SPEC Lab REU R Resources: Intro to R - Vectors & Tibbles

Alix Ziff, Gaea Morales, Zachary Johnson

Summer 2022

1 Walk Through Work: Vectors & Tibbles

A **vector** is the simplest type of data you can work with in `R`. “A vector or a one-dimensional array simply represents a collection of information stored in a specific order” (Imai 2016: 6). It is essentially a list of data of a single type (either numerical, character, or logical). (See also, Minions https://www.youtube.com/watch?v=A05n32Bl0aY). Another way to think about it is as a column of data in a table (or dataframe in R–more on this later).

1.1 Variable types

There are four main variable types that you should be familiar with:

- **Numerical**: Well, any number. These can include:
- **numeric** (num): any number (including decimals); less commonly, these values may also be called **double**, which simply describes numerical values with decimals (non-integer)
- **integer** (int): whole numbers
- **Character**: We typically store any alphanumeric data that is not ordered as a character vector.
- **Logical**: A collection of `TRUE` and `FALSE` values.
- **Factor**: Think about it as an ordinal variable, i.e. an ordered categorical variable.

Different Ways of Checking Data Type 1) Global Environment - the abbreviation (e.g., num or int) will show up next to the object 2) Using the str() command 3) Using the class() command

1.2 Vectors

To create a vector we use the function `c()` (`concatenate`) to combine separate data points. The general format for creating a vector in R is as follows:

```
name_of_vector <- c("what you want to put into the vector")
```

Suppose, we have data on the population in millions for 5 the five most populous countries in 2016. We will create a vector with numerical values separated by a comma.

```
c(1379, 1331, 325, 258, 207)
```

```
## [1] 1379 1331  325  258  207
```

Yay our vector now exists! But where? And how do we access it? When we name a vector, it is saved as an *object* (how R stores information) which we can later modify or use with other functions. To assign values to objects, we use the assignment operator `<-`.

```
pop1 <- c(1379, 1331, 325, 258, 207)
pop1
```

```
## [1] 1379 1331  325  258  207
```

That’s better.

We can also use the function `c()` to combine two vectors. Suppose we had data on 5 additional countries.

```
pop2 <- c(194, 187, 161, 142, 127)
pop <- c(pop1, pop2)
pop
```

```
##  [1] 1379 1331  325  258  207  194  187  161  142  127
```

What if we want to know more about or add data to our vector?

1.3 Tell me about it

First, lets check which variable type was used to store our population data. The below output using the string command (str()) tells us that the object `pop` is of class numeric, and has the dimensions `[1:10]`, that is 10 elements in one dimension.

```
str(pop)
```

```
##  num [1:10] 1379 1331 325 258 207 ...
```

Note: str() provides useful information on the structure of the vector, i.e., will breakdown how many observations, and the kind of data for each piece of information.

1.4 Indexing

What happens when we want to extract the nth value from a vector? The **index** is a number that tells you the position of the value of interest (e.g., an index of 1 = 1st value in vector). Let’s use the pop2 vector as an example.

Say we wanted to print the population of the 3rd country in the list. We would run this code:

```
pop2[3]
```

```
## [1] 161
```

What if we wanted to get rid of the last country in the set of 5 (i.e., the 5th element)? We would use the negative sign, and type:

```
pop2[-5]
```

```
## [1] 194 187 161 142
```

What if we wanted to find out multiple values from the same vector at once, e.g. the 2nd, 3rd, and 4th values? We can use concatenate, or use the colon (to mean from x value to y value, i.e. `x:y`):

```
pop2[c(2,3,4)] # or
```

```
## [1] 187 161 142
```

```
pop2[2:4]
```

```
## [1] 187 161 142
```

Can you think of what the code would look like if we wanted to *drop* values 2-4 instead?

```
pop2[-c(2,3,4)] # or
```

```
## [1] 194 127
```

```
pop2[-c(2:4)]
```

```
## [1] 194 127
```

```
# we add a minus sign before the concatenate function to say we want to remove everything in this list
```

1.4.1 Functions

This is a review from the ealier walkthrough on *calculations*. There are a number of special functions that operate on vectors and allow us to compute measures of location and dispersion.

	Function
`min()`	Returns the minimum of the values or object.
`max()`	Returns the maximum of the values or object.
`sum()`	Returns the sum of the values or object.
`length()`	Returns the length of the values or object.
`mean()`	Returns the average of the values or object.
`median()`	Returns the median of the values or object.
`var()`	Returns the variance of the values or object.
`sd()`	Returns the standard deviation of the values or object.

