Data Science Workshop: Session 1

SCRIPTS Data and Methodology Center

Therese Anders Allison Koh

January 10, 2020

Plan for this workshop series

This workshop series is geared toward learning basic data management in `R`. This includes tasks like manipulating variables, creating new variables, subsetting data, reshaping data, and merging. We will also cover some introductory regular expression applications. In this workshop series we will cover only basic visualization methods in `R`. Aspects like data analysis, web-scraping, or higher-level statistical programming are not covered.

Scheduled sessions:

1. **Introduction to R** (working directories, arithmetic, logical operators, basic indexing, data types, basic functions such as sum, mean, names, seq, rep, installing packages, reading and writing data, dealing with missing data, data frames, indexing on data frames, getting an overview of the data with multivariate numerical and graphical summaries).
2. **Basic data management** (`dplyr`,`tidyr`,`stringr`)

Getting started in `R`

`R` is a programming language for statistical computing and data visualization, that is a open source alternative to commercial statistical packages such as Stata or SPSS. `R` is maintained and developed by a vibrant community of programmers and statisticians and offers many user-written packages to extend basic functionality.

In this workshop, we will be using `R` together with the integrated development environment (IDE) **RStudio**. In addition to offering a 'cleaner' programming development than the basic `R` editor, RStudio offers a large number of added functionalities for integrating code into documents, built-in tools and web-development. To get started, please download the latest version of RStudio and `R` from this website:

https://www.rstudio.com/products/rstudio/download/

Getting Help

The key to learning `R` is: **Google**! This workshop will give you an overview over basic `R` functions, but to really learn `R` you will have to actively use it yourself, trouble shoot, ask questions, and google! The `R` mailing list and other help pages such as http://stackoverflow.com offer a rich archive of questions and answers by the `R` community. For example, if you google "recode data in r" you will find a variety of useful websites explaining how to do this on the first page of the search results. Also, don't be surprised if you find a variety of different ways to execute the same task.

RStudio also has a useful help menu. In addition, you can get information on any function or integrated data set in R through the console, for example:

```
?plot
```

In addition, there are a lot of free R comprehensive guides, such as Quick-R at http://www.statmethods.net or the R cookbook at http://www.cookbook-r.com.

Working Directories

R needs to know where to look for files if you want to read data and where to store files if you write data. This includes the code you will be typing in this session. The `getwd()` command returns the current working directory. We can change the working directory with `setwd()` (see below).

Think of your computer as a filing cabinet. During the course of this workshop, you will be writing a number of R scripts, that are essentially text files with commands for R. In order to execute these files, we need to tell R where to look for the list of commands we want to execute. Setting a working directory is analogous to telling R in which file in the filing cabinet we stored our document (code) and into which file in the filing cabinet to put new documents (such as graphs, new data frames, new code).

```
getwd() # Prints the current working directory
```

```
## [1] "/cloud/project/Session 1"
```

```
setwd("/cloud/project/Session 1")
```

Important for Windows users: In R, the backslash is an escape character (we will be talking more about this in the session about `stringr`). Therefore, entering file paths is a little different in Windows than on a Mac. On a windows machine you would enter:

```
setwd("C:/Documents and Settings/Data")
```

OR

```
setwd("C:\\Documents and Settings\\Data")
```

Arithmetic in R

You can use R as a calculator!

	Operator	Example
Addition	`+`	`2+4`
Subtraction	`-`	`2-4`
Multiplication	`*`	`2*4`
Division	`/`	`4/2`
Exponentiation	`^`	`2^4`
Square Root	`sqrt()`	`sqrt(144)`
Absolute Value	`abs()`	`abs(-4)`

```
4*9
```

```
## [1] 36
```

```
sqrt(144)
```

```
## [1] 12
```

Just like any regular calculator, you have to pay attention to the order of operations! Example:

```
6 * 8 - sqrt(7) + abs(-10) * (4/5)
```

```
## [1] 53.35425
```

```
6 * (8 - sqrt(7)) + abs(-10) * (4/5)
```

```
## [1] 40.12549
```

Logical operators

	Operator
Less than	`<`
Less than or equal to	`<=`
Greater than	`>`
Greater than or equal to	`>=`
Exactly equal to	`==`
Not equal to	`!=`
Not `x`	`!x`
`x` or `y`	`x | y`
`x` and `y`	`x & y`

Logical operators are incredibly helpful for any type of exploratory analysis, data cleaning and/or visualization task.

```
4 > 2
```

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

```
4 <= 2
```

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

Objects in R

Assigning values to objects

`R` stores information as an *object*. You can name objects whatever you like. Just remember to not use names that are reserved for build-in functions or functions in the packages you use, such as `sum`, `mean`, or `abs`. Most of the time, `R` will let you use these as names, but it leads to confusion in your code.

A few things to remember

- Do not use special characters such as `$` or `%`. Common symbols that are used in variable names include `.` or `_`.
- Remember that `R` is case sensitive.
- To assign values to objects, we use the assignment operator `<-`. Sometimes you will also see `=` as the assignment operator. This is a matter of preference and subject to debate among `R` programmers. Personally, I use `<-` to assign values to objects and `=` within functions.
- The `#` symbol is used for commenting and demarcation. Any code following `#` will not be executed.

