Hertie School/SCRIPTS Data Science Workshop Series

Session 1: A gentle introduction to base R and RStudio

*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 numerical and graphical summaries).
2. **Modern data management in R using the tidyverse** (`dplyr`, `tidyr`, `readr`, and `lubridate` packages)

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:

---

*Instructor, Hertie School/SCRIPTS, anders@hertie-school.org.

†Teaching assistant, Hertie School, kohallison3@gmail.com.

```
?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] "/Users/thereseanders/Dropbox/Dissertation/TerritorialControl/Analysis/tc_hmm_decay_191011/ds-wo:
```

```
#setwd("/cloud/project")
```

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:

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

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

```r
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.

```r
4 > 2
```

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

```r
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.

```r
result <- 5/3
result
```

```
## [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.

```r
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.

```r
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.

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

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

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

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

```r
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?

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

```r
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?)

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

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

```r
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.

---

[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))`.

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.

Let'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
```

Working with data frames in `R`

Importing data

Most data formats we commonly use are not native to `R` and need to be imported. Luckily, there is a variety of packages available to import just about any non-native format. One of the essential libraries is called `foreign` and includes functions to import .csv, .dta (Stata), .dat, ,.sav (SPSS), etc. The `foreign` package is pre-installed. However, we need to tell `R` that we want to use functions from this package by calling it into the working environment using the `library()` command.

```r
library(foreign)
```

In this example, we will use data from the Armed Conflict Location & Event Data Project (ACLED), which offers real-time data and analysis on political violence and protests around the world. This dataset includes riots and protests from January 2010 to December 2019.

When reading the data, remember that you need to either specify the complete file name or change your working directory using `setwd()` to the folder in which you saved the data.

```r
mydata <- read.csv("ACLED_countries.csv")
class(mydata)
```

```
## [1] "data.frame"
```

Dimensions of a data frame

Let's find out what this data looks like. First, use the `str()` function to explore the variable names and which data class they are stored in. Note: `int` stands for `integer` and is a special case of the class `numeric`.

```r
str(mydata)
```

```
## 'data.frame':    103 obs. of  5 variables:
##  $ country               : Factor w/ 102 levels "Afghanistan",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ region                : Factor w/ 10 levels "Caucasus and Central Asia",..: 9 3 6 4 1 1 5 9 3 10
##  $ nconflicts            : int  40765 582 7362 1108 3118 12627 1861 16802 293 215 ...
##  $ nconflict_no_fatalities: int  23946 581 5321 728 3110 12578 1846 13114 293 165 ...
##  $ fatalities            : int  119973 1 8451 13788 8 59 22 4640 0 114 ...
```

If we are only interested in what the variables are called, we can use the `names()` function.

```r
names(mydata)
```

```
## [1] "country"                "region"
## [3] "nconflicts"             "nconflict_no_fatalities"
## [5] "fatalities"
```

As we saw in the last session, we can alter the names of vectors by using the `names()` function and indexing. Because data frames are essentially just combinations of vectors, we can do the same for variable names inside data frames. Suppose we want to change the variable `nconflicts`.

```r
names(mydata)[3] <- "nconflict"
names(mydata)
```

```
## [1] "country"                "region"
## [3] "nconflict"              "nconflict_no_fatalities"
## [5] "fatalities"
```

We can use the `summary()` function to get a first look at the data.

```r
summary(mydata)
```

```
##        country                          region      nconflict
##  Tunisia    : 2   Europe                   :15   Min.   :    1.0
##  Afghanistan: 1   Middle East              :15   1st Qu.:  315.5
##  Albania    : 1   Western Africa           :15   Median : 1250.0
##  Algeria    : 1   Eastern Africa           :13   Mean   : 6216.3
##  Angola     : 1   Caucasus and Central Asia: 8   3rd Qu.: 5993.5
##  Armenia    : 1   Middle Africa            : 8   Max.   :70734.0
##  (Other)    :96   (Other)                  :29
```

10

```
##  nconflict_no_fatalities   fatalities
##  Min.   :    1        Min.   :      0.0
##  1st Qu.:  289        1st Qu.:     17.5
##  Median : 1037        Median :    236.0
##  Mean   : 4667        Mean   :   9543.3
##  3rd Qu.: 4536        3rd Qu.:   7020.0
##  Max.   :63665        Max.   :119973.0
##
```

A data frame has two dimensions: rows and columns.

