Introduction to R

Leonardo Carella ([leonardo.carella@mansfield.ox.ac.uk](mailto:leonardo.carella@mansfield.ox.ac.uk))

29/03/2022

Welcome to the Oxford Spring School in Advanced Research Methods!

In this preliminary class, we are going to learn how to operate in R, the software that will be used in most of the courses you will take. Whether you are a complete novice to computational social science, a Stata/SPSS user, or you’ve learnt R before and just need a refresher, this class will take you through the basics, from zero to regressions. No prior knowledge of the software is assumed.

#### 1. Installing R and R Studio

The first thing you need to do is to download **R and R Studio**.

* Step 1: download R itself, through <https://www.r-project.org/> (you will need to select a CRAN mirror depending on your location), or go directly to the UK mirror through <https://www.stats.bris.ac.uk/R/>. Select your operating system (Mac, Windows, and Linux are supported), and the version of R you want to download, depending on which version of the operating system your computer runs on.
* Step 2: download RStudio, through <https://rstudio.com/products/rstudio/download/>. “RStudio Desktop – Open source license” is what you want. Select the version of RStudio you want to download, depending on which version of your operating system your computer is running on.

What are these things, by the way?

* **R** is a programming language and environment for statistical computing. It’s one of the most popular softwares for data science in academia and increasingly in the industry. It’s free, open source, and collaborative: developers around the world are constantly increasing R’s functionalities, making them directly available to users through packages, which are also free to install (more about that later on in the class).
* **R Studio** is an Integrated Development Environment for R: a front-end that runs your code through R, allowing you to do so from a user-friendly interface instead of coding through R’s rather basic console. All the coding in this session and in the Spring School courses will be done in RStudio, but **you need to have R installed for RStudio to work**.

Once you’ve installed R and RStudio, **open RStudio** (not R), and let’s get to work.

#### 2. Basic Syntax: Operators, Functions, and Objects

First thing, click on the “New File” icon (the green plus on a white square) in the top-right corner of your screen and select “R Script”. This will be the blank canvas where we type in our code. If you want, you can also open the Intro\_to\_R\_script.R file that comes with the rest of the material. I’ve written there all the code that we are going to learn today. (Though I would suggest you to type out the code yourself rather than copy-pasting it from there.)

At its most basic, you can use R as a calculator: type out arithmetic operations in the *script* section (the field in the top left of the interface), highlight the chunk of code you want to run (say, 2+2), then press the “Run” button. You will get the result in the *console* section (the field in the bottom left of the RStudio interface). You can also tell R to run your code with command+shift instead of pressing “Run”.

2+2

R evaluates mathematical operators in the usual order: Parentheses first, then Exponentiation, then Multiplication, then Division, then Addition, and then Subtraction (PEDMAS). Only use round parentheses (brackets and braces are used for different purposes in R), and use the dot as decimal separator instead of commas, as it’s the convention in English-speaking countries.

-3\*5  
(20-6) / 2  
7^2  
64^(0.5)

This is a good place to introduce your first *functions*. Functions in R are expressions that come with parentheses and within parentheses you pass one or more arguments. These are some simple mathematical functions, where you pass one numerical argument and they return the result of an operation:

abs(-10) #computes the absolute value  
sqrt(81) #computes the square root  
log(4) #computes natural logarithm  
exp(0.5) #computes the exponential function (e to the power of a number)

In R, you can write down notes to your code by using the commenting symbol # before your text. R will not run anything that follows a hashtag on a line. In all other cases, R is not line-sensitive: you can space your code across lines, and as long as you select the whole chunk, it will still recognise it as a continuous block of code. Note also that R is not whitespace-sensitive: 2+2 and 2 + 2 will return the same answer (try it).

Sometimes you have to pass more than one argument to the function. Use commas to separate the arguments:

log(8, base = 2) #computes the logarithm of a number in base 2  
round(123.456789, digits = 2) #rounds a number to the 2nd decimal place

If you want to know more about what a function does, you can type ? and then the function, and run the code. Or indeed, use the help() function. This call will open the *help* window in the bottom right of your interface, with descriptions and examples of how the function can be used

?log  
?round

You can nest functions and operations within each other in a single expression using parentheses:

sqrt(10\*5-1)   
round(log(5+5)\*2, digits = 3)

It is often useful to “store” values as named objects, using the **all-important** <- (assign) operator. These values can be numbers or text strings; in the latter case you have to surround the text with quote marks.

x <- 7  
y <- 4+4  
my\_name <- "Leo"

When you assign values to a named object, R does not return the value in the *console*, but it will appear in the *global environment* window in the top right of the interface. This means that you have successfully stored this value, and at any point you can simply run your object’s label (e.g. my\_name, or x) and R will print the value you have assigned to it. (Also note the underscore in my\_name: when you name objects in R, the names must be continuous text.)

You can pass these named objects as arguments of functions or operations, just as we did numbers:

sqrt(x)  
(x\*y)/2

Obviously don’t try to calculate the square root of your name, because you will get an error:

sqrt(my\_name)

#### 3. The Building Blocs: Vectors and Dataframes

We normally work with variables taking a series of values across a number of observations rather than a single quantity. R handles these ordered sequences of values as *vectors*, and you can create your own vectors using the c() function (c stands for combine or concatenate, there is some controversy in the R community):

zero\_to\_ten <- c(0,1,2,3,4,5,6,7,8,9,10)  
friends <- c("Rachel", "Monica", "Joey", "Chandler", "Ross", "Phoebe")

Indeed, you have already worked with vectors before: the named objects x, y and my\_course were vectors of length 1, because they contained only one element. Now we have created two longer vectors: the integer vector zero\_to\_ten of length 11, and the character vector (also known as a string) friends of length 6.