Below, R stores the result of the calculation in an object named `result`. We can access the value by referring to the object name.

```
result <- 5/3
result # Implicitly printing the output
```

```
## [1] 1.666667
```

```
print(result) # Explicitly printing the output
```

```
## [1] 1.666667
```

If we assign a different value to the object, the value of the object will be changed.

```
result <- 5-3
result
```

```
## [1] 2
```

Vectors

R can deal with a variety of data types, including vectors, scalars, matrices, data frames, factors, and lists. Today, we will focus on vectors.

A **vector** is one of 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 2017: 14). It is essentially a list of data of a single type (either numerical, character, or logical). 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 the five most populous countries in 2016. The data come from the World Bank.

```
pop1 <- c(1379, 1324, 323, 261, 208)
pop1
```

```
## [1] 1379 1324  323  261  208
```

We can 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 1324  323  261  208  194  187  161  142  127
```

Variable types

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

- **Numerical**: Any number. **Integer** is a numerical variable without any decimals.
- **Character**: This is what Stata (and other programming languages such as Python) calls a string. 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.

First, lets check which variable type our population data were stored in. The output below 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 1324 323 261 208 ...
```

Suppose, we wanted to add information on the country names. We can enter these 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. Note that `R` recognizes both `TRUE` and `T` (and `FALSE` and `F`) as logical values.

```
asia <- c(TRUE, TRUE, F, T, F)
str(asia)
```

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

Lastly, we define a factor variable for the regime type of a country in 2016. This variable can take on one of four values (based on data from the Economist Intelligence Unit): Full Democracy, Flawed Democracy, Hybrid Regimes, and Autocracy. 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
```

Data types are important! `R` will not perform certain operations if you don't get the variable type right. The good news is that we can switch between data types. This can sometimes be tricky, especially when you are switching from a factor to a numerical type[1]. We won't go into this too much here; just remember: Google is your friend!

Let's convert the factor variable `regime` into a character. Also, for practice, lets convert the `asia` variable to character and back to logical.

```
regime <- as.character(regime)
str(regime)
```

```
##  chr [1:5] "Autocracy" "FlawedDem" "FullDem" "FlawedDem" "FlawedDem"
```

```
asia <- as.character(asia)
str(asia)
```

```
##  chr [1:5] "TRUE" "TRUE" "FALSE" "TRUE" "FALSE"
```

```
asia <- as.logical(asia)
str(asia)
```

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

Exercise 1: Why won't `R` let us do the following?

```
no_good <- (a,b,c)
```

```
no_good_either <- c(one, two, three)
```

Exercise 2: What's the difference? (Bonus: What do you think is the class of the output vector?)

[1] Sometimes you have to do a work around, like switching to a character first, and then converting the character to numeric. You can concatenate commands: `myvar <- as.numeric(as.character(myvar))`.

```
diff <-c(TRUE,"TRUE")
```

Exercise 3: What is the class of the following vector?

```
vec <- c("1", "2", "3")
```

Vector operations

You can do a variety of things like have `R` print out particular values or ranges of values in a vector, replace values, add additional values, etc. We will not get into all of these operations today, but be aware that (for all practical purposes) if you can think of a vector manipulation operation, `R` can probably do it.

We can do arithmatic operations on vectors! Let's use the vector of population counts we created earlier and double it.

```
pop1
```

```
## [1] 1379 1324  323  261  208
```

```
pop1_double <- pop1 * 2
pop1_double
```

```
## [1] 2758 2648  646  522  416
```

Exercise 4: What do you think this will do?

```
pop1 + pop2
```

Exercise 5: And this?

```
pop_c <- c(pop1, pop2)
```

Functions

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 varianve of the values or object.
`sd()`	Returns the varianve of the values or object.

```
min(pop)
```

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

```
max(pop)
```

```
## [1] 1379
```

```
mean(pop)
```

```
## [1] 430.6
```

Exercise 6: Using functions in `R`, how else could we compute the mean population value?

```
## [1] 430.6
```

Accessing elements of vectors

There are many ways to access elements that are stored in an object. Here, we will focus on a method called *indexing*, using square brackets as an operator.

Below, we use square brackets and the index `1` to access the first element of the top 5 population vector and the corresponding country name vector.

```
pop1[1]
```

```
## [1] 1379
```

```
cname[1]
```

```
## [1] "CHN"
```

We can use indexing to access multiple elements of a vector. For example, below we use indexing to implicitly print the second and fifth elements of the population and the country name vectors, respectively.

```
pop[c(2,5)]
```

```
## [1] 1324  208
```

```
cname[c(2,5)]
```

```
## [1] "IND" "BRA"
```

We can assign the first element of the population vector to a new object called `first`.