```
nrow(mydata) # Number of rows
```

```
## [1] 103
```

```
ncol(mydata) # Number of columns
```

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

```
dim(mydata) # Rows first then columns.
```

```
## [1] 103   5
```

Accessing elements of a data frame

As a rule, whenever we use two-dimensional indexing in R, the order is: `[row,column]`. To access the first row of the data frame, we specify the row we want to see and leave the column slot following the comma empty.

```
mydata[1,]
```

```
##        country       region nconflict nconflict_no_fatalities fatalities
## 1 Afghanistan Southern Asia     40765                   23946     119973
```

We can use the concatenate function `c()` to access multiple rows (or columns) at once. Below we print out the first and second row of the dataframe.

```
mydata[c(1,2),]
```

```
##        country       region nconflict nconflict_no_fatalities fatalities
## 1 Afghanistan Southern Asia     40765                   23946     119973
## 2      Albania       Europe       582                     581          1
```

We can also access a range of rows by separating the minimum and maximum value with a `:`. Below we print out the first five rows of the dataframe.

```
mydata[1:5,]
```

```
##        country                    region nconflict nconflict_no_fatalities
## 1 Afghanistan             Southern Asia     40765                   23946
## 2      Albania                    Europe       582                     581
## 3      Algeria           Northern Africa      7362                    5321
## 4       Angola             Middle Africa      1108                     728
## 5      Armenia Caucasus and Central Asia      3118                    3110
##   fatalities
## 1     119973
## 2          1
## 3       8451
## 4      13788
## 5          8
```

11

If we try to access a data point that is out of bounds, R returns the value NULL.

```
mydata[3,3]
```

```
## [1] 7362
```

Exercise 1 Access the element of the data frame `mydata` that is stored in row 1, column 1.

```
## [1] Afghanistan
## 102 Levels: Afghanistan Albania Algeria Angola Armenia ... Zimbabwe
```

Exercise 2 Access the element of the data frame `mydata` that is stored in column 2, row 3.

```
## [1] Northern Africa
## 10 Levels: Caucasus and Central Asia Eastern Africa ... Western Africa
```

The $ operator

The $ operator in R is used to specify a variable within a data frame. This is an alternative to indexing.

```
mydata$nconflict
```

```
##   [1] 40765    582   7362   1108   3118  12627   1861  16802    293    215    459
##  [12]     52    765   2105   7760   1919   2621   4625    912    411    312  14519
##  [23]    112  10260     50    191    190   5122    215    197   1446    858   1312
##  [34]   1121    151  67561   3371   5708  22354   1580   1838    491    374   6427
##  [45]    289     39    319     41   2956     83   1294   8857   1365    512    384
##  [56]   3193    598    444    213   2177   1250   9070    682   5143    971  15011
##  [67]    275      9  54496   6568   9690      5    243   1134   3734    370   6096
##  [78]    963   1275   1079  29667  11667   6227      2   3855  12528  70734     68
##  [89]    782   7420    321   5756      1  10334     21   4866  30131      4    102
## [100]    503  45298   1148   5891
```

Exercise 3: How would you access all elements of the variable `country` using indexing rather than the $ operator?

```
##  [1] Afghanistan               Albania
##  [3] Algeria                   Angola
##  [5] Armenia                   Azerbaijan
##  [7] Bahrain                   Bangladesh
##  [9] Belarus                   Benin
## [11] Bosnia and Herzegovina    Botswana
## [13] Bulgaria                  Burkina Faso
## [15] Burundi                   Cambodia
## [17] Cameroon                  Central African Republic
## [19] Chad                      Croatia
## [21] Cyprus                    Democratic Republic of Congo
## [23] Djibouti                  Egypt
## [25] Equatorial Guinea         Eritrea
## [27] eSwatini                  Ethiopia
## [29] Gabon                     Gambia
## [31] Georgia                   Ghana
## [33] Greece                    Guinea
## [35] Guinea-Bissau             India
## [37] Indonesia                 Iran
## [39] Iraq                      Israel
## [41] Ivory Coast               Jordan
## [43] Kazakhstan                Kenya
## [45] Kosovo                    Kuwait
```

