lab1_jupyter_vectors_factors

January 29, 2022

Vectors and Factors

0.0.1 Welcome!

By the end of this notebook, you will have learned about **vectors and factors**, two very important data structures in R.

Estimated time needed: **30** minutes

0.1 Objectives

After completing this lab you will be able to:

- Understand R vector via coding practices
- Perform vector operations
- Understand R factor via coding practices
- Perform factor operations

0.2 Table of Contents

About the Dataset

You have received many movie recomendations from your friends and compiled all of the recommendations into a table, with information about each movie.

This table has one row for each movie and several columns.

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)

- **age_restriction** - The age restriction for the movie

Here's what the data looks like:

Vectors

Vectors are collections of numbers, characters or logical data organized in a one dimensional array. In other words, a vector is a simple tool to store your grouped data, element by element.

In R, you create a vector with the combine function **c()**. You place the vector elements separated by a comma between the brackets.

Vectors will be very useful in the future as they allow you to apply operations on a series of data easily.

Note that the elements in a vector must be of the same class, for example all should be either number, character, or logical.

0.2.1 Numeric, Character, and Logical Vectors

Let's say we have four movie release dates (1985, 1999, 2015, 1964) and we want to assign them to a single variable, `release_year`. This means we'll need to create a vector using the `c()` function to **c**ombine them.

Using numbers, this becomes a **numeric vector**.

[2]:
```
release_year <- c(1985, 1999, 2015, 1964)
```

[3]:
```
release_year
```

1. 1985 2. 1999 3. 2015 4. 1964

What if we use single or double quotation marks? Then this becomes a **character vector**.

[4]:
```
# Create genre vector and assign values to it
titles <- c("Toy Story", "Akira", "The Breakfast Club")
titles
```

1. 'Toy Story' 2. 'Akira' 3. 'The Breakfast Club'

There are also **logical vectors**, which consist of TRUEs and FALSEs. They're particular important when you want to check its contents

[5]:
```
titles == "Akira" # which item in `titles` is equal to "Akira"?
```

1. FALSE 2. TRUE 3. FALSE

Coding Exercise: in the code cell below, find which item in `titles` equals to "Toy Story"

[7]:
```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
which(titles=="Toy Story")
```

1

Click here for the solution

```
titles == "Toy Story"
```

[Tip] TRUE and FALSE in R

Did you know? R only recognizes `TRUE`, `FALSE`, `T` and `F` as special values for true and false. That means all other spellings, including *True* and *true*, are not interpreted by R as logical values.

Vector Operations

0.2.2 Adding more elements to a vector

You can add more elements to a vector with the same `c()` function you use the create vectors:

[ ]:
```
release_year <- c(1985, 1999, 2015, 1964)
release_year
```

[ ]:
```
release_year <- c(release_year, 2016:2018)
release_year
```

0.2.3 Length of a vector

How do we check how many items there are in a vector? We can use the **length()** function:

[ ]:
```
release_year
length(release_year)
```

0.2.4 Head and Tail of a vector

We can also retrieve just the **first few items** using the **head()** function:

[ ]:
```
head(release_year) #first six items
```

[ ]:
```
head(release_year, n = 2) #first n items
```

[ ]:
```
head(release_year, 2)
```

We can also retrieve just the **last few items** using the **tail()** function:

[ ]:
```
tail(release_year) #last six items
```

[ ]:
```
tail(release_year, 2) #last two items
```

0.2.5 Sorting a vector

We can also sort a vector:

[ ]:
```
sort(release_year)
```

We can also **sort in decreasing order**:

[ ]:
```
sort(release_year, decreasing = TRUE)
```

But if you just want the minimum and maximum values of a vector, you can use the **`min()`** and **`max()`** functions

[ ]:
```
min(release_year)
max(release_year)
```

0.2.6 Average of Numbers

If you want to check the average cost of movies produced in 2014, what would you do?

Of course, one way is to add all the numbers together, then divide by the number of movies:

[ ]:
```
cost_2014 <- c(8.6, 8.5, 8.1)

# sum results in the sum of all elements in the vector
avg_cost_2014 <- sum(cost_2014)/3
avg_cost_2014
```

You also can use the mean function to find the average of the numeric values in a vector:

[ ]:
```
mean_cost_2014 <- mean(cost_2014)
mean_cost_2014
```

0.2.7 Giving Names to Values in a Vector

Suppose you want to remember which year corresponds to which movie.

