STAT 209 Lab 1: Introduction to R and RStudio

Overview

The goal of this lab is to introduce you to R and RStudio, which you'll be using throughout the course both to learn the statistical concepts discussed in the course and also to analyze real data and come to informed conclusions.

To clarify which is which:

1. R is the name of the programming language itself
2. RStudio is what's called an Interactive Development Environment (IDE); an interface through which we can use the R language

As the labs progress, you are encouraged to explore beyond what the labs dictate; a willingness to experiment will make you a much better data scientist. Before we get to that stage, however, you need to build some basic fluency in R.

Before we start, let's get RStudio running.

Getting RStudio Running

There are two ways you can run R and RStudio:

1. log in to the Oberlin RStudio server from a web browser, or
2. install the software on your own computer.

Why use the server?

1. you can get to your account, and your files, from anywhere,
2. you don't have to install anything.

Why not use the server?

1. you have to upload and download files to the server if you create them on your computer and want to use them in RStudio, or if you want to do something on your computer with a file that you created in RStudio
2. there may be occasional server outages or slow downs when a lot of people are using it at once.

Logging in to the server

Unless you already have R and RStudio installed on your machine and are used to using it there, you should log in to the server.

Here's how to do it

1. In your web browser, visit rstudiopro.oberlin.edu (rstudiopro.oberlin.edu).
2. Unless you registered since yesterday, you should have an account
    1. Your username should be your Obie ID
    2. the initial password is the same as your username (some old accounts may have had passwords reset)
3. Before moving on, change your password
    1. Tools > Shell…
    2. Type `passwd` at the terminal
    3. Enter your old password, followed by a new password twice (the cursor will not move or show anything, but the computer is registering your keystrokes)
    4. Close the terminal window

RStudio IDE

Today we begin with the fundamental building blocks of R and RStudio: the interface, reading in data, and basic commands.

Environment pane

The panel in the upper right has tabs for your

1. *Environment* (variables and datasets currently loaded in memory for your session) as well
2. *History*: commands that you've previously entered

Later, this is also where we'll interface with `git` and GitHub.

Files, Plots, Packages, Help, Viewer

The pane in the lower right has tabs for

1. *Files*: The files stored in your user account (by default shows your home directory; you can create folders within here to organize your stuff). This is where you can Upload files from your computer ("Upload" button), or download files to your computer ("More" > "Export").
2. *Plots*: Shows plots you generate in the console or in a script
3. *Packages*: Shows R packages currently installed, provides a graphical interface to install new ones, and load installed packages into your session (more on this below). I discourage using the checkboxes here to do the latter; better to load packages using commands in your documents, so it is recorded that you are using that package

The Console

The panel on the left is called the *console*. When you first launch RStudio, it shows whate version of R that you're running, etc.

Below that information is the *prompt*. As its name suggests, this prompt is really a request, namely a request for a command: a line of R code that tells the computer to do something.

R Packages

R is an open-source programming language, meaning that users can contribute packages that make our lives easier, and we can use them for free. Much of what we need for this course is contained in a particular “meta-package” (a package containing a collection of other packages) called the `tidyverse`.

Within the `tidyverse`, two particularly important packages are

1. dplyr: for data wrangling
2. ggplot2: for data visualization

The tidyverse is already available on the RStudio server. If you are working on your own local installation of R and RStudio, you may need to install it, by typing the following line of code into the console and pressing the enter/return key.

Note that you can check to see which packages (and which versions) are installed by inspecting the Packages tab in the lower right panel of RStudio. **Don’t install packages yourself if on the server: This could result in two different versions being installed at once, causing you to see different behavior than others in the class**

If and only if you’re working on a local installation, type the following at the console.

```
install.packages("tidyverse")
```

You may need to select a server from which to download; any of them will work. You only need to run the above line once per computer.

Next, you need to load these packages in your working environment. You will need to do this once for each new lab, project, or report that you create. The best way to do that is by including the command that loads the package of interest right at the top of the file that contains your work. So we need to know how to create and edit such a file.

R Scripts

Go to the File menu, and under “New File” select “R Script”.

This will cause the left-hand part of R Studio to split in half, the console moving to the bottom, and a new pane appearing in the upper left that says “Untitled1” above it.

This pane just has a text editor, currently showing a blank file, where you can create a list of R commands that you want to run to perform an analysis or create your data graphic.

Since many of the commands we will want to use are supplied by the `tidyverse` package, we should first load it.

We do this with the `library()` function. Type the following line at the top of your new R script.

```
library(tidyverse)
```

Before we get any further, let's save the script. Select File > Save As… and give the script a name which ends in `.R`, the standard extension for R scripts. This ending will tell RStudio, when you go to open this file later, to treat it as an R script. Let's call it `lab1.R`.

