R Basics

COMM 205 - Lecture 17 - R2

Hasan Cavusoglu

2020

Agenda

- RScript/ R file
- Logical AND (`&`) and OR(|)
- Vectors
- `length()`
- Subsetting (i.e., element Extraction)
- `which()`
- Implicit coercion in vector creation

R Script (R file)

- Earlier, we have seen how to execute a piece of R code on R console.
  - The downside: you *cannot* retain what you have already executed.
- If you want to retain you code, you should store in (i.e., save to) a file.
  - `R Script` is a file type that we will use to save our R code.
  - We will use `R Script` and `R file`, *interchangeably*

R Script

R Script is a *plain text file* with an extension of **.R**.

- `R Script` allows you to
  - save your code
  - re-execute it when needed
  - share your code

R Script/ R file (cont'd)

- For interactive programming in this course, I encourage you to use R file to store your codes so that you can re-execute them as needed.
- Saving your code in a file makes your code *reproducible*. That is, if someone shares an R file with you, you would get the same output when you run the code. When you open an R file, the source pane becomes notebook interface.

How to open an R file

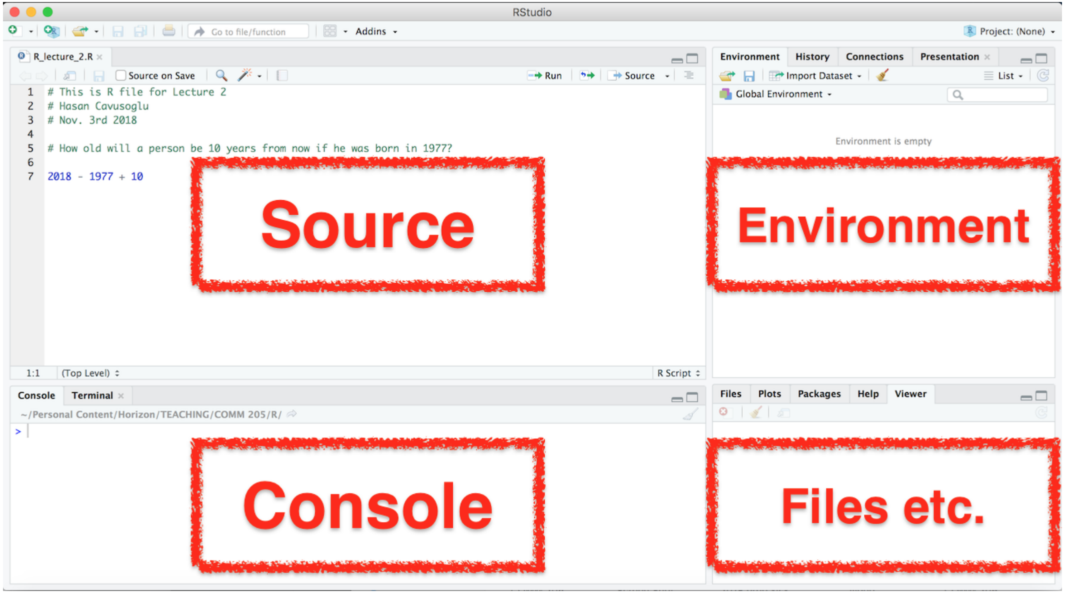
Follow the steps below to open an R file on R Studio:

- Click on the folder icon on the top left of the screen (*or* `File -> Open File...`)
- Locate the file and hit `Open`

The R will be displayed in the **code pane** on the top left. Now, you should see four windows (i.e., panes) as follows:

- Source (or code editor) on the top left
- Console on the bottom left
- Environment on the top right
- Files/Plots/Packages/Help/Viewer on the bottom right

When a source file is open



```
# This is R file for Lecture 2
# Hasan Cavusoglu
# Nov. 3rd 2018

# How old will a person be 10 years from now if he was born in 1977?

2018 - 1977 + 10
```

~/Personal Content/Horizon/TEACHING/COMM 205/R/

Environment is empty

How to Execute Code on R File

Depending on how much of the code you want to execute, you can use one of the three ways of running code in an R file.

- **Running the code in a given line of R File**: You need to send the individual line of code from the editor/source to the console.
  - Click the line of code you want to run,
  - Click on the Run button or press `Ctrl` + `Enter` (OS X: `Cmd` + `Enter`)
- **Running a segment of the code**: You need the send the block of highlighred code to the console.
  - Highlight a segment of the code and press `Ctrl` + `Enter` (OS X: `Cmd` + `Enter`) to run just the highlighted segment.
- **Running the entire code in R File**: You need to send the entire script to the console (which is called sourcing a script).
  - Click the Source button or highlight the entire code and press `Ctrl` + `Enter` (OS X: `Cmd` + `Enter`) to run just the highlighted segment.