Here are some other ways to create vectors:

one\_to\_onehundred <- 1:100  
#creates a vector with all integers between 1 and 100  
  
decimals <- seq(0, 1, by = 0.1)   
#creates a vector with all numbers from 0 to 1 by intervals of 0.1  
  
lots\_of\_friends <- rep(friends, times = 5)   
#creates a vector repeating the first argument (the vector   
#'friends'), n times, where n is the second argument (100).

You can now pass the functions and operations described above to the whole sequence. The function is applied to each element of the original vector, and R will compile a vector with the sequence of results:

zero\_to\_ten^2

You can also obtain new vectors by manipulating vectors you already have, for instance, this syntax

zero\_to\_ten+decimals

returns a vector where the first element is the sum of element 1 of zero\_to\_ten and element 1 of decimals, the second element is the sum of element 2 of zero\_to\_ten and element 2 of decimals, and so on. (This will be very useful when we learn how to create new variables in a dataframe.)

Other functions return a single value computed from all the elements in the vector:

length(zero\_to\_ten) #returns the number of items in a vector  
sum(zero\_to\_ten) #returns the sum of elements of the vector  
mean(zero\_to\_ten) #returns the mean of elements of the vector  
median(zero\_to\_ten) #returns the median of elements of the vector  
max(zero\_to\_ten) #returns the maximum value in the vector  
min(zero\_to\_ten) #returns the minimum value in the vector  
sd(zero\_to\_ten) #returns the standard deviation of the vector  
var(zero\_to\_ten) #returns the variance of the vector

You can also ask R to evaluate each element of a vector according to a logical expression. This will return a vector of logical values TRUE/FALSE. *Stick a pin on this because it will be very useful when we index and subset dataframe variables later on.* For instance:

zero\_to\_ten > 6 #is the element larger than six?  
zero\_to\_ten >= 4 #is the element larger than or equal to four?  
zero\_to\_ten == 1 #is the element a one?   
zero\_to\_ten != 0 #is the element different from 0?

**Note the double equal in the penultimate line of code**. When we want to express equality as a logical or mathematical statement in R, we need to use ==. The single = is used within functions; otherwise, it works as an equivalent of the <- (assign) operator. So zero\_to\_ten = 1 would be creating an object called zero\_to\_ten with only one element, the number 1.

Just like vectors are collections of objects, we can create collections of vectors: *dataframes*. These are different from vectors in two key respects: (1) they have two dimensions, rows and columns, and (2) they can take values of different type (numbers, characters, logicals) in different columns. You can create your first dataframe so by passing your vectors in the data.frame() function. By default, your vectors will be treated as columns, and the vector names will become column names.

zero\_to\_ten  
squares <- zero\_to\_ten^2  
  
my\_dataframe <- data.frame(zero\_to\_ten, squares)  
  
my\_dataframe #print it

To visualise my\_dataframe, use the View() function:

View(my\_dataframe)

Note that if you were to type view with a lower-case V, you’d get an error: **R is case sensitive**, so whether it’s functions or objects, you want to pay attention to capitalisation.

You can pass as many arguments to data.frame() as you want, and give your dataframe columns name within the data.frame()function. Remember to separate the arguments of the function with commas:

new\_dataframe <- data.frame(zero\_to\_ten, squares, cubes = zero\_to\_ten^3)

Be careful with the length of your vectors when using the data.frame() function. If the vectors are *not* the same length of each other *but one is a multiple of the other*, the smaller vector will be replicated (or “looped”) when you bind them. If the two vector lengths are *not even multiples of each other*, you will not be able to bind them into a dataframe:

length(friends)  
#what's the length of my vector?  
  
data.frame(friends, binary\_variable = c(0,1))   
#the second vector is of length 2, which is a divisor of 6, so it gets replicated.  
  
data.frame(friends, binary\_variable = c(0,1,0,0,1))   
#the second vector is of length 5, which is \*not\* a divisor of 6, so you get an error.

Dataframes can take text, numerical and logical vectors. As long as they’re the same length, you’re good to go:

students <- c("Aditya", "Betty", "Charlie", "Diana")  
grade <- c(70, 55, 35, 81)  
pass <- grade >= 50  
  
data.frame(students, grade, pass)

#### 4. Setting up a Working Directory

Now let’s work with a real dataset.

Make sure you have downloaded the file quality\_of\_government.csv that comes with this course material. It might be useful to move it to a folder dedicated to the material for the Spring School: this will be your working directory, and it might be easier to have all your Spring School material in one place.

**A working directory is the default folder on your computer where R will look for files you want to load and where it will put any files you save.** The easiest way to set up a working directory is to select the ‘Session’ drop-down menu at the top of the RStudio interface, select Set Working Directory, then select Choose Directory, then search for the folder you’ve saved the file in, and click Open. Something like this should appear in your console:

setwd("~/Desktop/SpringSchool")

To check if this has worked, run

getwd()

Does the file path appear in the console? If so, you’ve set up your working directory and you can now skip to point 5. You will have to set up a working directory every time you close RStudio and open a new RStudio session (or when it crashes).

