Lesson

Rohit Rawat

January 24, 2016

Outline

We have covered basic R usage:

- Reading data files
- Creating and manipulating variables
- Data types
- Calling built-in functions

Now we will cover:

- Writing R scripts
- Writing functions
  - Scope issues in functions, Missing arguments
- Calling R scripts from the command line

Writing R scripts

You have a basic idea of what scripts are from the shell lesson.

Just like we learned to use shell commands like `grep`, `sort`, and `wc`, and then later learned how to write shell scripts. A script helps us deal with a bigger or more complicated task. It will consist of a logical sequence of steps needed to solve a problem. This could include

- reading in data
- creating variables to store data
- performing data analysis
- creating more variables to store intermediary and final results
- outputting the results as numbers or plots

Challenge 2.1 Goal : to identify the data manipulation actions needed to perform a task

Look at this data:

country	year	pop	continent	lifeExp	gdpPercap
Afghanistan	1952	8425333	Asia	28.801	779.4453
Afghanistan	1957	9240934	Asia	30.332	820.8530
Afghanistan	2007	31889923	Asia	43.828	974.5803
Albania	1952	1282697	Europe	55.230	1601.0561
Albania	1967	1984060	Europe	66.220	2760.1969
Albania	1972	2263554	Europe	67.690	3313.4222
Albania	1977	2509048	Europe	68.930	3533.0039
Argentina	1952	17876956	Americas	62.485	5911.3151
Argentina	1957	19610538	Americas	64.399	6856.8562

All of this data is loaded into a `data.frame` called `gapminder`. Your boss wants you to compute the average life expectancy over all the years of data available for *Albania*. **Think of what other variables you will need to create to store intermediary results and the final result. Also think about the type each variable will have to be (data.frame, vector, scalar, string?)**. There are many correct answers to this problem. (no code is needed).

Solution You will want to maybe create a subset `data.frame` with only the rows for Algeria. You would then want to cut out life expectancy column into a `vector`. You would then compute the mean and store it in a `scalar`.

First script in RStudio

Instead of writing individual statements on the command line, we are going to write our code directly into a script. Scripts are meant to be run in one fell swoop once they are complete. But for new programmers, we will be executing every statement as soon as we type it to look out for errors. In RStudio, we can run the statement under the cursor by pressing the `Ctrl-Enter` key combination.

In the default layout, RStudio has a text editor in the upper left corner.

We start by reading the data. We know the csv file is in the data folder. Recall relative paths from shell. The relative path would be `data/gapminderData.csv`.

Windows Users: What happened to using \s in paths like `data\gapminderData.csv`? RStudio prefers Unix style `/` even on Windows. Use `/`. When pasting paths that already have `\`, you must either replace them with a `/` or `\\`. A single `\` does not work.

```r
# This script computes the average GDP for Albania using the gapminder data

# location of the data
fileName <- 'data/gapminderData.csv'
```

We now use the `read.csv` function to read in the data file

```r
# read the data file
gapminder <- read.csv(fileName)
```

Note: Does the . mean anything? Unlike other languages, in R, it may or or may not mean anything. In this case, the `read.` prefix is used to group some of the file I/O related functions together.

Let's take a look at the data.

```r
View(gapminder)
```

`gapminder$country` yeilds the country column from the gapminder data.frame. The statement `gapminder$country == 'Albania'` produces a logical vector which is TRUE when the country name matches 'Albania'

On the console, show:

```r
gapminder$country == 'Albania'
```

```r
# select the rows where the country is Albania and store it in albaniaData
albaniaData <- gapminder[gapminder$country == 'Albania',]

# select the column containing the GDP per capita from the Albania data
albaniaGdp <- albaniaData$gdpPercap

# compute the average GDP value
albaniaAverageGdp <- mean(albaniaGdp)
```

Note: These statements can be written in a more compact way if desired.

```r
albaniaAverageGdp <- mean(gapminder[gapminder$country == 'Albania', 'gdpPercap'])
```

```r
# print a message with the result of our computation
print(paste('The average GDP of Albania is', albaniaAverageGdp))
```

```
## [1] "The average GDP of Albania is 3255.36663266667"
```

Instead of typing them one by one, we wrote all our commands into a single script.