Example

How old will a person be 10 years from now if he was born in 1977?

I have already coded this. Please open `R_lecture_1.R`, and then execute the segment of the code that answers the question with one of the ways described in the previous slide.

- You can provide a **comment** for you or those who will read your R code. Anything after a number sign, hash, or pound sign, i.e.# will be ignored by R.
- R executes one line of code at a time. If there is an output needs to be displayed, it will display the output after the corresponding line. If there is no output to displayed, R moves on to execute the next line.

Saving / Creating an R File

Saving changes to an exiting R file

You may want to save changes to an existing R file after modified. To do so, hit the save button, or File -> Save.

Creating a new R File

To create a new R file,

- click on the small triangle next to the blank document icon on the top left of the screen and select `R Script`
- An untitled new R file is created as a new tab on the source pane
  - You need to save it by hitting the save button, or File -> Save.

Logical Operators

- **Logical Operators** are typically used with logical values. They return a logical value.
  - AND: &
  - OR: |
  - NOT: !

Logical AND

& is the logical AND operation. & returns `TRUE` **only** when both values it connects are `TRUE`.

Logical OR

| is the logical OR operation. | returns `TRUE` when either value is `TRUE`

Logical NOT

! is the logical NOT operator. ! negates the logical value it preceeds.

Logical Operators (AND, OR, NOT)

Logical Operation	Result
TRUE & TRUE	TRUE
TRUE & FALSE	FALSE
FALSE & TRUE	FALSE
FALSE & FALSE	FALSE

Logical Operation	Result
TRUE | TRUE	TRUE
TRUE | FALSE	TRUE
FALSE | TRUE	TRUE
FALSE | FALSE	FALSE

Logical Operation	Result
!TRUE	FALSE
!FALSE	TRUE

Precedence in logical operators

The precedence in logical operators (from the highest to the lowest)

- NOT (i.e., !)
- AND (i.e., &)
- OR (i.e., |)

The following example illustrate the precedence.

```
> TRUE | !FALSE & FALSE
[1] TRUE
```

- First priority is NOT operator. Since `!FALSE` is equal to `TRUE`, the expression is identical to `TRUE | TRUE & FALSE`.
- The second priority is AND operator. Since `TRUE & FALSE` is equal to `FALSE`, the expression is identical to `TRUE | FALSE`.
- And, we know `TRUE | FALSE` is equal to (which is `TRUE`).

The complete list of precedence that R gives to the operators can be found in R documentation.

Combining Comparisons

- Logical Operators (&,|, or !) can combine multiple comparisons
- Suppose you want to confirm
  - Toronto's population is greater than 1 million and less than 3 million.
  - Vancouver's population is not greater than Toronto's and Toronto's population is not greater than 3 million

```
> toronto <-  2790000
> vancouver <- 632000
> toronto > 1000000 & toronto < 3000000
[1] TRUE
> !vancouver>toronto &
+                !toronto>3000000
[1] TRUE
```

Vectors

- Remember in the last class, each object holds one value. Each of these objects holding a single value is referred to as *atomic data*
  - Excel analogy to **atomic data** would be data in a **cell**.

Vector

- A **vector** is a set of multiple values of the same data type (i.e., numeric, character or logical) in the same object.
- Accordingly, you can have numeric vectors, character vectors or logical vectors.
  - Excel analogy to **vector** would be data in a **a column/row array**.

Vector Examples

- You can think of a vector as a collection of observations or measurements of a single variable.
- Examples:

```
city <- c("Vancouver", "Victoria", "Delta", "Surrey")
today_temp_C <- c(25, 20, 27, 30)
is.sunny <- c(FALSE, FALSE, TRUE, TRUE)
```

c() function

- `c()` function with desired elements as its arguments creates a vector.

Vector's Data Type

Please note that each element in a vector has to be of the same data type. You can verify the data type of a vector by using `typeof()`. Note that it will give you the data type of the vector's elements. From our previous example:

```
> typeof(city)
[1] "character"
> typeof(today_temp_C)
[1] "double"
> typeof(is.sunny)
[1] "logical"
```

Vector in Logical Comparisons

- Logical comparison (e.g. >,>=,<,<=,==, or !=) involving a vector and an atomic value

- An atomic value will be compared to each element of the vector

Today's temperature values for four cities were recorded in `today_temp_C`. You want to identify if there are cities hotter than 26 Celsius,

```
today_temp_C <- c(25, 20, 27, 30)
today_temp_C > 26
```

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

- The last two temperatures are greater than 26 (judging from the locations of TRUE).