```
first <- pop[1]
```

Below, we make a copy of the country name vector and delete the *last* element. Note, that we can use the `length()` function to achieve the highest level of *generalizability* in our code. Using `length()`, we do not need to know the index of the last element of out vector to drop the last element.

```
cname_copy <- cname
## Option 1: Dropping the 5th element
cname_copy[-5]
```

```
## [1] "CHN" "IND" "USA" "IDN"
```

```
## Option 2 (for generalizability): Getting the last element and dropping it.
length(cname_copy)
```

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

```
cname_copy[-length(cname_copy)]
```

```
## [1] "CHN" "IND" "USA" "IDN"
```

Indexing can be used to alter values in a vector. Suppose, we notice that we wrongly entered the fifth element of the regime type vector (or the regime type changed).

```
regime
```

```
## [1] "Autocracy" "FlawedDem" "FullDem"   "FlawedDem" "FlawedDem"
```

```
regime[5] <- "FullDem"
regime
```

```
## [1] "Autocracy" "FlawedDem" "FullDem"   "FlawedDem" "FullDem"
```

Exercise 7: We made even more mistakes when entering the data! We want to subtract 10 from the third and fifth element of the top 5 population vector. *How would you do it*?

More functions

The myriad of functions that are either built-in to base `R` or parts of user-written packages are the greatest stength of `R`. For most applications we encounter in our daily programming practice, `R` already has a function, or someone smart wrote one. Below, we introduce a few additional helpful functions from base `R`.

	Function
`seq()`	Returns sequence from input1 to input2 by input3.
`rep()`	Repeats input1 input2 number of times.
`names()`	Returns the names (labels) of objects.
`which()`	Returns the index of objects.

Le't's create a vector of indices for our top 5 population data.

```
cindex <- seq(from = 1, to = length(pop1), by = 1)
cindex
```

```
## [1] 1 2 3 4 5
```

We can use the `rep()` function to repeat data.

```
rep(30, 5)
```

```
## [1] 30 30 30 30 30
```

Exercise 8 Use `rep()` and `seq` to print the following data.

```
##  [1] 1.0 1.5 2.0 2.5 3.0 1.0 1.5 2.0 2.5 3.0
```

Suppose, we wanted to record whether we had completed the data collection process for the top 10 most populous countries. First, suppose we completed the process on every second country.

```
completed <- rep(c("yes","no"), 5)
completed
```

```
##  [1] "yes" "no"  "yes" "no"  "yes" "no"  "yes" "no"  "yes" "no"
```

Now suppose that we have completed the data collection process for the first 5 countries, but not the latter 5 countries (we don't have their names, location, or regime type yet).

```
completed2 <- rep(c("yes","no"), each = 5)
completed2
```

```
##  [1] "yes" "yes" "yes" "yes" "yes" "no"  "no"  "no"  "no"  "no"
```

We can give our data informative labels. Let's use the country names vector as labels for our top 5 population vector.

```
names(pop1)
```

```
## NULL
```

```
cname
```

```
## [1] "CHN" "IND" "USA" "IDN" "BRA"
```

```
names(pop1) <- cname
names(pop1)
```

```
## [1] "CHN" "IND" "USA" "IDN" "BRA"
```

```
pop1
```

```
##  CHN  IND  USA  IDN  BRA
## 1379 1324  323  261  208
```

We can use labels to access data using indexing and logical operators. Suppose, we wanted to access the population count for Brazil in our top 5 population data.

```
pop1[names(pop1) == "BRA"]
```

```
## BRA
## 208
```

Exercise 9 Access all top 5 population ratings that are greater or equal than the mean value of population ratings.

```
## [1] 699
```

```
##  CHN  IND
## 1379 1324
```

Exercise 10 Access all top 5 population ratings that are less than the population of the most populous country, but not the US.

```
##  IND  IDN  BRA
## 1324  261  208
```

Operating on multiple vectors simultaneously

We did not work with data frames yet, but remember that our data input is ordered. The first element of the `pop1` vector corresponds with the first element of the `cname`, `regime`, and `asia` vectors. We can use this to run more sophisticated queries on our data.

Suppose, we wanted to know the regime type of Indonesia. Given that our vectors are ordered, we can use indexing to extract the data. First, lets see what happens if we run a simple logical query.

```
cname == "IDN"
```

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

```
regime[cname == "IDN"]
```

```
## [1] "FlawedDem"
```

We can also use the `which()` function that returns the index of the vector element.

```
which(cname == "IDN")
```

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

```
regime[which(cname == "IDN")]
```

```
## [1] "FlawedDem"
```

Exercise 11 Print out the population count for all Asian countries within the top 5 most populous countries that are not autocracies.

```
##  IND  IDN
## 1324  261
```

Sources

Economist Intelligence Unit (2017): *Democracy Index*. https://infographics.economist.com/2017/DemocracyIndex/.

Imai, Kosuke (2017): *Quantitative Social Science. An Introduction*. Princeton and Oxford: Princeton University Press.

World Bank (2017): *Population, total*. https://data.worldbank.org/indicator/sp.pop.totl?end=2016&start=2015.