Lab 2: Vectors and other data structures

Data Science UTB

Enrique J. De La Hoz D.

Learning Objectives

- Work with vectors of different data types
- Understand the concept of *atomic* structures
- Learn how to subset and slice R vectors
- Understand the concept of *vectorization*
- Understand *recycling* rules in R

Getting the Data File

The data that you are going to work with is part of the information that wikipedia has about the States of the United States https://en.wikipedia.org/wiki/List_of_states_and_territories_of_the_United_States

Open a new session in Rstudio, and make sure you have a clean workspace:

```
# remove existing objects
rm(list = ls())
```

Use the following code to read the data contained in the file `usa-states.RData` (from the course's github repository)

```
# download RData file into your working directory
rdata <- "https://github.com/ucb-stat133/stat133-fall-2016/raw/master/data/usa-states.RData"
download.file(url = rdata, destfile = 'usa-states.RData')

# load data in your R session
load('usa-states.RData')

# list the available objects with ls()
ls()
```

Confirm that you have the following objects:

- `state` (names of the States)
- `capital` (names of the capitals)
- `area` (total area in km2)
- `water` (water area in km2)
- `seats` (number of house seats)

Inspecting the data objects

Once you have some data objects to work with, the first step is to inspect some of their characteristics. R has a handful of functions that allows you to examine objects:

- `typeof()` type of storage of any object
- `class()` gives you the class of the object
- `str()` displays the structure of an object in a compact way

- `mode()` gives the data type (as used in R)
- `object.size()` gives an estimate of the memory space used by an object
- `length()` gives the length (i.e. number of elements)
- `head()` take a peek at the first elements
- `tail()` take a peek at the last elements
- `summary()` shows a summary of a given object

Let's find out what is the class of each of the objects:

```
# your code here
```

Now use `length()`, `head()`, `tail()`, and `summary()` to start exploring the content of the objects:

```
# your code here
```

Vectors in R

Vectors are the most basic type of data structures in R. Learning how to manipulate data structures in R requires you to start learning how to manipulate vectors.

Remember that R vectors are **atomic structures**, which is just the fancy name to indicate that all the elements of a vector must be of the same type, either all numbers, all characters, or all logical values.

How do you know that a given vector is of a certain data type? With `typeof()`

```
typeof(state)
typeof(capital)
typeof(area)
typeof(water)
```

Manipulating Vectors: Subsetting

Subsetting refers to extracting elements of a vector (or another R object). To do so, you use what is known as **bracket notation**. This implies using (square) brackets `[ ]` to get access to the elements of a vector:

```
# first element of 'state'
state[1]

# first five elements of 'state'
state[1:5]
```

What type of things can you specify inside the brackets? Basically:

- numeric vectors
- logical vectors (the length of the logical vector must match the length of the vector to be subset)
- character vectors (if the elements have names)

Subsetting with Numeric Indices

Here are some subsetting examples using a numeric vector inside the brackets:

```
# fifth element of 'state'
state[5]

# numeric range
state[2:8]
```

```r
# numeric vector
state[c(1, 3, 5, 7)]

# different order
state[c(20, 9, 10, 50)]

# third element (four times)
state[rep(3, 4)]
```

Subsetting with Logical Indices

Logical subsetting involves using a logical vector inside the brackets. This type of subsetting is very powerful because it allows you to extract elements based on some logical condition.

To do logical subsetting, the vector that you put inside the brackets, must match the length of the manipulated vector.

Here are some examples of logical subsetting:

```r
# area of California
area[state == 'California']

# name of states with areas greater than 400,000 square km
state[area > 400000]

# name of states with areas between 100,000 and 125,000 square km
state[area > 100000 & area < 125000]
```

Your turn

Write commands to answer the following questions:

```r
# name of the state with largest area


# name of the state with smallest area


# name of the state with largest number of seats


# capital of the state with the smallest water area


#
```

Subsetting with Character Vectors

A third type of subsetting involves passing a character vector inside brackets. When you do this, the characters are supposed to be names of the manipulated vector.

None of the vectors `state`, `capital`, `area`, `water`, and `seats` have names. You can confirm that with the `names()` function applied on any of the vectors:

```
names(state)
```

Create a new vector `total` by adding `area` and `water`, and then assign `state` as the names of `total`

```
# create 'total


# assign 'state' as names of 'total'
```

You should have a vector `total` with named elements. Now you can use character subsetting:

```
total["Alabama"]

total[c("California", "Oregon", "Washington")]

total[c("Texas", "Alaska")]
```

Some plotting

Use the function `plot()` to make a scatterplot of `area` and `water`

```
plot(area, water)
```

Keep in mind tht `plot()` is a generic function. This means that the behavior of `plot()` depends on the type of input. When you pass two numeric vectors to `plot()` it will create a scatter plot.

Looking at the generated plot, can you see any issues?

To get a better display of the scatterplot, let's create two vectors `log_area` and `log_water` by transforming `area` and `water` with the logarithm function `log()`

```
log_area <- log(area)
log_water <- log(water)
```

Make another scatterplot but now use the log-transformed vectors:

```
plot(log_area, log_water)
```

To add the names of the states in the plot, you can use `text()`:

```
plot(log_area, log_water)
text(log_area, log_water, labels = state)
```

Now we have another problem. The labels in the plot are a bit messy. A quick and dirty fix is to use `abbreviate()` to shorten the displayed names:

```
plot(log_area, log_water)
text(log_area, log_water, labels = abbreviate(state))
```

Vectorization

When you create the vectors `log_area <- log(area)` and `log_water <- log(water)`, what you're doing is applying a function to a vector, which in turn acts on all elements of the vector.