Vector in Logical Comparisons

Logical comparison involving vectors

- Which city today is colder than yesterday?

```
yesterday_temp_C <- c(20, 22, 29, 30)
yesterday_temp_C
```

```
## [1] 20 22 29 30
```

```
today_temp_C
```

```
## [1] 25 20 27 30
```

```
today_temp_C < yesterday_temp_C
```

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

- The 2nd and 3rd values are smaller than those yesterday (i.e., the locations of TRUE).

Vector in Logical Operations

- Logical vectors can be combined using AND (&) or OR (|)

If vectors are combined with logical operator (& or |), elements in the corresponding locations are used (element-wise).

```
a <- c(TRUE,FALSE,TRUE)
b <- c(TRUE,TRUE, FALSE)
a & b
```

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

```
a | b
```

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

Vector in Arithmetic Operations

- Arithmetic operations (+, -, *, /, and ˆ) involving a vector and an atomic value

- Tomorrow temperature will be 2 degrees higher in all four cities than today. What are the tomorrow's temperature values

```
today_temp_C
```

```
## [1] 25 20 27 30
```

```
tomorrow_temp_C <- today_temp_C + 2
tomorrow_temp_C
```

```
## [1] 27 22 29 32
```

- The atomic value is applied to each element of the vector.

Vector in Arithmetic Operations

- Arithmetic operations involving vectors

- Later, suppose the forecasts have changed. Now,tomorrow temperature will be 2 degrees higher in the first two cities and 1 degree higher in the last two cities. What are the tomorrow's temperature values?

```
today_temp_C
```

```
## [1] 25 20 27 30
```

```
tomorrow_temp_C <- today_temp_C + c(2,2,1,1)
tomorrow_temp_C
```

```
## [1] 27 22 28 31
```

- The arithmetic operation is applied element-wise. Values in the corresponding positions are added together.

length() function

- This is a function to determine how many elements exists in an R object specified within the parentheses (i.e., passed on as its argument).

Syntax of length()

length(x)
where x is an *R object*. if x is a vector, *length(x)* will output the number of elements in x.

```
> length(city)
[1] 4
> length(c(80, 75, 90, 85, 90))
[1] 5
```

Subsetting (i.e., Element Extraction) from a Vector

Subsetting is retrieving specific element(s) from a vector. There are two ways of subsetting:

- Using the *location(s)* (i.e., index(es))
- Using *logical vector* (outside the scope of this course)

Subsetting by Using Index

Index is the position of an element within a vector. Index starts from 1. Subsetting is retrieving specific element(s) from a vector by specifying the indexes of desired elements.

Subsetting with Index/Indices

name[index/indices]
where name is the name of the vector

Subsetting Example

- Extract a single element

```
> city
[1] "Vancouver" "Victoria"  "Delta"      "Surrey"
> city[2]
[1] "Victoria
```

```
> city[3]
[1] "Delta"
```

- Extract a range of indexes

```
> city[1:3]
[1] "Vancouver" "Victoria"  "Delta"
```

Examples (cont'd)

- Extract a particular set of indices

```
> city[c(1,2,4)]
[1] "Vancouver" "Victoria"  "Surrey"
```

- Extract everything except a particular element

```
> city[-3]
[1] "Vancouver" "Victoria"  "Surrey"
```

- Extract everything except a set of elements

```
> city[-c(2,3)]
[1] "Vancouver" "Surrey"
```

what else you can do with subsetting

- You can assign a subset vector to a new vector. The right-hand side is the subset and assigned to a new vector.

```
> expensive_city <- city[1:2]
> expensive_city
[1] "Vancouver" "Victoria"
```

- You can also update the value(s) of the subset of a vector.

```
> expensive_city[2] <- "Toronto"
> expensive_city
[1] "Vancouver" "Toronto"
```

- You can add new element(s) to an existing vector.

```
> expensive_city[3:4] <- c("New York","Los Angeles")
> expensive_city
[1] "Vancouver"   "Toronto"     "New York"    "Los Angeles"
```

Caution

- Note that you can add new element(s) ***only*** to *an existing* vector. R will give you an error if you try to add new elements to non-existing object.

```
> inexpensive_city[1:3] <- c("Dallas","Phoenix","Tampa")
Error in inexpensive_city[1:3] <- c("Dallas", "Phoenix", "Tampa") :
  object 'inexpensive_city' not found
```

- The correct approach would have been to create the `inexpensive_city` vector as follows.

```
inexpensive_city <- c("Dallas","Phoenix","Tampa")
```

Now, `inexpensive_city` vector is created, you can add new elements to it as usual.

```
inexpensive_city[4:6] <- c("Winnipeg", "Saskatoon", "Edmonton")
```

which() function

syntax of which()

'which' function gives the indexes of TRUE's of the vector in its argument.
The output is a vector of position numbers.
which(x)
where x is a logical vector

```
> which(c(TRUE,FALSE,TRUE))
[1] 1 3
```

- This function is very useful to identify particular observation(s) which satisfy the condition specified.