With vectors, you can give names to the elements of a vector using the **`names()`** function:

[ ]:
```
#Creating a year vector
release_year <- c(1985, 1999, 2010, 2002)

#Assigning names
names(release_year) <- c("The Breakfast Club", "American Beauty", "Black Swan",␣
 ↪"Chicago")

release_year
```

Now, you can retrieve the values based on the names:

[ ]:
```
release_year[c("American Beauty", "Chicago")]
```

Note that the values of the vector are still the years. We can see this in action by adding a number to the first item:

[ ]:
```
release_year[1] + 100 #adding 100 to the first item changes the year
```

And you can retrieve the names of the vector using **`names()`**

[ ]:
```
names(release_year)
```

Coding Exercise: in the code cell below, calculate the release year difference between 'Black Swan' and 'The Breakfast Club'

[8]:
```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
year_difference <- release_year['Black Swan'] - release_year['The Breakfast␣
 ↪Club']
year_difference
```

```
<NA>
```

Click here for the solution

```
year_difference <- release_year['Black Swan'] - release_year['The Breakfast Club']
year_difference
```

0.2.8 Summarizing Vectors

You can also use the **"summary"** function for simple descriptive statistics: minimum, first quartile, mean, third quartile, maximum:

[ ]:
```
summary(cost_2014)
```

0.2.9 Using Logical Operations on Vectors

A vector can also be comprised of **TRUE** and **FALSE**, which are the **logical values** in R. These boolean values are used to indicate whether a condition is true or false.

Let's check whether a movie year of 1997 is older than (**greater in value than**) 2000.

[ ]:
```
movie_year <- 1997
movie_year > 2000
```

You can also make a logical comparison across multiple items in a vector. Which movie release years here are "greater" than 2014?

[ ]:
```
movies_years <- c(1998, 2010, 2016)
movies_years > 2014
```

We can also check for **equivalence**, using **==**. Let's check which movie year is equal to 2015.

[ ]:
```
movies_years == 2015 # is equal to 2015?
```

If you want to check which ones are **not equal** to 2015, you can use **!=**

[ ]:
```
movies_years != 2015
```

[Tip] Logical Operators in R

You can do a variety of logical operations in R including:

Checking equivalence: $1 == 2$

Checking non-equivalence: $TRUE! = FALSE$

Greater than: $100 > 1$

Greater than or equal to: $100 >= 1$

Less than: $1 < 2$

Less than or equal to: $1 <= 2$

Subsetting Vectors

What if you wanted to retrieve the second year from the following **vector of movie years**?

[1]:
```
movie_years <- c(1985, 1999, 2002, 2010, 2012)
movie_years
```

1. 1985 2. 1999 3. 2002 4. 2010 5. 2012

To retrieve the **second year**, you can use square brackets `[]`:

[2]:
```
movie_years[2] #second item
```

1999

To retrieve the **third year**, you can use:

[3]:
```
movie_years[3]
```

2002

And if you want to retrieve **multiple items**, you can pass in a vector:

[4]:
```
movie_years[c(1,3)] #first and third items
```

1. 1985 2. 2002

You can also get a sequential subset as follows:

[5]:
```
movie_years[c(2:4)] #second to fourth items
```

1. 1999 2. 2002 3. 2010

or, more succinctly:

[6]:
```
movie_years[2:4]
```

1. 1999 2. 2002 3. 2010

Retrieving a vector without some of its items

To retrieve a vector without an item, you can use negative indexing. For example, the following returns a vector slice **without the first item**.

[ ]:
```
titles <- c("Black Swan", "Jumanji", "City of God", "Toy Story", "Casino")
titles[-1]
```

You can assign the new vector to a new variable:

[ ]:
```
new_titles <- titles[-1] #removes "Black Swan", the first item
new_titles
```

Missing Values (NA)

Sometimes values in a vector are missing and you have to show them using NA, which is a special value in R for "Not Available". For example, if you don't know the age restriction for some movies, you can use NA.

[ ]:
```
age_restric <- c(14, 12, 10, NA, 18, NA)
age_restric
```

[Tip] Checking NA in R

You can check if a value is NA by using the **is.na()** function, which returns TRUE or FALSE.

Check if NA: is.na(NA)

Check if not NA: !is.na(2)

0.2.10 Subsetting vectors based on a logical condition

What if we want to know which movies were created after year 2000? We can simply apply a logical comparison across all the items in a vector:

[ ]:
```
release_year > 2000
```

To retrieve the actual movie years after year 2000, you can simply subset the vector using the logical vector within **square brackets "[]"**:

[ ]:
```
release_year[release_year > 2000] #returns a vector for elements that returned␣
 ↪TRUE for the condition
```

As you may notice, subsetting vectors in R works by retrieving items that were TRUE for the provided condition. For example, `year[year > 2000]` can be verbally explained as: *"From the vector `year`, return only values where the values are TRUE for `year > 2000`"*.

You can even manually write out TRUE or T for the values you want to subset:

[ ]:
```
release_year
release_year[c(T, F, F, F)] #returns the values for which the value's index is␣
 ↪TRUE
```

Coding Exercise: in the code cell below, find movies whose release year in or prior to (include) 1999

[ ]:
```
# Write your code below. Don't forget to press Shift+Enter to execute the cell
```

[9]:
```
release_year[release_year <= 1999]
```

1. 1985 2. 1999 3. 1964

Click here for the solution

```
release_year[release_year <= 1999]
```

Factors

Factors are variables in R which take values from a fixed, discrete set of values, or levels. Such variables are commonly referred to as **categorical variables**.