The advantages are:

- we have a well commented record of our code
- we can change a value in the beginning of the script and run it again to get the new results (like change 'Albania' to something else, or use a different `fileName`)

Advantages of using the RStudio editor

- Syntax highlighting
- Automatic code indentation and bracket pairing
- Variable name completion with [tab] (like bash)
- Easier debugging

Our completed script looks like this:

myScript.R

```r
# This script computes the average GDP per capita for Albania using the gapminder data

# location of the data
fileName <- 'data/gapminderData.csv'

# read the data file
gapminder <- read.csv(fileName)

# select the rows where the country is Albania and store it in albaniaData
albaniaData <- gapminder[gapminder$country == 'Albania',]

# select the column containing the GDP per capita from the Albania data
albaniaGdp <- albaniaData$gdpPercap
```

```r
# compute the average GDP per capita value
albaniaAverageGdp <- mean(albaniaGdp)

# print a message with the result of our computation
print(paste('The average GDP per capita of Albania is', albaniaAverageGdp))
```

```
## [1] "The average GDP per capita of Albania is 3255.36663266667"
```

Now that the script is complete, we can run it as a complete unit. Scripts are executed using the `source()` function

`source(filename)`

You would normally use the `setwd()` command to change the working directory to where the script is stored and then call `source()`.

```r
setwd('~/Dropbox/Carpentry/R_lesson/gapminder')
source('script.R')
```

```
## [1] "This is script.R"
```

You can also press the `Source` button, which launches your script irrespective of the working directory. Although this may lead to problems if you are sourcing other scripts or loading files using a relative path. You can use the menu option Session -> Set Working Directory -> To source file location to set the current working directory to where your R code is.

Question: Why do we need print() around paste() when sourcing, but not when we ran paste() on the console?

Challenge 2.2 Goal : write a basic script

Write and run your own script that will

- Set `x` to 25
- Set `y` to 15
- Calculate the sum and store it in a variable `z`
- print `z`

To start a new script, open the File menu -> New File -> R script. Save it as **`myAdder.R`** and run it.

Solution

```r
# a simple script to add two numbers

# create two variables x and y
x <- 25
y <- 15

# calculate the sum z
z <- x + y

# print the result
print(paste('The sum of 25 and 15 is', z))
```

```
## [1] "The sum of 25 and 15 is 40"
```

Creating Functions

Functions are "canned scripts" that automate something complicated or convenient or both. Many functions are predefined, or become available when using the function library() (more on that later). A function usually gets one or more inputs called arguments. Functions often (but not always) return a value. A typical example would be the function sqrt(). The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called calling the function. An example of a function call is:

b <- sqrt(a)

Functions gather a sequence of operations into a whole, preserving it for ongoing use. Functions provide:

```
- a name we can remember and invoke it by
- relief from the need to remember the individual operations
- a defined set of inputs and expected outputs
- rich connections to the larger programming environment
```

Let's crate a new script, call it functions-lesson.R, and write some examples of our own. Let's define a function `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5/9)) + 273.15
    kelvin
}
```

The definition opens with the name of your new function, which is followed by the call to make it a `function` and a parenthesized list of parameter names. You can have as many input parameters as you would like (but too many might be bad style). The body, or implementation, is surrounded by curly braces `{ }`. In many languages, the body of the function - the statements that are executed when it runs - must be indented, typically using 4 spaces. While this is not a mandatory requirement in R coding, we strongly recommend you to adopt this as good practice.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. The last line within the function is what R will evaluate as a returning value. For example, let's try running our function. Calling our own function is no different from calling any other function:

```
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

```
print(paste('boiling point of water:', fahr_to_kelvin(212)))
```

```
## [1] "boiling point of water: 373.15"
```

We've successfully called the function that we defined, and we have access to the value that we returned.