If it has not worked, you can manually set the working directory by copying the file path of the dataset you want to import and passing it through the setwd() function. For me, this is what the file path looks like:

"/Users/leonardocarella/Desktop/SpringSchool"

* For Mac users: you can get the file path by right-clicking on the file quality\_of\_government.csv in your documents, selecting Get Info and then copying what comes after Where: in the Info box.
* For Windows users: right-click on quality\_of\_government.csv in your documents, select Copy as path. **You may have to change the backward slashes to forward slashes in the R Script.**

Now pass your file path through the setwd() function, making sure to enclose it within quote marks:

setwd("/Users/leonardocarella/Desktop/SpringSchool")

#### 5. Loading and Exploring a Dataset

Once you have set a working directory, import the dataframe and store it as a dataframe called qog (quality of government) with the read.csv() function and the assign operator <-

qog <- read.csv("quality\_of\_government.csv")

Hopefully, a new object called qog will have appeared in your environment panel. **For some Mac users, this may not have worked because some of the newer Macs treat csv files as Numbers file**. In this case, you want to open quality\_of\_government in Numbers, and then export it as a .csv file named “quality\_of\_government.csv” (File > Export To... > CSV).

This is the dataset we are going to work with, it comes from the Quality of Government institute website (<https://www.gu.se/en/quality-government>), affiliated with the University of Gotenburg. It contains the following variables for 184 countries, collected in 2014:

|  |  |
| --- | --- |
| Variable | Description |
| iso3c | ISO 3 character country code |
| country | Country name |
| year | Year |
| region | Region |
| income | World Bank Income Group |
| electoral\_fraud | Serious electoral fraud in last election? (0,1) |
| proportional\_rep | Proportional representation? (0,1) |
| air\_quality | Air quality (higher is best) |
| fragile\_state\_index | Fragile State Index (higher scores more fragile) |
| educ\_years1524\_female | Years of education for females, aged 15-24 |
| educ\_years1524\_male | Years of education for males, aged 15-24 |
| govt\_revenue | Government revenue as % of GDP |
| polity | 21-point Polity score (-10 most autocratic; +10 most democratic) |
| freedomhouse\_status | Freedom House assessment of freedom |
| corruption\_perceptions\_index | Corruption Perceptions Index (lower is more corrupt) |
| human\_devt\_index | Human Development Index (higher is more developed) |
| access\_electricity | % of population with access to electricity |
| population | Population |
| gdp | GDP total, converted at purchasing power parity |

Some useful functions to inspect a new dataset:

View(qog) # shows the whole dataset in a new window  
dim(qog) # returns number of rows and columns in the dataset  
colnames(qog) # returns a vector of variable names  
summary(qog) # returns a summary of each variable

#### 6. Describing Variables and Handling Missing Data

**An essential operator when we work with dataframes is $ (extract)**. It allows us to select one variable in the dataframe to work with. So, for instance if the column we’re looking for is access\_electricty (i.e. the percentage of the population with access to electricity in a given country) in the dataframe qog, we call

qog$access\_electricity

and it will print out the variable. Note that this is a *vector*: exactly like our friend zero\_to\_ten, and we can treat it exactly in the same way. We can apply the functions we learnt before, and some new ones as well:

mean(qog$access\_electricity)  
median(qog$access\_electricity)  
sd(qog$access\_electricity)  
summary(qog$access\_electricity) #prints a summary of the variable distribution

Annoyingly, when a variable contains missing values (denoted by NA), we have to remind R to remove them from our calculations by setting na.rm = TRUE in our function, otherwise R will be unsure of how to compute the mean between numerical and logical values:

mean(qog$air\_quality) # produces NA (a missing value)  
mean(qog$air\_quality, na.rm = TRUE) # works fine! :)

To learn about categorical variables, we can use the unique() function to get a vector of the unique elements occurring in a vector, or the table() function to see how many times a value occurs.

unique(qog$region)  
table(qog$region)

The table() function can also be used to produce cross-tabulations between two variables:

table(qog$region, qog$income)

If you can’t remember what a variable is called, you can always run colnames(qog) to print out a vector of variable names. (When coding in R, copy-paste is your friend.)

#### 7. Indexing

R uses square brackets [ ] to select some values out of an object, such as vectors and dataframes, according to their position. For instance, if I wanted to select the third row and the second column of my dataset, I would call:

qog[3,2]

Remember: for rectangular data like dataframes, the first dimension is the *row* and the second dimension is the *column*.

You can also pass vectors instead of single values, for instance this prints rows from 1 to 10 of columns 2 and 4:

qog[1:10, c(2,4)]

If you want to select all the rows or all the columns, you can leave one of the dimensions empty, but you should retain the comma:

qog[40:50, ] #selects rows 40 through 50, all columns  
qog[, 2:6] #selects columns 2 through 6, all rows

You can also index a single column by using in combination the extract operator $ and indexing. In this case, you obviously only have to pass one dimension, because that’s the only dimension vectors have:

qog$income[1]  
qog$income[1:10]

A very important thing that indexing allows you to do is selecting values of a variable *conditional* on the values of another variable, for instance if I wanted to know what the value of ‘access\_electricity’ is for India, I would run…

qog$access\_electricity[qog$country == "India"] #Note the double equal!