```
##  [47] Kyrgyzstan             Laos
##  [49] Lebanon                Lesotho
##  [51] Liberia                Libya
##  [53] Madagascar             Malawi
##  [55] Malaysia               Mali
##  [57] Mauritania             Moldova
##  [59] Montenegro             Morocco
##  [61] Mozambique             Myanmar
##  [63] Namibia                Nepal
##  [65] Niger                  Nigeria
##  [67] North Macedonia        Oman
##  [69] Pakistan               Palestine
##  [71] Philippines            Qatar
##  [73] Republic of Congo      Romania
##  [75] Russia                 Rwanda
##  [77] Saudi Arabia           Senegal
##  [79] Serbia                 Sierra Leone
##  [81] Somalia                South Africa
##  [83] South Sudan            Spain
##  [85] Sri Lanka              Sudan
##  [87] Syria                  Tajikistan
##  [89] Tanzania               Thailand
##  [91] Togo                   Tunisia
##  [93] Tunisia                Turkey
##  [95] Turkmenistan           Uganda
##  [97] Ukraine                United Arab Emirates
##  [99] Uzbekistan             Vietnam
## [101] Yemen                  Zambia
## [103] Zimbabwe
## 102 Levels: Afghanistan Albania Algeria Angola Armenia ... Zimbabwe

##   [1] Afghanistan            Albania
##   [3] Algeria                Angola
##   [5] Armenia                Azerbaijan
##   [7] Bahrain                Bangladesh
##   [9] Belarus                Benin
##  [11] Bosnia and Herzegovina Botswana
##  [13] Bulgaria               Burkina Faso
##  [15] Burundi                Cambodia
##  [17] Cameroon               Central African Republic
##  [19] Chad                   Croatia
##  [21] Cyprus                 Democratic Republic of Congo
##  [23] Djibouti               Egypt
##  [25] Equatorial Guinea      Eritrea
##  [27] eSwatini               Ethiopia
##  [29] Gabon                  Gambia
##  [31] Georgia                Ghana
##  [33] Greece                 Guinea
##  [35] Guinea-Bissau          India
##  [37] Indonesia              Iran
##  [39] Iraq                   Israel
##  [41] Ivory Coast            Jordan
##  [43] Kazakhstan             Kenya
##  [45] Kosovo                 Kuwait
```

```
##  [47] Kyrgyzstan               Laos
##  [49] Lebanon                  Lesotho
##  [51] Liberia                  Libya
##  [53] Madagascar               Malawi
##  [55] Malaysia                 Mali
##  [57] Mauritania               Moldova
##  [59] Montenegro               Morocco
##  [61] Mozambique               Myanmar
##  [63] Namibia                  Nepal
##  [65] Niger                    Nigeria
##  [67] North Macedonia          Oman
##  [69] Pakistan                 Palestine
##  [71] Philippines              Qatar
##  [73] Republic of Congo        Romania
##  [75] Russia                   Rwanda
##  [77] Saudi Arabia             Senegal
##  [79] Serbia                   Sierra Leone
##  [81] Somalia                  South Africa
##  [83] South Sudan              Spain
##  [85] Sri Lanka                Sudan
##  [87] Syria                    Tajikistan
##  [89] Tanzania                 Thailand
##  [91] Togo                     Tunisia
##  [93] Tunisia                  Turkey
##  [95] Turkmenistan             Uganda
##  [97] Ukraine                  United Arab Emirates
##  [99] Uzbekistan               Vietnam
## [101] Yemen                    Zambia
## [103] Zimbabwe
## 102 Levels: Afghanistan Albania Algeria Angola Armenia ... Zimbabwe
```

Exercise 4: Print out every second element from the variable `nconflict` using indexing methods and the sequence function `seq()`.