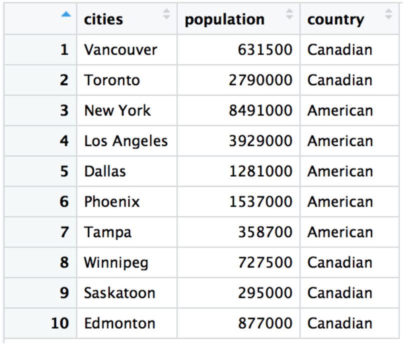
which() Function Example

- To illustrate which, let's first create a vector containing expensive and inexpensive cities. And then create two vectors: one vector which contains population of each corresponding city in cities vector and another one which indicates when the corresponding city is Canadian or American.

```
cities <- c(expensive_city, inexpensive_city)
population <- c(631500, 2790000, 8491000, 3929000,
                1281000, 1537000,358700, 727500,
                295000, 877000)
country <- c("Canadian","Canadian","American","American",
             "American","American","American","Canadian",
             "Canadian","Canadian")
```

Question

What are those cities whose population is greater than 1,000,000?

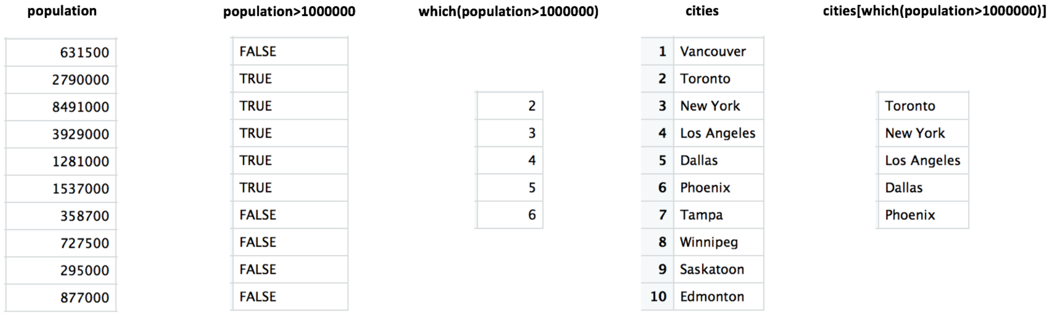


	cities	population	country
1	Vancouver	631500	Canadian
2	Toronto	2790000	Canadian
3	New York	8491000	American
4	Los Angeles	3929000	American
5	Dallas	1281000	American
6	Phoenix	1537000	American
7	Tampa	358700	American
8	Winnipeg	727500	Canadian
9	Saskatoon	295000	Canadian
10	Edmonton	877000	Canadian

- Solution approach:
  - First, we should find the indexes of population vector whose elements are greater than 1,000,000.
  - Then, we should subset the cities vector for those indexes.
  - Analogy in Excel would be =INDEX(MATCH())

Answer

```
> population>1000000
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
> which(population>1000000)
[1] 2 3 4 5 6
> cities[which(population>1000000)]
[1] "Toronto"     "New York"    "Los Angeles" "Dallas"      "Phoeni
```



population
631500
2790000
8491000
3929000
1281000
1537000
358700
727500
295000
877000

population>1000000
FALSE
TRUE
TRUE
TRUE
TRUE
TRUE
FALSE
FALSE
FALSE
FALSE

which(population>1000000)
2
3
4
5
6

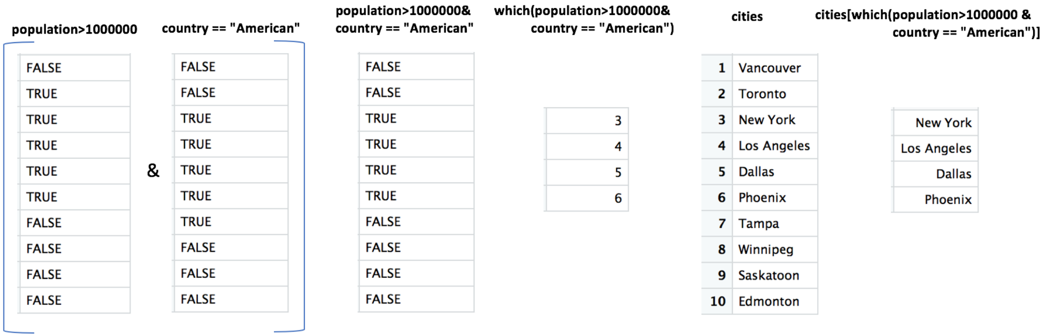
cities	
1	Vancouver
2	Toronto
3	New York
4	Los Angeles
5	Dallas
6	Phoenix
7	Tampa
8	Winnipeg
9	Saskatoon
10	Edmonton