What is going on ‘behind the scenes’ is that R is evaluating TRUE/FALSE for the logical statement qog$country == "India", and then it’s returning the value of access\_electricity **for the row in which the statement is TRUE**.

Some more examples of what we can do with indexing in combination with functions and operators:

qog$access\_electricity[qog$human\_devt\_index < 0.6] # index by a continuous variable  
mean(qog$access\_electricity[qog$human\_devt\_index < 0.6]) #combine with mean() function  
  
qog$access\_electricity[qog$income == "High income" | qog$income == "Upper middle income"]  
  
# use the OR operator '|' to condition on multiple criteria.  
  
qog$access\_electricity[qog$income %in% c("High income", "Upper middle income")]  
  
# you can do the same thing with the %in% operator, passing a vector of variables  
# for which the logical statement can be true.  
  
qog$access\_electricity[qog$income == "High income" &   
 qog$region == "Europe & Central Asia"]  
  
# use the AND operator '&' to condition on multiple criteria   
# (in this case, across more than one variable)

How would you compute the median value of air quality for high-income countries in the region “Europe & Central Asia”?

median(qog$air\_quality[qog$income == "High income" & qog$region == "Europe & Central Asia"])

Didn’t work. Why? *Missing values*. You need to specify na.rm = TRUE in the median() function.

median(qog$air\_quality[qog$income == "High income" & qog$region == "Europe & Central Asia"], na.rm = TRUE)

#### 8. Creating new variables

We can create new variables in an existing dataframe using a combination of the extract operator $ and the assign operator <-. When we create new variables, it is good practice to start always with an empty variable-vector, assigning NA (missing) to all entries. Then we can fill it up with mathematical operations or logical statements from the variables we already have.

The logic is exactly the same as we learnt when we summed our zero\_to\_ten vector with our decimals vector and got a new numerical vector as output, or when we got a logical TRUE/FALSE vector by asking R whether the elements of zero\_to\_ten were larger than six. Only, this time we are attaching the vector output to an existing dataframe as a new variable.

Let’s try for instance to create a new variable in our qog dataframe for GDP per capita: gdp\_pc, as the ratio between the existing gdp variable and the population variable:

qog$gdp\_pc <- NA  
qog$gdp\_pc <- qog$gdp/qog$population

What is happening behind the scene is that R is taking the first value of qog$gdp (the GDP of the country in row 1) and dividing it by the first value of qog$population (the population of the country in row 1), repeats this for all 184 rows, and then spits out a vector of length 184 which is attached to our dataframe. We can go even further, and create a log\_gdp\_pc variable, which is the logaritmic transformation of gdp\_pc (this may be useful because, say, GDP per capita is a highly skewed variable):

qog$log\_gdp\_pc <- NA  
qog$log\_gdp\_pc <- log(qog$gdp\_pc)

We can combine this procedure with indexing. For instance, let’s say we want a binary variable named free that takes the value of 1 when a country is “Free” according to its freedomhouse\_status, and takes the value of 0 if it is classed as “Not Free” or “Partly Free”. We would run:

qog$free <- NA  
qog$free[qog$freedomhouse\_status == "Free"] <- 1  
qog$free[qog$freedomhouse\_status == "Partly Free"] <- 0  
qog$free[qog$freedomhouse\_status == "Not Free"] <- 0

Or, more succinctly:

qog$free <- NA  
qog$free[qog$freedomhouse\_status == "Free"] <- 1  
qog$free[qog$freedomhouse\_status %in% c("Not Free", "Partly Free") ] <- 0

#### 9. Data Visualisation in Base R

Now that we’ve learnt how to describe, manipulate and create variables, let’s try our hand at visualisation. In this part of the class, we’ll learnt some of the most common visualisation functions that come with ‘base’ R. These functions are pretty simple to use for basic plots, but they can get a bit fiddly when it comes to adding many graphical parameters: we’ll learn an alternative approach to make prettier plots with a package called ggplot at the end of the class.

But for now let’s start with the basics: histograms.

hist(qog$polity)

Within the hist() function, we can pass some more arguments to make our plot more understandable: add labels with xlab and ylab, add a title with main.

hist(qog$polity, xlab = "Polity Scores",   
 ylab = "Count",   
 main = "Distribution of Polity Scores")

If we want smaller bins, we have to pass a vector of axis breaks to the argument breaks. We can also change the y axis from frequency to density by setting the argument freq to FALSE.

hist(qog$polity, xlab = "Polity Scores",   
 ylab = "Density",   
 main = "Distribution of Polity Scores",  
 breaks = seq(-10,10, by = 1), freq = FALSE)

After we’ve created a plot, we can *add* graphical elements For instance to add a red vertical line that intercepts the y axis at the mean value of polity, we call:

abline(v = mean(qog$polity, na.rm = T), col = "red")

The function to create a boxplot is equally intuitive:

boxplot(qog$polity)

We can plot more than one boxplot side-by-side, by passing more than one vector to the boxplot() function. For instance, if I want two separate boxplots for “Free” and “Partly/Not Free” countries, I can use the variable I created before and a bit of indexing magic:

boxplot(qog$polity[qog$free == 1],   
 qog$polity[qog$free == 0])