The difference between a categorical variable and a numeric variable is that a categorical variable expresses qualitative values –attributes such as 'age group', 'gender', or 'favourite music genre', while numerical variables express quantitative information – data that can be interpreted mathematically, such as 'fuel economy in MPG', 'price per bushel','age'.

For example, the height of a tree is a measured numeric variable, but the titles of books held in a particular library, or the various species of animals on earth are categorical variables. Categorical variables need not be expressed as text. Indeed, factors in R are actually stored as a vector of integer values – they carry a corresponding set of character values but these are only used for display purposes.

One of the most important uses of factors in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors ensure that the modeling functions will treat such data correctly.

Let's start with a ***vector*** of genres:

[ ]:
```
genre_vector <- c("Comedy", "Animation", "Crime", "Comedy", "Animation")
genre_vector
```

As you may have noticed, you can theoretically group the items above into three categories of genres: *Animation*, *Comedy* and *Crime*. In R-terms, we call these categories **"factor levels"**.

The function **factor()** converts a vector into a factor, and creates a factor level for each unique element.

[ ]:
```
genre_factor <- as.factor(genre_vector)
levels(genre_factor)
```

0.2.11 Summarizing Factors

When you have a large vector, it becomes difficult to identify which levels are most common (e.g., "How many 'Comedy' movies are there?").

To answer this, we can use **summary()**, which produces a **frequency table**, as a named vector.

[ ]:
```
summary(genre_factor)
```

And recall that you can sort the values of the table using **sort()**.

```
[ ]: sort(summary(genre_factor)) #sorts values by ascending order
```

0.2.12 Ordered factors

There are two types of categorical variables: a **nominal categorical variable** and an **ordinal categorical variable**.

A **nominal variable** is a categorical variable for names, without an implied order. This means that it is impossible to say that 'one is better or larger than the other'. For example, consider **movie genre** with the categories *Comedy*, *Animation*, *Crime*, *Comedy*, *Animation*. Here, there is no implicit order of low-to-high or high-to-low between the categories.

In contrast, **ordinal variables** do have a natural ordering. Consider for example, **movie length** with the categories: *Very short*, *Short* , *Medium*, *Long*, *Very long*. Here it is obvious that *Medium* stands above *Short*, and *Long* stands above *Medium*.

```
[12]: movie_length <- c("Very Short", "Short", "Medium","Short", "Long",
                        "Very Short", "Very Long")
      movie_length
```

1. 'Very Short' 2. 'Short' 3. 'Medium' 4. 'Short' 5. 'Long' 6. 'Very Short' 7. 'Very Long'

`movie_length` should be converted to an ordinal factor since its categories have a natural ordering. By default, the function factor() transforms `movie_length` into an unordered factor.

To create an **ordered factor**, you have to add two additional arguments: `ordered` and `levels`.

- `ordered`: When set to `TRUE` in `factor()`, you indicate that the factor is ordered.
- `levels`: In this argument in `factor()`, you give the values of the factor in the correct order.

```
[13]: movie_length_ordered <- factor(movie_length, ordered = TRUE,
                                     levels = c("Very Short", "Short", "Medium",␣
       ↪"Long", "Very Long"))
      movie_length_ordered
```

1. Very Short 2. Short 3. Medium 4. Short 5. Long 6. Very Short 7. Very Long

Levels: 1. 'Very Short' 2. 'Short' 3. 'Medium' 4. 'Long' 5. 'Very Long'

Now, lets look at the summary of the ordered factor, movie_length_ordered:

```
[ ]: summary(movie_length_ordered)
```

Coding Exercise: in the code cell below, update the order of the movie_length factor from `Very Long` to `Very Short`

```
[14]: # Write your code below. Don't forget to press Shift+Enter to execute the cell
      movie_length_ordered <- factor(movie_length, ordered = TRUE ,
                                      levels = c("Very Long", "Long", "Medium",
                                                 "Short", "Very Short"))
      movie_length_ordered
```

1. Very Short 2. Short 3. Medium 4. Short 5. Long 6. Very Short 7. Very Long

Levels: 1. 'Very Long' 2. 'Long' 3. 'Medium' 4. 'Short' 5. 'Very Short'

Click here for the solution

```
movie_length_ordered <- factor(movie_length, ordered = TRUE ,
                                levels = c("Very Long", "Long", "Medium",
                                           "Short", "Very Short"))
movie_length_ordered
```

0.2.13 Excellent! You have just completed the Vectors and Factors lab

Scaling R with big data As you learn more about R, if you are interested in exploring platforms that can help you run analyses at scale, you might want to sign up for a free account on IBM Watson Studio, which allows you to run analyses in R with two Spark executors for free.

0.3 About the Author:

Hi! It's Helly Patel, the author of this notebook. I hope you found R easy to learn! There's lots more to learn about R but you're well on your way. Feel free to connect with me if you have any questions.

0.3.1 Other Contributors

Yan Luo

0.4 Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2021-03-03	2.0	Yan	Added coding tasks

##