This is called **Vectorization** in R parlance. Most functions that operate with vectors in R are **vectorized** functions. This means that an action is applied to all elements of the vector without the need to explicitly type commands to traverse all the elements.

In many other programming languages, you would have to use a set of commands to loop over each element of a vector (or list of numbers) to transform them. But not in R.

Another example of vectorization would be the calculation of the square root of all the elements in `area` and `water`:

```
sqrt(area)
sqrt(water)
```

Or the sum of `area` plus `water` to get the total area:

```
area + water
```

Why should you care about vectorzation?

If you are new to programming, learning about R's vectorization will be very natural (you won't stop to think about it too much). If you have some previous programming experience in other languages (e.g. C, python, perl), you know that vectorization does not tend to be a native thing.

Vectorization is essential in R. It saves you from typing many lines of code, and you will exploit vectorization with other useful functions known as the *apply* family functions (we'll talk about them later in the course).

Recycling

Closely related with the concept of *vectorization* we have the notion of **Recycling**. To explain *recycling* let's see an example.

`area` and `water` are given in square kilometers, but what if you need to obtain the areas in square miles?. Let's create two new vectors `area_square_miles` and `water_square_miles` with the converted values in square miles. To convert from square kilometers to aquare miles use the following conversion: 1 sqr km = 0.386 sqr mi

```
# your code here
```

What you just did (assuming that you did things correctly) is called **Recycling**. To understand this concept, you need to remember that R does not have a data structure for scalars (single numbers). Scalars are in reality vectors of length 1.

Converting square kms to square miles requires this operation: `area * 0.836`. Although it may not be obvious, we are multiplying two vectors: `area` and `0.836`. Moreover (and more important) **we are multiplying two vectors of different lengths!**. So how does R know what to do in this cases?

Well, R uses the **recycling rule**, which takes the shorter vector (in this case `0.386`) and recycles its elements to form a temporary vector that matches the length of the longer vector (i.e. `area`).

Another recycling example

Here's another example of recycling. Areas of elements in an odd number position will be transformed to square miles; areas of elements in an even number position will be transformed to acres:

```
units <- c(0.386, 247.105)
new_area <- area * units
```

The elements of `units` are recycled and repeated as many times as elements in `area`. The previous command is equivalent to this:

```r
new_units <- rep(c(0.386, 247.105), length.out = length(area))
area * new_units
```

Your turn

Make a scatterplot of `area_square_miles` and `water_square_miles` transforming the vectors into log scale (use `log()`)

```r
# your scatter plot
```

Factors

As mentioned before, vectors are the most essential type of data structure in R. They are *atomic* structures (can contain only one type of data): integers, real numbers, logical values, characters, complex numbers.

Related to vectors, there is another important data structure in R called **factor**. Factors are data structures exclusively designed to handle categorical data.

The term *factor* as used in R for handling categorical variables, comes from the terminology used in *Analysis of Variance*, commonly referred to as ANOVA. In this statistical method, a categorical variable is commonly referred to as *factor* and its categories are known as *levels*.

Creating Factors

To create a factor you use the homonym function `factor()`, which takes a vector as input. The vector can be either numeric, character or logical.

```r
# numeric vector
num_vector <- c(1, 2, 3, 1, 2, 3, 2)

# creating a factor from num_vector
first_factor <- factor(num_vector)

first_factor
```

```
## [1] 1 2 3 1 2 3 2
## Levels: 1 2 3
```

You can also obtain a factor from a character vector:

```r
# string vector
str_vector <- c('a', 'b', 'c', 'b', 'c', 'a', 'c', 'b')

str_vector
```

```
## [1] "a" "b" "c" "b" "c" "a" "c" "b"
```

```r
# creating a factor from str_vector
second_factor <- factor(str_vector)

second_factor
```

```
## [1] a b c b c a c b
## Levels: a b c
```

Notice how `str_vector` and `second_factor` are displayed. Even though the elements are the same in both the vector and the factor, they are printed in different formats. The letters in the string vector are displayed with quotes, while the letters in the factor are printed without quotes.

How does R store factors?

Under the hood, a factor is internally stored using two arrays: one is an integer array containing the values of categories, the other array is the "levels" which has the names of categories which are mapped to the integers.

One way to confirm that the values of the categories are mapped as integers is by using the function `storage.mode()`

```
# storage of factor
storage.mode(first_factor)
```

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

Manipulating Factors

Because factors are internally stored as integers, you can manipulate factors as any other vector:

```
first_factor[1:5]
```

```
## [1] 1 2 3 1 2
## Levels: 1 2 3
```

```
first_factor[c(1, 3, 5)]
```

```
## [1] 1 3 2
## Levels: 1 2 3
```

```
first_factor[rep(1, 5)]
```

```
## [1] 1 1 1 1 1
## Levels: 1 2 3
```

```
second_factor[second_factor == 'a']
```

```
## [1] a a
## Levels: a b c
```

```
second_factor[second_factor == 'b']
```

```
## [1] b b b
## Levels: a b c
```