Pass a character vector to the names argument to name your boxplots:

boxplot(qog$polity[qog$free == 1],   
 qog$polity[qog$free == 0],  
 names = c("Free", "Not/Partly Free"),  
 main = "Distribution of Polity Scores")

An alternative way of plotting the boxplot of a continuous variable conditional on the value of a categorical variable is using the ~ symbol (it’s called a tilde and we read it as ‘by’: make a note of where it is on your keyboard, because we’re going to need it).

boxplot(qog$polity ~ qog$income,  
 xlab = "Income Categories",   
 ylab = "Polity Scores")

To export a plot, click on “Export” in the plots window, then select “Save as Image…”

Finally, the plot() function calls a scatterplot by default, where the first argument is the x-axis variable and the second argument is the y-axis. This is, for instance, how to create a scatterplot showing the relationship between the two continuous variables, log\_gdp\_pc and corruption\_perceptions\_index (counterintuitively, higher values mean *less* perceived corruption, so we may want to give the axis a label that reflects that):

plot(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 xlab = "Log of GDP per capita",   
 ylab = "Perceived integrity",  
 main = "GDP and Corruption")

##### 9.1 Scatter Plots in Base R: Some Extra Material

After you’ve called a scatter plot, you can use the ‘points’ function to plot a subset of the dataset (say, East Asian or Pacific countries) in a different colour. For instance,

plot(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 xlab = "Log of GDP per capita",   
 ylab = "Perceived integrity",  
 main = "GDP and Corruption")  
  
points(x = qog$log\_gdp\_pc[qog$region == "East Asia & Pacific"],   
 y = qog$corruption\_perceptions\_index[qog$region == "East Asia & Pacific"],   
 col = "red")

Use the text() function to plot the label (variable name iso3c) of some countries you may be interested in, specifying the text label with the labels argument:

text(x = qog$log\_gdp\_pc[qog$country %in% c("China", "Japan", "India")],   
 y = qog$corruption\_perceptions\_index[qog$country %in% c("China", "Japan", "India")],   
 labels = qog$iso3c[qog$country %in% c("China", "Japan", "India")])

If you only want to plot country codes instead of points, call an empty plot, by specifying type = "n", and then add the text labels with the text() function:

plot(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 xlab = "Log of GDP per capita",   
 ylab = "Perceived integrity",  
 main = "GDP and Corruption",  
 type = "n")  
  
text(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 labels = qog$iso3c, cex = 0.5)

To build more complex plots, it’s often useful to proceed step by step, starting from the blank canvas of an empty plot. For instance, here’s how to proceed to build a colour-coded plot, with separate regression lines for the subsets of ‘Free’ and ‘Not/Partly Free’ countries. In my opinion, this way of proceeding is **really fiddly** – that’s why many R users these days prefer the ggplot approach which we’ll see at the end of the class. But part of the beauty of R is that there are often many ways of doing one thing, and with experience you can find out what works best for your workflow, your coding skills, and your research needs.

plot(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 xlab = "Log of GDP per capita",   
 ylab = "Perceived integrity",  
 main = "GDP and Corruption",  
 type = "n")  
  
points(x = qog$log\_gdp\_pc[qog$free == 0],   
 y = qog$corruption\_perceptions\_index[qog$free == 0],   
 col = "red")  
   
points(x = qog$log\_gdp\_pc[qog$free == 1],   
 y = qog$corruption\_perceptions\_index[qog$free == 1],   
 col = "blue")  
   
legend("topleft", legend=c("Free", "Not/Partly Free"),  
 fill=c("blue", "red"))  
   
abline(lm(qog$corruption\_perceptions\_index[qog$free == 0] ~   
qog$log\_gdp\_pc[qog$free == 0]), col = "red", lty = "dashed")  
  
#adds a regression line (lm is the function for linear model, see below)  
  
abline(lm(qog$corruption\_perceptions\_index[qog$free == 1] ~   
qog$log\_gdp\_pc[qog$free == 1]), col = "blue", lty = "dashed")

#### 10. Regressions in R

The function for a linear regression (OLS) in R is lm() (linear model), and it takes the syntax lm(Y\_variable ~ X\_variable1 + X\_variable2...). Let’s now regress corruption\_perceptions\_index on log\_gdp\_pc, and store the output in an object named model1 (or whatever you prefer).

model1 <- lm(qog$corruption\_perceptions\_index ~ qog$log\_gdp\_pc)

If you call model1 it will print out the intercept and slope; if you call summary(model1), it will return a lot more information (this is normally what you want to see):

model1  
summary(model1)

An alternative, equivalent syntax specifies the dataframe under the argument data, so you can pass the variable names without having to specify the dataframe for each of them:

model1 <- lm(data = qog, corruption\_perceptions\_index ~ log\_gdp\_pc)  
summary(model1)

Taking a step back to visualisation, you can plot regression lines by passing as argument of abline() a bivariate model that has as dependent variable the variable on the Y axis and as indepedent variable the variable on the X axis.

plot(x = qog$log\_gdp\_pc,   
 y = qog$corruption\_perceptions\_index,   
 xlab = "Log of GDP per capita",   
 ylab = "Perceived integrity",  
 main = "GDP and Corruption")  
  
abline(model1)

To run a multivariate OLS model, simply add independent variables to the right of the tilde with a plus sign:

model2 <- lm(qog$corruption\_perceptions\_index ~ qog$log\_gdp\_pc +   
 qog$freedomhouse\_status + qog$human\_devt\_index)  
summary(model2)

