Week 2: Vectors

Ellen Bledsoe

2024-01-23

1-Dimensional Data: Vectors

We can also assign more complex group of elements of the same type to a particular object. This is called a **vector**, a basic data structure in R.

A vector is a sequence of values that are all the same data type. We can create them using the `c()` function, which stands for "combine".

```
states <- c("AZ", "AZ", "NM", "CA")
states
```

```
## [1] "AZ" "AZ" "NM" "CA"
```

Using the `str` function we learned last time shows that this is a vector of 4 character strings. The output of the `str` function is essentially the same as what is shown in the environment tab.

```
str(states)
```

```
##  chr [1:4] "AZ" "AZ" "NM" "CA"
```

Sub-setting

Sometimes we want to pull out and work with specific values from a vector. This is called sub-setting (taking a smaller set of the original).

We do this with square brackets: `[]`. In general, square brackets in R means "give me part of something."

Sub-setting by Index

One way that we can subset in R is by index, meaning based on the position of the value(s) we want to pull out.

```
# first value
states[1]
```

```
## [1] "AZ"
```

```r
# first 3 values
states[1:3] # one through three
```

```
## [1] "AZ" "AZ" "NM"
```

Using the colon (1:3) is shorthand for making a vector of the whole numbers 1 through 3. This means that `states[1:3]` is the same as `states[c(1, 2, 3)]`.

You can use a vector to get any subset or order you want `states[c(4, 1, 3)]`.

Many functions in R take a vector as input and return a value. This includes the function `length`, which determines how many items are in a vector.

```r
length(states)
```

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

We can also calculate common summary statistics. For example, if we have a vector of population counts:

```r
# create count vector
count <- c(9, 16, 3, 10)

# summary statistics
mean(count)
```

```
## [1] 9.5
```

```r
max(count)
```

```
## [1] 16
```

```r
min(count)
```

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

```r
sum(count)
```

```
## [1] 38
```

Let's Practice

Practice working with vectors in Question 6 of the Assignment.

Null Values

So far, we've worked with vectors that contain no missing values. In most real world data, however, there are often values that are missing for a variety of reasons.

For example, kangaroo rats don't like being caught by humans and are pretty good at escaping before you've finished measuring them (speaking from personal experience here!).

Missing values, known as "null" values, are written in R as `NA` with no quotes, which is short for "not available". A vector of 4 population counts with the third value missing would look like

```
count_na <- c(9, 16, NA, 10)
```

If we try to take the mean of this vector, we get `NA`. This is pretty weird!

```
mean(count_na)
```

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

This happens with most calculations when `NA` is in the data. Thankfully, we can tell many functions to remove the `NA` before calculating.

We do this using an optional argument, which is an argument that we don't have to include unless we want to modify the default behavior of the function. We add optional arguments by providing their name (`na.rm`), `=`, and the value that we want those arguments to take (`TRUE`).

```
mean(count_na, na.rm = TRUE)
```

```
## [1] 11.66667
```

Let's Practice

Work on Question 7.

Working with Multiple Vectors

Let's build on example where we have information on states and population counts by areas, all stored in vectors.

```
states <- c("AZ", "AZ", "NM", "CA")
count <- c(9, 16, 3, 10)
area <- c(3, 5, 1.9, 2.7)
```

Vector Math

We can perform the same mathematical operation on each value in a vector by treating it like we would a single value. Let's say we wanted to double all of the values in the `area` vector.

```
area * 2
```

```
## [1]  6.0 10.0  3.8  5.4
```

When do this multiplication, R multiplies the first value in the vector by 2, then multiplies the second values in the vector by 2, and so on. This is called *element-wise*, operating on one element of the vector at a time.

Remember, this isn't saved unless we store it. For now, `area` hasn't changed.

```
area
```

```
## [1] 3.0 5.0 1.9 2.7
```

If we want to keep the results of the calculation in a new variable, we use the assignment operator.

```
doubled_area <- area * 2
doubled_area
```

```
## [1]  6.0 10.0  3.8  5.4
```

We can also do element-wise math with multiple vectors of the same length. Let's divide the count vector by the area vector to get a vector of the density of individuals in that area.

```
density <- count / area
```

When we divide the two vectors, R divides the first value in the first vector by the first value in the second vector, then divides the second values in each vector, and so on.

Conditional Sub-setting

Sometimes we want to subset vectors based on whether a condition is being met rather than based on position or index.

We can still do this with the square brackets: `[]`.

Let's pull the density values for sites in Arizona.

When setting a condition, we need to use `==`. This is how we indicate "equal to" in most programming languages. I often think of this as meaning "if and only if."

```
density[states == 'AZ']
```

```
## [1] 3.0 3.2
```

We can also do the opposite, setting the condition as "not equal to."

```
density[states != 'AZ']
```

```
## [1] 1.578947 3.703704
```

We can also set numerical conditional statements like greater or less than.

Let's select states that meet with some restrictions on density

```
states[density > 3]
```

```
## [1] "AZ" "CA"
```

```
states[density < 3]
```

```
## [1] "NM"
```

```
states[density <= 3]
```

```
## [1] "AZ" "NM"
```

- Can subset a vector based on itself
- If we want to look at the densities greater than 3
- `density` is both the vector being subset and part of the condition

```
density[density > 3]
```

```
## [1] 3.200000 3.703704
```

How is this actually working?

Let's look more closely at the code inside the `[]`.

```
density > 3
```

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

This does an element-wise check to see if each value is > 3. If it is the result is `TRUE`, if not it is `FALSE`.

The `density[]` part of the code is saying to look in the `density` vector and return only those values where this inner vector value is `TRUE`.

Let's Practice

Wrap up the Assignment by working Question 8 (and the optional Question 9).