What is the average population size of these countries?

```
mean(pop)
```

```
## [1] 431.1
```

How many people total live in all these places?

```
sum(pop)
```

```
## [1] 4311
```

How long (i.e., how many elements) are in our vector? (We already know the answer, but this is useful to know for when we are working with larger vectors).

```
length(pop)
```

```
## [1] 10
```

We can also get fancy and incorporate multiple arithmetic functions. For example, which values are above the median value in our vector? And how many values are above the mean?

```
pop[pop > median(pop)] # gives us the values
```

```
## [1] 1379 1331  325  258  207
```

```
length(pop[pop > median(pop)]) # tells us how many the values are
```

```
## [1] 5
```

```
# Or, calculating separately:
median(pop)
```

```
## [1] 200.5
```

```
pop[pop > 200.5]
```

```
## [1] 1379 1331  325  258  207
```

##More please! Suppose, we wanted to add data on the country names. We enter this data in character format. To save time, we will only do this for the five most populous countries.

```
cname <- c("CHN", "IND", "USA", "IDN", "BRA")
str(cname)
```

```
##  chr [1:5] "CHN" "IND" "USA" "IDN" "BRA"
```

Now, lets code a logical variable that shows whether the country is in Asia or not.

```
asia <- c(TRUE, TRUE, F, T, F) # TRUE and T, and FALSE and F are interchangeable,
  # but do not forget to capitalize
str(asia)
```

```
##  logi [1:5] TRUE TRUE FALSE TRUE FALSE
```

Lastly, we define a factor variable for the regime type that can take one of four values (based on Economist Intelligence Unit): Full Democracy, Flawed Democracy, Hybrid Regimes, Autocracies. Note that empirically, we don't have a "hybrid category" here. We could define an empty factor level, but we will skip this step here.

```
regime <- c("Autocracy", "FlawedDem", "FullDem", "FlawedDem", "FlawedDem")
regime <- as.factor(regime)
str(regime)
```

```
##  Factor w/ 3 levels "Autocracy","FlawedDem",..: 1 2 3 2 2
```

1.5 Introducing Tibbles

As social scientists, the data we work with rarely consist of single vectors. For example, a list of populations is not incredibly informative if we do not know that they are referring to specific countries, and in what year the data were collected. We might also be curious about other characteristics of these countries, such as what region they are in, or their regime type or levels of democracy.

Now that we have some background on vectors, we can begin to think about the next level: tibbles. We can thinking of `tibbles` as tables, or very basic forms of `dataframes`, consisting of columns of data. In other words, tibbles are named lists of vectors *of the same length*. Equal lengths are important because tibbles are organized sequentially: the first element in vector `a` will be paired with the first element in vectors `b` and `c`, the second element in `a` to the second element in `b` and `c`, and so on.

2 Creating Tibbles

We can create a tibble using the three vectors describing the five most populous countries in 2016 (`cname`, `pop1`, `asia`, `regime`) we just saved into our global environment and the `tibble()` command. We can respecify the information in each of our vectors using the concatenate (`c()`) function, or more simply type in the object names as we have the data saved in the environment.

```
# Option 1
tib1 <- tibble::tibble(
  cname = c("CHN", "IND", "USA", "IDN", "BRA"),
  pop1 = c(1379, 1331, 325, 258, 20),
  asia = c(TRUE, TRUE, F, T, F),
  regime = c("Autocracy", "FlawedDem", "FullDem", "FlawedDem", "FlawedDem"))

# Option 2 (Shortcut)
tib2 <- tibble::tibble(cname, pop1, asia, regime)

View(tib1)
View(tib2)
```

Here you can see that both options yield the same results. Our vectors have been grouped into a tibble which is named and saved in our global environment.

Just to demonstrate, let's see what happens when we try to put together vectors of different lengths.

```
tib_test <- tibble::tibble(
  cname = c("CHN", "IND", "USA", "IDN", "BRA"),
  pop = c(1379, 1331, 325, 258, 20, 194, 187, 161, 142, 127),
  asia = c(TRUE, TRUE, F, T, F),
  regime = c("Autocracy", "FlawedDem", "FullDem", "FlawedDem", "FlawedDem"))
```

```
## Error:
## ! Tibble columns must have compatible sizes.
## * Size 5: Existing data.
## * Size 10: Column ‘pop‘.
## i Only values of size one are recycled.
```

As we expected, the `tibble()` command produces an error saying that “tibble columns must have compatible sizes”. What this means, is that tibbles must have the same length. Later on, we will learn to more on working with `data frames`. Data frames are not unlike tibbles, but allow for greater flexibility for data analysis in terms of merging data, renaming variables, printing (displaying results), and subsetting (selecting only key columns or rows) among others.