Note that R is dropping one of the categories for the freedomhouse\_status categorical variable: this is the reference category that you want to compare your slope coefficients to. To change reference category, use the relevel function:

qog$freedomhouse\_status <- relevel(qog$freedomhouse\_status, ref = "Not Free")  
  
model2 <- lm(qog$corruption\_perceptions\_index ~ qog$log\_gdp\_pc +   
 qog$freedomhouse\_status + qog$human\_devt\_index)  
summary(model2)

You can add interaction factors with an asterisk. When you interact two variables in this way, R will automatically include both variables *plus* the interaction. Here for instance, we’re interacting GDP per capita with freedomhouse\_status. Can you interpret substantively the results?

model3 <- lm(data = qog, corruption\_perceptions\_index ~   
 log\_gdp\_pc\*freedomhouse\_status +   
 human\_devt\_index)  
summary(model3)

We won’t go much beyond OLS in this session, but one thing that might be useful is the glm function, which stands for generalised linear models, such as logistic regressions. This allows you to specify the details of the model assumptions under the family argument:

model4 <- glm(data = qog, free ~ log\_gdp\_pc + region, family = binomial)  
summary(model4)

The default link for family = binomial is logit. But you can specify, for instance, family = binomial(link = "probit") for a probit link model, family = poisson for a Poisson (count) model, etc.

#### 11. Installing and Loading Packages: Example with Stargazer

Packages are bundles of code and data that the R community has produced, and that you can use in your own code to expand the functionalities of R. They are currently over 19,000 of them available on the CRAN repository (<https://cran.r-project.org/web/packages/available_packages_by_name.html>), ranging from packages for the natural language processing of Korean text, to packages for advanced Bayesian modelling, to packages like fortunes, whose only function is to return random fortune cookie-style quotes. In most cases, the point of installing packages is using additional functions that base R does not have.

The first example we are going to look at today is the package stargazer, which is used to format and export regression outputs in html, LaTeX and ASCII text.

To install packages from the CRAN repository, which is where the things you need are going to be in 99% of the cases, you simply have to pass the name of the package as a character in the install.packages() function:

install.packages("stargazer")

At this point, you may get a message in the console asking you if you want to install dependencies (other packages that are needed to make your package work). Just type ‘yes’ in the console, and leave it to compile until it says DONE (stargazer). The package is now installed on your device’s library. You won’t have to run this command again in the future for stargazer. An alternative way to install packages is to click on the ‘Packages’ window on the bottom-right panel of your scree, then selecting ‘Install’, and searching manually for the package you need. Make sure you have ‘install dependencies’ ticked.

To use stargazer in this R session, we need to load it form our library, with the function:

library(stargazer)

The function from this package that we are going to use is also called stargazer(). Make sure you have stored our regressions from the last section and let’s try and format the output of model1:

stargazer(model1, type = "text")

If you use LaTeX, the following code will return a regression table that you can directly copy-paste there:

stargazer(model1, type = "latex")   
  
# this is equivalent to stargazer(model1), as if you don't   
# specify type the default output is LaTeX.

You can use stargazer to format plots side-by-side, and change other graphical parameters (run ?stargazer for more info) by specifying other arguments. For instance, here I am putting standard errors on the same row as the estimates, and rounding to the second decimal place:

stargazer(model1, model2, type = "text", single.row = TRUE, digits = 2)

Finally, you can use the out argument to save your regression output as an html file. This will appear in your working directory (from there, you may for instance copy-paste it into a word document):

stargazer(model1, model2, model3, out = "my\_model.html")

#### 12. Tidyverse I: data wrangling with dplyr

One of the most widely used packages in R is tidyverse. Technically, it is a *collection* of packages for many different purposes, which work well in combination and share a common philosophy of high-level design and ease of use. In my view, the R community is moving towards doing more and more things with tidyverse, and many of the best courses start straight away with teaching this ‘style’ of R coding over base R, which is what we’ve seen so far.

Today, we’re going to look at two of the tidyverse packages: dplyr, a grammar for data manipulation, and ggplot2, a grammar for data visualisation. You can do virtually all of the things we will do in tidyverse with base R functions or with other packages, but the tidyverse approach has some key advantages, which have made it very popular over the years. Some of the courses in your Spring School courses will include some tidyverse functions, so it’s worth having a rough idea of how they work.

First, install and load tidyverse (this is likely to require compilation of dependencies, so be ready to type out ‘Yes’ in the console if prompted):

install.packages("tidyverse")  
library(tidyverse)

One basic dplyr function is filter(). It subsets a dataframe conditional on a statement. For instance, if I wanted a dataframe with only the observations in qog that are in the South Asian region, I can run:

filter(qog, region == "South Asia")

So far, so good. A key characteristic of the tidyverse approach is the **pipe operator** %>% (read as ‘and then’). This allows you to build up your code by sequencing functions after another. For instance, if I wanted to tell R ‘take my dataframe qog, and then filter by region’, I may type:

qog %>% filter(region == "South Asia")

We can store the output as a new dataframe by using the assign and the pipe operator in combination:

south\_asia <- qog %>%  
 filter(region == "South Asia")