```
##  [1] 40765  7362  3118  1861   293   459   765  7760  2621   912   312
## [12]   112    50   190   215  1446  1312   151  3371 22354  1838   374
## [23]   289   319  2956  1294  1365   384   598   213  1250   682   971
## [34]   275 54496  9690   243  3734  6096  1275 29667  6227  3855 70734
## [45]   782   321     1    21 30131   102 45298  5891
```

Exercise 5: What are two ways to find the mean value of the variable `nconflict` using indexing (i.e. not using the $ operator)?

```
## Warning in mean.default(mydata[, 2]): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

```
## [1] 6216.272
```

Exercise 6: How would you find the maximum value for number of conflicts using the $ operator?

```
## [1] 70734
```

Exercise 7: Print the country that corresponds to the maximum world population value using the $ operator and indexing!

```
## [1] Syria
## 102 Levels: Afghanistan Albania Algeria Angola Armenia ... Zimbabwe
```

NAs in `R`

`NA` is how `R` denotes missing values. For certain functions, `NA`s cause problems.

```r
vec <- c(4, 1, 2, NA, 3)
mean(vec) #Result is NA!
```

```
## [1] NA
```

```r
sum(vec) #Result is NA!
```

```
## [1] NA
```

We can tell `R` to remove the NA and execute the function on the remainder of the data.

```r
mean(vec, na.rm = T)
```

```
## [1] 2.5
```

```r
sum(vec, na.rm = T)
```

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

Adding observations

First, lets add another observation to the data. Suppose we wanted to add an observation for Germany, which will be a missing value. We can use the same operations we used for vectors to add data. Here, we will will use the `rbind()` function to do so. `rbind()` stands for "row bind." Save the output in a new data frame!

```r
obs <- c("Germany", NA, NA)
mydata_new <- rbind(mydata, obs)
```

```
## Warning in `[<-.factor`(`*tmp*`, ri, value = "Germany"): invalid factor
## level, NA generated
```

```r
dim(mydata_new)
```

```
## [1] 104    5
```

We can also create new variables that use information from the existing data. If we know the number of conflicts without fatalities by country, we can calculate the number of conflicts *with* fatalities to generate the variable `nconflict_fatalities`. By using the `$` operator, we can directly assign the new variable to the data frame `mydata_new`.

```r
mydata$nconflict_fatalities <- mydata$nconflict - mydata$nconflict_no_fatalities
head(mydata, 3) #prints out the first 3 rows of the data frame
```

```
##       country          region nconflict nconflict_no_fatalities fatalities
## 1 Afghanistan   Southern Asia     40765                   23946     119973
## 2     Albania          Europe       582                     581          1
## 3     Algeria Northern Africa      7362                    5321       8451
##   nconflict_fatalities
## 1                16819
## 2                    1
## 3                 2041
```

Saving data

Suppose we wanted to save this newly created data frame. We have multiple options to do so. If we wanted to save it as a native `.RData` format, we would run the following command.

```
# Make sure you specified the right working directory!
# save(mydata, file = "mydata_new.RData")
```

Most of the time, however, we would want to save our data in formats that can be read by other programs as well. .csv is an obvious choice. After saving the new file, check the folder that is your working directory to see whether the newly saved file shows up there!

```
# write.csv(mydata_new, file = "mydata_new.csv")
```

(Very basic) data visualization

Today, we will be covering some basics of data visualization in R using the native plotting functions. For more advanced data visualization functions, see the `ggplot2` package and the related material for a two-session introductory workshop on data visualization https://github.com/thereseanders/Workshop-Intro-to-ggplot2.

