0] to see that we get all of the positive elements of y, which are also the positive elements of our original vector x.

```
y[y > 0]
```

```
## [1] 0.2113760 1.0513367 0.1652287 2.1757459 0.1240534 0.4969341
```

You might wonder why we didn't just start with x[x > 0] to isolate the positive elements of x. Try that now to see why.

```
x[x > 0]
```

```
##  [1]        NA        NA        NA 0.2113760 1.0513367        NA 0.1652287
##  [8]        NA        NA        NA        NA 2.1757459        NA        NA
## [15]        NA 0.1240534        NA        NA        NA        NA        NA
## [22]        NA        NA        NA 0.4969341        NA
```

Since NA is not a value, but rather a placeholder for an unknown quantity, the expression NA > 0 evaluates to NA. Hence we get a bunch of NAs mixed in with our positive numbers when we do this.

Combining our knowledge of logical operators with our new knowledge of subsetting, we could do this – x[!is.na(x) & x > 0]. Try it out.

```
x[!is.na(x) & x > 0]
```

```
## [1] 0.2113760 1.0513367 0.1652287 2.1757459 0.1240534 0.4969341
```

In this case, we request only values of x that are both non-missing AND greater than zero.

I've already shown you how to subset just the first ten values of x using x[1:10]. In this case, we're providing a vector of positive integers inside of the square brackets, which tells R to return only the elements of x numbered 1 through 10.

Many programming languages use what's called 'zero-based indexing', which means that the first element of a vector is considered element 0. R uses 'one-based indexing', which (you guessed it!) means the first element of a vector is considered element 1.

Can you figure out how we'd subset the 3rd, 5th, and 7th elements of x? Hint – Use the c() function to specify the element numbers as a numeric vector.

```
x[c(3, 5, 7)]
```

```
## [1]          NA -0.3554039 -0.2970860
```

It's important that when using integer vectors to subset our vector x, we stick with the set of indexes {1, 2, …, 40} since x only has 40 elements. What happens if we ask for the zeroth element of x (i.e. x[0])? Give it a try.

```
x[0]
```

```
## numeric(0)
```

As you might expect, we get nothing useful. Unfortunately, R doesn't prevent us from doing this. What if we ask for the 3000th element of x? Try it out.

```
x[3000]
```

```
## [1] NA
```

Again, nothing useful, but R doesn't prevent us from asking for it. This should be a cautionary tale. You should always make sure that what you are asking for is within the bounds of the vector you're working with.

What if we're interested in all elements of x EXCEPT the 2nd and 10th? It would be pretty tedious to construct a vector containing all numbers 1 through 40 EXCEPT 2 and 10.

Luckily, R accepts negative integer indexes. Whereas x[c(2, 10)] gives us ONLY the 2nd and 10th elements of x, x[c(-2, -10)] gives us all elements of x EXCEPT for the 2nd and 10 elements. Try x[c(-2, -10)] now to see this.

```
x[c(-2, -10)]
```

```
##  [1]          NA          NA  0.2113760 -0.3554039  1.0513367 -0.2970860
##  [7]          NA  0.1652287 -0.9420543          NA          NA          NA
## [13] -2.3115248          NA  2.1757459          NA          NA          NA
## [19] -0.8957935  0.1240534          NA -0.8260144 -1.8283301          NA
## [25] -0.1303914 -0.7140089          NA -0.9317052 -1.0501094          NA
## [31]          NA          NA -0.0243425 -1.1922810          NA          NA
## [37]  0.4969341          NA
```

A shorthand way of specifying multiple negative numbers is to put the negative sign out in front of the vector of positive numbers. Type x[-c(2, 10)] to get the exact same result.

```
x[-c(2, 10)]
```

```
##  [1]          NA          NA  0.2113760 -0.3554039  1.0513367 -0.2970860
```

```
##  [7]         NA  0.1652287 -0.9420543         NA         NA         NA
## [13] -2.3115248         NA  2.1757459         NA         NA         NA
## [19] -0.8957935  0.1240534         NA -0.8260144 -1.8283301         NA
## [25] -0.1303914 -0.7140089         NA -0.9317052 -1.0501094         NA
## [31]         NA         NA -0.0243425 -1.1922810         NA         NA
## [37]  0.4969341         NA
```

So far, we've covered three types of index vectors – logical, positive integer, and negative integer. The only remaining type requires us to introduce the concept of 'named' elements.

Create a numeric vector with three named elements using vect <- c(foo = 11, bar = 2, norf = NA).

```
vect <- c(foo = 11, bar = 2, norf = NA)
```

When we print vect to the console, you'll see that each element has a name. Try it out.

```
vect
```

```
##  foo  bar norf
##   11    2   NA
```

We can also get the names of vect by passing vect as an argument to the names() function. Give that a try.

```
names(vect)
```

```
## [1] "foo"  "bar"  "norf"
```

Alternatively, we can create an unnamed vector vect2 with c(11, 2, NA). Do that now.

```
vect2 <- c(11, 2, NA)
```

Then, we can add the `names` attribute to vect2 after the fact with names(vect2) <- c("foo", "bar", "norf"). Go ahead.

```
names(vect2) <- c("foo", "bar", "norf")
```

Now, let's check that vect and vect2 are the same by passing them as arguments to the identical() function.

```
identical(vect, vect2)
```

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

Indeed, vect and vect2 are identical named vectors.

Now, back to the matter of subsetting a vector by named elements. Which of the following commands do you think would give us the second element of vect?

1. vect["bar"]
2. vect[bar]
3. vect["2"]

vect["bar"]

Now, try it out.

```
vect["bar"]
```

```
## bar
##   2
```

Likewise, we can specify a vector of names with vect[c(“foo”, “bar”)]. Try it out.

```
vect[c("foo", "bar")]
```

```
## foo bar
##  11   2
```

Actually, you can also use integer indexes on named vectors. First, notice that one of the options in the question above was vect[“2”]. Does this work? Try it out.

```
vect["2"]
```

```
## <NA>
##   NA
```

As above, integer indexes need to be integers. Putting the number 2 in quotes tells R that it is in fact a character.

So, how would you index the second element of vect using an integer?

```
vect[2]
```

```
## bar
##   2
```

BEST PRACTICE: If you have names, it is best to use them. This issue will become more important when we start working with data frame and lists. The problem is that if the columns in a dataframe get out of order, column 2 may no longer be what it was. Using a name ensures that R always gets the correct thing you are interested in.

Yay!! Now you know all four methods of subsetting data from vectors. Different approaches are best in different scenarios and when in doubt, try it out!