Another useful function in dplyr is select(). This allows to select columns. So for instance, if we only wanted to see how South Asian countries fare in terms of years of education of 15-24 males and 15-24 females, we can use the piping approach to say ‘take my dataframe qog, and then filter by region, and then select the columns country, educ\_years1524\_male and educ\_years1524\_female; store the output as an object named south\_asia:

south\_asia <- qog %>%  
 filter(region == "South Asia") %>%  
 select(country, educ\_years1524\_male, educ\_years1524\_female)

Now let’s say we want to look at gender gaps in education through a simple variable that records the difference in average years of education between men and women, instead of looking at the two variables side-by-side. We could create a new variable in the way we’ve learnt in section 8, or we may use the mutate() function. mutate() takes the syntax mutate(new\_variable = some parameters)

qog %>%  
 filter(region == "South Asia") %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 select(country, education\_gap)

That "Pakistan (1971-)" text string sticks out like a sore thumb. How would we change it to simply "Pakistan"? Use the mutate() function in combination with the recode() function. recode() takes the syntax recode(variable, new\_value = old\_value).

qog %>%  
 filter(region == "South Asia") %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 select(country, education\_gap)%>%  
 mutate(country = recode(country, "Pakistan (1971-)" = "Pakistan"))

Now let’s add the arrange() variable to sort our observation by the value of education\_gap, and store the output as a new dataframe, south\_asia, which will override the one we created before:

south\_asia <- qog %>%  
 filter(region == "South Asia") %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 select(country, education\_gap) %>%  
 mutate(country = recode(country, "Pakistan (1971-)" = "Pakistan")) %>%  
 arrange(by = education\_gap)

Now let’s try a different trick. Imagine we want to see how the education gender gap varies across regions rather than across countries within a region. We can get the mean value of the gap, by asking R to group\_by() observations according to the variable region, and then summarise the information in education\_gap as a mean. (A good thing about dplyr is that it supports both the American spelling, e.g. summarize() and the British spelling summarize().) Remember to pass our friend na.rm = TRUE to the mean() function.

qog %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 group\_by(region) %>%  
 summarise(mean\_education\_gap = mean(education\_gap, na.rm = TRUE))

Remember to pass our friend na.rm = TRUE to the mean() function.

We can create more than one new variable within the summarise() argument. For instance, let’s repeat the same operate, but this time at the same time we want to create a count variable, recording the number of countries in each region. We can do that by nesting the n() (count) function in mutate():

qog %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 group\_by(region) %>%  
 summarise(no\_countries = n(),  
 mean\_education\_gap = mean(education\_gap, na.rm = TRUE)) %>%  
 arrange(mean\_education\_gap)

We can go all the way, and create variables for the mean, median, maximum and minimum value as well, and save the output as a new dataframe education\_gaps:

education\_gaps <- qog %>%  
 mutate(education\_gap = educ\_years1524\_male - educ\_years1524\_female) %>%  
 group\_by(region) %>%  
 summarise(no\_countries = n(),  
 mean\_education\_gap = mean(education\_gap, na.rm = TRUE),  
 median\_education\_gap = median(education\_gap, na.rm = TRUE),  
 max\_education\_gap = max(education\_gap, na.rm = TRUE),  
 min\_education\_gap = min(education\_gap, na.rm = TRUE)) %>%  
 arrange(mean\_education\_gap)

Note that you can also group by more than one variable. For instance, if we wanted to know the mean human development index for all the combinations of Freedom House Status and Income categories, we could call:

qog %>%  
 group\_by(freedomhouse\_status, income) %>%  
 summarise(mean\_hdi = mean(human\_devt\_index, na.rm = TRUE))

#### 13. Tidyverse II: data visualisation with ggplot2

In my opinion the strongest suit of the tidyverse approach is the data visualisation package ggplot2. It allows you to produce much neater plots with fewer lines of code than base R. The core function you will use to plot any graph is ggplot(). Within that function you pass the data argument (your dataset) and aesthetic parameters within aes(): the axes, the colour-coding, etc. Then you add to that call other functions, separated by a plus sign +. These will specify, for instance, the type of graph you want to plot, your labels etc.

Try for instance to plot a scatter plot by appending the geom\_point() to ggplot()…

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index)) + geom\_point()

Now let’s say we want to colour-code these points by freedomhouse\_status. We can do so by adding to aes() a colour (or color, if you prefer) argument:

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,  
 colour = freedomhouse\_status)) + geom\_point()

Add axes labels with xlab() and ylab() and a main title with ggtitle. Remember to use plus signs (these cannot be at the beginning of a new line):

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,  
 colour = freedomhouse\_status)) +   
 geom\_point() +   
 xlab("Log of GDP per capita") +   
 ylab("Corruption (high = less corrupt)") +   
 ggtitle("GDP and corruption")

Now let’s say we want to change the colours (the red and green in the default palette may not be colourblind-friendly, for instance). You can do that with scale\_colour\_manual() (or color, or simply col):

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,  
 colour = freedomhouse\_status)) +   
 geom\_point() +   
 xlab("Log of GDP per capita") +   
 ylab("Corruption (high = less corrupt)") +   
 ggtitle("GDP and corruption") +   
 scale\_color\_manual(values = c("violet", "orange", "darkgreen"), name = "Freedom")

Colour is not the only graphical parameters we can use to visualise information. Try for instance to make the size of the points conditional on a country’s population by adding a size argument to our aes().