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5/9)) + 273.15
    kelvin
}
```

Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

```r
kelvin_to_celsius <- function(temp) {
    Celsius <- temp - 273.15
    Celsius
}

print(paste('absolute zero in Celsius:', kelvin_to_celsius(0)))
```

```
## [1] "absolute zero in Celsius: -273.15"
```

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

```r
fahr_to_celsius <- function(temp) {
    temp_k <- fahr_to_kelvin(temp)
    result <- kelvin_to_celsius(temp_k)
    result
}

print(paste('freezing point of water in Celsius:', fahr_to_celsius(32.0)))
```

```
## [1] "freezing point of water in Celsius: 0"
```

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-large chunks to get the effect we want. Real-life functions will usually be larger than the ones shown here—typically half a dozen to a few dozen lines—but they shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on. **Modular programming**

Challenge 2.3 Goal : Wrapped function calls.

As we've seen in our print statements, we can use `paste` or `paste0` to concatenate strings.

1. Write a function called `fence` that takes two parameters called `original` and `wrapper` and returns a new string that has the `wrapper` character at the beginning and end of the `original`:

Example function call and output:

```
fence('name', '---')
---name---
```

Solution

```r
# function to surround a piece of text with other text
fence <- function(original, wrapper) {
  result <- paste0(wrapper, original, wrapper)

  return(result)
}

# test function call
print(fence('Rohit', '***'))
```

```
## [1] "***Rohit***"
```

Explaining the R Environments

Lets go back to the script we were writing before:

```r
# This script computes the average GDP for Albania using the gapminder data

# location of the data
fileName <- 'data/gapminderData.csv'

# read the data file
gapminder <- read.csv(fileName)

# select the rows where the country is Albania and store it in albaniaData
albaniaData <- gapminder[gapminder$country == 'Albania',]

# select the column containing the GDP per capita from the Albania data
albaniaGdp <- albaniaData$gdpPercap

# compute the average GDP value
albaniaAverageGdp <- mean(albaniaGdp)

# print a message with the result of our computation
print(paste('The average GDP of Albania is', albaniaAverageGdp))
```

```
## [1] "The average GDP of Albania is 3255.36663266667"
```

And make a small change:

```r
# This script computes the average GDP for Albania using the gapminder data

# location of the data
fileName <- 'data/gapminderData.csv'

# read the data file
gapminder <- read.csv(fileName)

# create a variable to store a country name
countryName <- 'Algeria'

# select the rows where the country is Albania and store it in albaniaData
albaniaData <- gapminder[gapminder$country == countryName,]     # note the change here

# select the column containing the GDP per capita from the Albania data
albaniaGdp <- albaniaData$gdpPercap

# compute the average GDP value
albaniaAverageGdp <- mean(albaniaGdp)

# print a message with the result of our computation
print(paste('The average GDP of', countryName, 'is', albaniaAverageGdp)) # and the change here
```

```
## [1] "The average GDP of Algeria is 4426.02597316667"
```

Would using country instead of countryName cause any conflict with gapminder$country? No.

In a new script file, write a function `getAverageGdpPerCapita` that takes the name of a country and the gapminder dataset as its two inputs, then computes the averaged GDP per capita and returns it.

```r
# This script computes the average GDP for a country using the gapminder data

# clear old variables
rm(list=ls())

# location of the data
fileName <- 'data/gapminderData.csv'

# read the data file
gapminder <- read.csv(fileName)
```

```r
getAverageGdpPerCapita <- function(country, gapminder) {
  # extract gdpPercap from the gapminder data for the specified country.

  selectedCountryData <- gapminder[gapminder$country == country, 'gdpPercap']

  mean(selectedCountryData)
}
```

Try out our new function

```r
gdpUSA <- getAverageGdpPerCapita('United States', gapminder)
gdpCanada <- getAverageGdpPerCapita('Canada', gapminder)
gdpMexico <- getAverageGdpPerCapita('Mexico', gapminder)
```

What will happen if we remove the second argument to the function? Will it throw an error?

```r
getAverageGdpPerCapita <- function(country) {
  # extract gdpPercap from the gapminder data for the specified country.

  selectedCountryData <- gapminder[gapminder$country == country, 'gdpPercap']

  mean(selectedCountryData)
}

gdpUSA <- getAverageGdpPerCapita('United States')
gdpCanada <- getAverageGdpPerCapita('Canada')
gdpMexico <- getAverageGdpPerCapita('Mexico')

print(paste('GDP of USA is', gdpUSA))
```

```
## [1] "GDP of USA is 26261.1513466667"
```

```r
print(paste('GDP of Canada is', gdpCanada))
```

```
## [1] "GDP of Canada is 22410.74634"
```

```r
print(paste('GDP of Mexico is', gdpMexico))
```

```
## [1] "GDP of Mexico is 7724.11267458333"
```

When we create new variables, they are created in the current frame, which is initially the global frame:

```r
search()
```

```
##  [1] ".GlobalEnv"        "package:knitr"     "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

When we access a variable, those variables are also searched for in the current frame.