cities[which(population>1000000)]
Toronto
New York
Los Angeles
Dallas
Phoenix

Finding indexes satisfying multiple conditions

Suppose you want to find names of the American cities whose population is larger than 1 million.

```
> cities[which(population>1000000 & country == "American")]
[1] "New York"    "Los Angeles" "Dallas"      "Phoenix"
```



population>1000000		country == "American"	population>1000000& country == "American"
FALSE		FALSE	FALSE
TRUE		FALSE	FALSE
TRUE		TRUE	TRUE
TRUE		TRUE	TRUE
TRUE	&	TRUE	TRUE
TRUE		TRUE	TRUE
FALSE		TRUE	FALSE
FALSE		FALSE	FALSE
FALSE		FALSE	FALSE
FALSE		FALSE	FALSE

which(population>1000000& country == "American")
3
4
5
6

	cities
1	Vancouver
2	Toronto
3	New York
4	Los Angeles
5	Dallas
6	Phoenix
7	Tampa
8	Winnipeg
9	Saskatoon
10	Edmonton

cities[which(population>1000000 & country == "American")]
New York
Los Angeles
Dallas
Phoenix

Some Basic Functions

Here are some of the basic functions that we will need.

Function	What it returns to
`max(x)`	the maximum value of x
`min(x)`	the minimum value of x
`range(x)`	the minimum and maximum values of x
`length(x)`	the number of elements in x
`sum(x)`	the summation of the elements in x
`mean(x)`	the mean value of the elements in x
`sd(x)`	standard deviation of the elements in x
`var(x)`	variance of of the elements in x
`sqrt(x)`	square root of the elements of x
`sort(x)`	put elements in the ascending order (note this is not an assignment)

Examples

```
> max(today_temp_C)
[1] 30
> min(today_temp_C)
[1] 20
> mean(today_temp_C)
[1] 25.5
> sd(today_temp_C)
[1] 4.203173
```

Examples (cont'd)

You can cascade the functions in other functions. For example, the sample standard deviation can be calculated by using the following formula.

$$S = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

You can calculate the formula in R as follows:

```r
sqrt(sum((today_temp_C-mean(today_temp_C))^2)/
       (length(today_temp_C)-1))
```

```
## [1] 4.203173
```

Implicit Coercion

- Each element of a vector is supposed to be of the same data type.
- How about this?

```
> x <- c("apple", "orange", 66)
> typeof(x)
[1] "character"
```

The reason is **implicit coercion**.

Coercion in Vector Creation

If the elements of a vector are specified in different data types, R will coerce them into one data type. Not all the details of **implicit coercion** are within the scope of this course.

Remember a data type is coerced into the other data types when entered in an arithmetic operation as follows.

- **logical:** `logical` $\longmapsto$ `integer` $\longmapsto$ `double`.
- **integer:** `integer` $\longmapsto$ `double`

Coercion in Vector Creation (cont'd)

Similarly, R will implicitly coerce elements of a vector when different data types are used. Elements of a vector with mixed data types will be coerced to highest data type of the data types of the elements.

Coercion from the lowest to the highest data type

`logical` $\longmapsto$ `integer` $\longmapsto$ `double` $\longmapsto$ `character`

Knowing the order would be useful to predict the behavior of R when coercion is unavoidable. First line of defense against implicit coercion in vector creation is to avoid it by not mixing data types. But, if not possible, we should know that the resultant vector's data type will be the data type of the element whose data type is the highest in the order above.

Examples

```
> y <- c(1L, FALSE, 3, 4, 5, "k")
> y
[1] "1"     "FALSE" "3"     "4"     "5"     "k"
> typeof(y)
[1] "character"
> z <- c(TRUE, 9L, 8, 7, FALSE)
> z
[1] 1 9 8 7 0
> typeof(z)
[1] "double"
> w <- c(TRUE, 1L, 2L)
> w
[1] 1 1 2
> typeof(w)
[1] "integer"
```

A rule of thumb:

The type of the elements of a vector is the data type of the element with the highest data type.

The End

Thanks for watching

See you in next time!