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,  
 colour = freedomhouse\_status,  
 size = population)) +   
 geom\_point() +   
 xlab("Log of GDP per capita") +   
 ylab("Corruption (high = less corrupt)") +   
 ggtitle("GDP and corruption") +   
 scale\_color\_manual(values = c("violet", "orange", "darkgreen"),   
 name = "Freedom") +   
 theme\_minimal()

A few more touches. Let’s make a sensible size scale with scale\_size and remove the ugly grey background (a pet hate of mine) with theme\_minimal():

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,  
 colour = freedomhouse\_status,  
 size = population)) +   
 geom\_point() +   
 xlab("Log of GDP per capita") +   
 ylab("Corruption (high = less corrupt)") +   
 ggtitle("GDP and corruption") +   
 scale\_color\_manual(values = c("violet", "orange", "darkgreen"),   
 name = "Freedom") +   
 scale\_size(range = c(1,6), breaks = c(10^7, 10^8, 10^9),  
 labels = c("10", "100", "1000"),   
 name = "Population (millions)") +   
 theme\_minimal()

Much better. Now we can save the plot (in our working directory) with ggsave()

ggsave("my\_first\_ggplot.png")

Let’s go back to a very basic plot and add a regression line with stat\_smooth(method = lm):

ggplot(data = qog,mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index)) +   
 geom\_point() + theme\_minimal() + stat\_smooth(method = lm)

How could we show the difference in the effect of GDP on corruption across different types of regime? One option is to specify a group parameter, so that stat\_smooth will compute the linear model for the three different regime categories:

ggplot(data = qog,mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,   
 group = freedomhouse\_status)) +   
 geom\_point() + theme\_minimal() + stat\_smooth(method = lm, mapping = aes(   
 col = freedomhouse\_status))

How to distinguish between the three regime types? We can specify a colour and a line type parameter with aes() within stat\_smooth()

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index,   
 group = freedomhouse\_status)) +   
 geom\_point() + theme\_minimal() + stat\_smooth(method = lm, mapping = aes(   
 colour = freedomhouse\_status,   
 linetype = freedomhouse\_status), se = FALSE)

Note that when you pass an aes() in the ggplot() function, it will be applied to all functions appended: both the points and the lines will be colour-coded. When you pass it within only one function, as we did with stat\_smooth() it will only apply to that aesthetic.

Finally, we could visualise the same information by creating side-by-side plots of the three regime types instead of colour-coding. You can do this easily in ggplot() by using facet\_wrap(~) (read ‘facet wrap by’)

ggplot(data = qog, mapping = aes(x = log\_gdp\_pc,   
 y = corruption\_perceptions\_index)) +   
 geom\_point() + stat\_smooth(method = lm) +   
 facet\_wrap(~freedomhouse\_status) + xlab("Log of GDP per capita") +   
 ylab("Corruption (high = less corrupt)") +   
 ggtitle("GDP, corruption and regime type") + theme\_minimal()

#### 14. Learn More: Useful Resources

A great deal of learning in R is self-learning and proceeds roughly as follows: (1) googling what you need to do – “how to centre a plot title in ggplot?”, “how to weight a regression model in R?”, “how do I export a table from R to LaTeX?” – and (2) trouble-shooting from there. There is a massive community of R users on Stack Overflow (<https://stackoverflow.com/>) that over the years have helped out newcomers to the language overcome their coding issues. It’s likely that your question will already have been answered (if not, you can post it there!).

No course will cover all your needs, but here are some resources for learning and practising more in R:

* **swirl**: a package dedicated to learning R. To start an interactive course in swirl, which goes from the basics to modelling and exploratory data analysis, you just need to install, load and launch swirl:

install.packages("swirl") # installs the package   
library(swirl) # loads it  
swirl() # launches it

After you run swirl(), the programme will prompt you to select the courses you want to learn about in the console, and guide you through the topic step by step. The enthusiasm with which it greets successful code is a bit patronising, but apart from that it’s a very useful resource. I’d recommend going through the courses “1: R Programming: The basics of programming in R” and “2: Regression Models: The basics of regression modeling in R”.

* Chris Hanretty’s **ConveRt** course. Designed for STATA and SPSS users who want to learn R, it’s a brilliant resource that **covers the content of this introduction more in depth and introduces a few more topics as well** (including more advanced regression modelling). It makes extensive use of the tidyverse approach from the get go. You should be able to go through it in 3-4 days. It’s available at <http://chrishanretty.co.uk/conveRt/#1>.
* **R for Data Science** by Hadley Wickham and Garrett Grolemund: it’s an online textbook freely available at <https://r4ds.had.co.nz/>. It espouses in full the tidyverse approach, and **takes you a lot further than this short course** in terms of coding. If you want to get seriously good at R, this is where you want to start.
* **Quantitative Politics in R** by Erik Gahner Larsen and Zoltán Fazekas: another excellent free online textbook, aimed specifically at political scientists. Like Wickham and Grolemund’s book, the material is tidyverse-based, but overall it’s a much **more synthetic** course. You can go through it in a week or so. Available at <http://www.qpolr.com/index.html>.
* **YaRrr! The Pirate’s Guide to R** by Nathaniel Phillips. An online textbook available at <https://bookdown.org/ndphillips/YaRrr/>. Unlike the last three resources, it **sticks largely to base R** and has extensive exercise sections to “test your R might”. An accessible and entertaining introduction to R.