When a function is called, R searches for variables that are arguments to the function and any new variables that are created in the function. If a variable cannot be found, it searched for in its parent frame. The parent frame is the frame where it is defined, and not where it is called.

Default values

Challenge

Modify the function getAverageGdpPerCapita that accepts 3 arguments, a country name, a start year, and an end year. The function should compute the average gdp of that country between the range of years supplied. Use the logical & operator to combine the multiple conditions required. Or you can apply the three conditions one afer the other.

```
getAverageGdpPerCapita <- function(country, startYear, endYear) {
  # extract gdpPercap from the gapminder data for the specified country.

  selectedCountryData <- gapminder[gapminder$country == country &
                                   gapminder$year >= startYear &
                                   gapminder$year <= endYear, 'gdpPercap']

  mean(selectedCountryData)
}

gdpUSA <- getAverageGdpPerCapita('United States', 1980, 1989)
gdpCanada <- getAverageGdpPerCapita('Canada')

print(paste('GDP of USA is', gdpUSA))
print(paste('GDP of Canada is', gdpCanada))
```

Let’s say that if start and end year are not specified, we want to default to a start year of 1952 and an end year of 2007.

```r
getAverageGdpPerCapita <- function(country, startYear = 1952, endYear = 2007) {
  # extract gdpPercap from the gapminder data for the specified country.

  selectedCountryData <- gapminder[gapminder$country == country &
```

```
                                     gapminder$year >= startYear &
                                     gapminder$year <= endYear, 'gdpPercap']

  mean(selectedCountryData)
}

gdpUSA <- getAverageGdpPerCapita('United States', 1980, 1989)
gdpCanada <- getAverageGdpPerCapita('Canada')

print(paste('GDP of USA is', gdpUSA))
```

```
## [1] "GDP of USA is 27446.954775"
```

```
print(paste('GDP of Canada is', gdpCanada))
```

```
## [1] "GDP of Canada is 22410.74634"
```

Note the use of an = sign instead of the arrow `<-`.

Show the documentation of the read.csv function.

```
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

That is why

```
file <- 'myfile.cs'
read.csv(file)
```

works.

What if we switch the start and end years?

```
gdpUSA <- getAverageGdpPerCapita('United States', 1989, 1980)
```

It won't work, as R matches the arguments by position. But, we can specify arguments by name as

```
gdpUSA <- getAverageGdpPerCapita('United States', endYear = 1989, startYear = 1980)
```

We can even say

```
gdpUSA <- getAverageGdpPerCapita('United States', endYear = 1989)
```

in which case the start year will take on its default value.

Processing large vectors using the `apply` functions

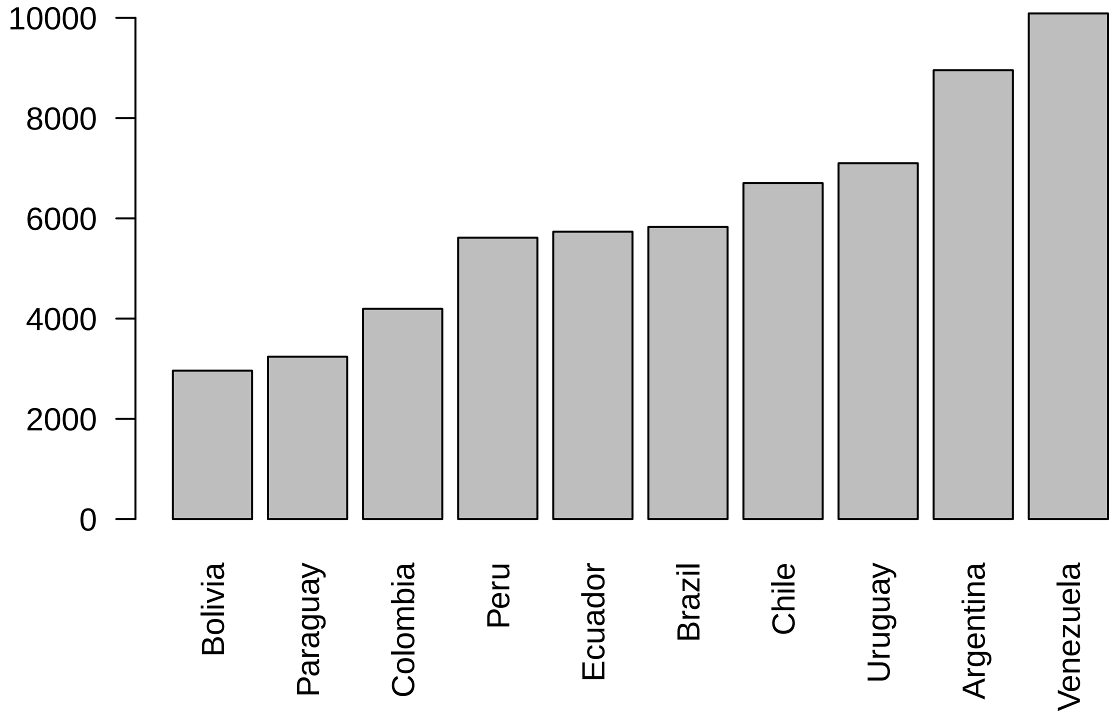
In our later sessions, we are going to see control statements like loops. One of their uses is to go through an array and do a task on its elements one by one. Without getting into loops, we will quickly demonstrate the `apply` family of functions which let us achieve something similar:

```
southAmericanCountries <- c('Argentina', 'Bolivia', 'Brazil', 'Chile', 'Colombia', 'Ecuador', 'Paraguay

# use sapply to invoke getAverageGdpPerCapita() on all elements of southAmericanCountries
averagedGdpSouthAmericanCountries <- sapply(southAmericanCountries, getAverageGdpPerCapita)

# sort them in ascending order
averagedGdpSouthAmericanCountries <- averagedGdpSouthAmericanCountries[order(averagedGdpSouthAmericanCo

barplot(averagedGdpSouthAmericanCountries, las=2)
```



Key Points

- Define a function using `function` name(...params...).
- The body of a function should be indented.
- Call a function using name(...values...).
- Numbers are stored as integers or floating-point numbers.
- Each time a function is called, a new stack frame is created on the call stack to hold its parameters and local variables.
- R looks for variables in the current environment before looking for them at the top level.
- Use help(thing) to view help for something.
- Put docstrings in functions to provide help for that function.
- Annotate your code!
- Specify default values for parameters when defining a function using name=value in the parameter list.
- Parameters can be passed by matching based on name, by position, or by omitting them (in which case the default value is used).

Calling R scripts from the command line

Just like we can call other scripts and functions from our R programs, we can call an R script from the command line. We use the Rscript program to do this.

```
Rscript filename.R
```

If Rscript is not on your system's path, you can invoke Rscript using it's full path name.

Try it out:

script.R:

```
print('This is a simple R script')
```

On the command line:

```
Rscript filename.R
```

Even if you do your data analysis in a different programming language, you can still use R's plotting capabilities (which we will learn more about in the afternoon session). So if the majority of your work is in another programming language, you can still make use of R to do parts of the analysis, or only to make publication quality plots.

We will write a script that will produce a barplot from a given datafile. In the previous example, script.R prints a fixed message and has no way of getting any information from the shell. Rscript allows you to send arguments to the R script by specifying them after the file name:

```
Rscript filename.R argument_1 argument_2 ...
```

These arguments can be accesed from within the R script by using the `commandArgs()` function. By default the list will include the name of the R executable, the script file name, and several switches. Using `commandArgs(FALSE)` only presents us with the arguments starting from `argument_1` onwards.

Updated script.R

```
countryList <- commandArgs(TRUE)

# location of the data
fileName <- 'data/gapminderData.csv'

# read gapminder data
gapminder <- read.csv(fileName)

getAverageGdpPerCapita <- function(country) {
  # extract gdpPercap from the gapminder data for the specified country.

  selectedCountryData <- gapminder[gapminder$country == country, 'gdpPercap']

  mean(selectedCountryData)
}

averagedGdp <- sapply(countryList, getAverageGdpPerCapita)
barplot(averagedGdp)

print(averagedGdp)
```

We now write a shell script that is going to call this method on a list of countries

Create a file makePlots.sh

```
countries="Canada Belgium"
Rscript cmdScript.R $countries
```

Run it from bash:

```
bash makePlots.sh
```

The generated plots are saved in PDF format in the file `Rplots.pdf`.