Basic graphical summaries of data

Type	Operator
Histogram	`hist()`
Stem and leaf plot	`stem()`
Boxplot	`boxplot()`
Kernel density plot	`plot(density())`
Basic scatterplot	`plot()`

Boxplot of population density

We can get an overview of the number of conflicts per country using the `boxplot()` function.

```
boxplot(mydata$nconflict)
```
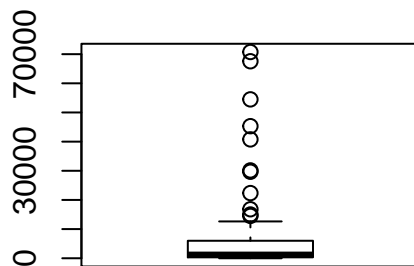

Suppose we wanted to know whether there are more conflicts with fatalities on average for countries with a higher overall number of conflicts. Let's first look at the distribution of number of conflicts.

```
summary(mydata$nconflict)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0   315.5  1250.0  6216.3  5993.5 70734.0
```

Create a new dummy variable that codes whether a state has a relatively high number of conflicts (greater than the median).

```
median(mydata$fatalities)
```

```
## [1] 236
```

```
mydata$fatalities_high <- ifelse(mydata$fatalities > median(mydata$fatalities), 1, 0)
head(mydata$fatalities_high)
```
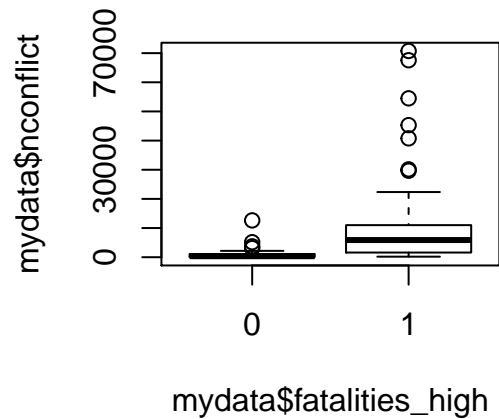
```
## [1] 1 0 1 1 0 0
```

```
table(mydata$fatalities_high) # We split the observations exactly in half.
```

```
##
##  0  1
## 52 51
```

We can display two indicators in the same boxplot. We can use this feature to answer the question whether larger states are on average more or less dense than smaller states.
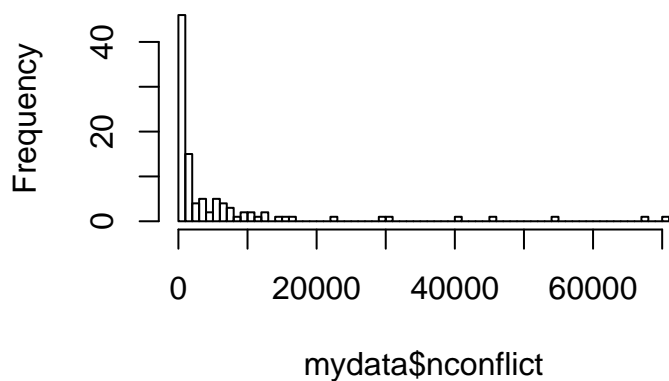
```
boxplot(mydata$nconflict ~ mydata$fatalities_high)
```


Histogram of number of conflicts

```
hist(mydata$nconflict,breaks=100)
```
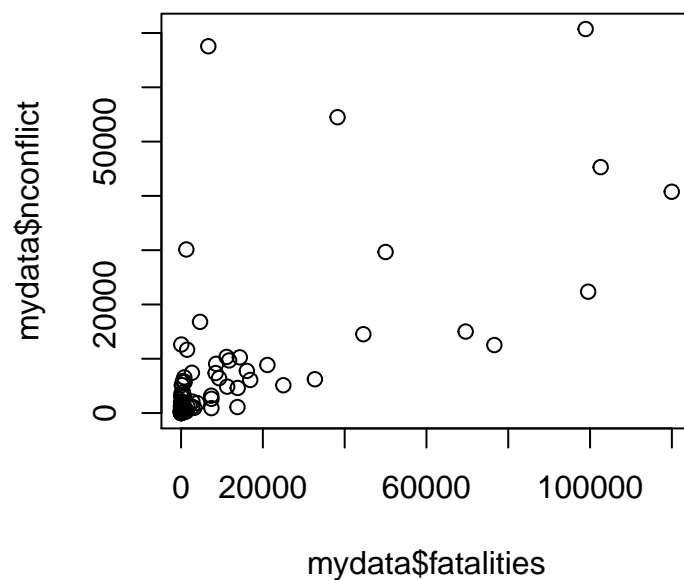

Density plot of fatalities

```
plot(density(mydata$fatalities))
```

density.default(x = mydata$fatalities)

N = 103   Bandwidth = 1861
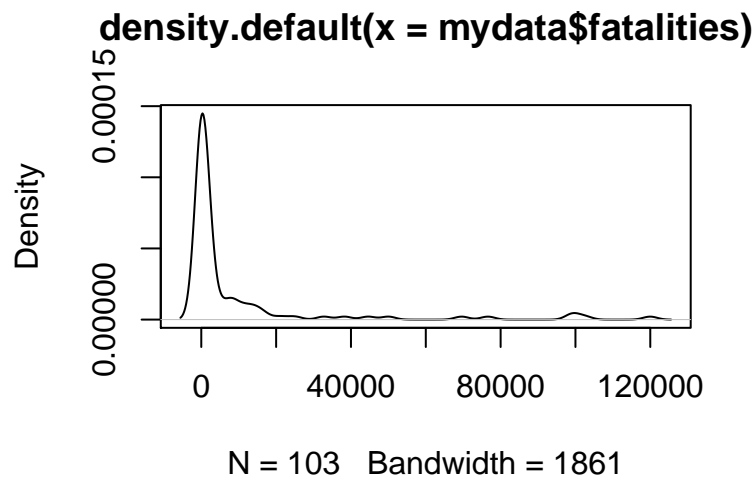
Basic scatter plots

How does population size vary with area? Are the correlated at all?

```
plot(mydata$fatalities,mydata$nconflict)
```


Basic graphic options

Aside from arguably not being very informative, the plot above is not very pretty. Lets give it titles, use color and shapes!

```
plot(mydata$fatalities,mydata$nconflict,
    main = "ACLED (2010-2019)", #Adding a main title.
    xlab = "Fatalities", #Adding a x-axis title.
    ylab = "Number of Conflicts", #Adding a y-axis title.
    col = "tomato", #Changing the color of the data points.
    pch = 18) #Changing the shape
```

ACLED (2010–2019)
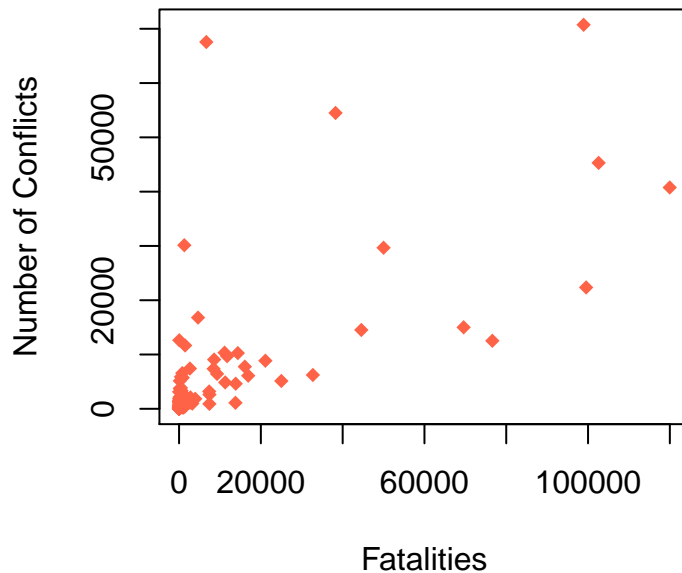
Yeah, ok. Its not much prettier (especially the labeling on the axes), but you get the point. . .

A few additional notes on graphical options:

- `R` can display any color in the RBG or HEX system. However it also has a ton of colors that you can just refer to by name, see http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.
- Same with the shapes and line types, see http://www.cookbook-r.com/Graphs/Shapes_and_line_types/.
- `R` colors in all their glory: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.

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.

Raleigh, Clionadh, Andrew Linke, Håvard Hegre and Joakim Karlsen. 2010. Introducing ACLED – Armed Conflict Location and Event Data. *Journal of Peace Research* 47(5), 651-660.

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