We haven't actually loaded the `tidyverse` package. To run that line, place your cursor anywhere on the line and hit Ctrl-Enter on your keyboard (or Cmd-Enter on a Mac). There is also a "Run" button you can click on, but it is much faster to get used to using the keyboard for things you will be doing often.

You should see the command appear in the console, and some messages below it telling you what's happening when R is loading the package.

If you looked in the Packages tab on the lower right, you would see that `tidyverse` (as well as the individual packages it contains) has a checkmark next to it, indicating that the commands it provides are now available to use in this session.

Loading Data from a File

Since we want to be working with data, let's load a dataset into our environment.

Reading data directly from the web

To get you started, put the following command on the next line in your script. This will load a data set from a file on the web.

```
arbuthnot <- read_csv("http://www.openintro.org/stat/data/arbuthno
  t.csv")
```

This command instructs R to fetch the data from the given URL, and put it in an object in our environment called `arbuthnot`, which will now appear in the Environment pane.

Reading data from a local file

You can also point to a file on your computer by supplying the file path in place of the URL.

Absolute paths

If instead of a URL you put something like `"~/data/some_fun_data.csv"` inside the parentheses in `read_csv()`, it would look for a file called `some_fun_data.csv` in the `data` subdirectory of your Home folder (which is abbreviated by the `~` symbol).

A location that starts with a forward slash `/` or a tilde `~` is an **absolute path**. That is, it doesn't matter what directory (folder) you're currently working in; it will look in that specific location.

Relative paths

Sometimes it is convenient to specify a file using a **relative path** instead. For example, when logging in to the RStudio server for the first time, we are by default working in our home directory (`~`), and so I could replace the absolute path above with `data/some_fun_data.csv`.

If I'm working in `~/data`, or if my file is directly inside my home folder, I could just type `some_fun_data.csv`.

If I'm working in `~/data` and want to reference something inside my home folder, I can use two periods as shorthand for "the directory containing this one". So, relative to `~/data`, the path `../more_fun_data.csv` refers to `~/more_fun_data.csv`. I can even do `../misc/yet_more_data.csv` to refer to `~/misc/yet_more_data.csv`.

Your working directory

You can see what directory is currently set as your working directory (relative to which all relative paths are interpreted) by typing `getwd()` in the console.

You can change your working directory (for example, to `~/data`) by typing `setwd("~/data")` (with the quotes).

Interacting with data sets

The `arbuthnot` data that we read in from the web consists of the Arbuthnot baptism counts for boys and girls, which you may have read about in the textbook.

You should see that the workspace area in the upper righthand corner of the RStudio window now lists a data set called `arbuthnot` that has 82 observations on 3 variables.

As you interact with R, you will create a series of objects. Sometimes you load them as we have done here, and sometimes you create them yourself as the byproduct of a computation or some analysis you have performed. Note that because you are accessing data directly from the web, this command (and the entire assignment) will work in a computer lab, in the library, or in your dorm room; anywhere you have access to the Internet.

The Arbuthnot data set refers to Dr. John Arbuthnot, an 18th century physician, writer, and mathematician. He was interested in the ratio of newborn boys to newborn girls, so he gathered the baptism records for children born in London for every year from 1629 to 1710. We can take a look at the data by typing its name into the console.

```
arbuthnot
```

However printing the whole dataset in the console is not that useful. One advantage of RStudio is that it comes with a built-in data viewer. Click on the name `arbuthnot` in the Environment pane (upper right window) that lists the objects in your workspace. This will bring up an alternative display of the data set in the Data Viewer (upper left window). You can close the data viewer by clicking on the "X" in the upper lefthand corner.

What you should see are four columns of numbers, each row representing a different year: the first entry in each row is simply the row number (an index we can use to access the data from individual years if we want), the second is the year, and the third and fourth are the numbers of boys and girls baptized that year, respectively. Use the scrollbar on the right side of the console window to examine the complete data set.

Note that the row numbers in the first column are not part of Arbuthnot's data. R adds them as part of its printout to help you make visual comparisons. You can think of them as the index that you see on the left side of a spreadsheet. In fact, the comparison to a spreadsheet will generally be helpful. R has stored Arbuthnot's data in a kind of spreadsheet or table called a *data frame*.

You can see the dimensions of this data frame by typing the following:

```
glimpse(arbuthnot)
```

Since we are not actually modifying our workspace with this command, just viewing something, we can do this at the console or in our script. If we were actually modifying something, however, we would want to put the command in our script so we remember to run it the next time we run the script.

This command should output the following

```
## Observations: 82
## Variables: 3
## $ year  <dbl> 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 16
37, 1638…
## $ boys  <dbl> 5218, 4858, 4422, 4994, 5158, 5035, 5106, 4917, 47
03, 5359…
## $ girls <dbl> 4683, 4457, 4102, 4590, 4839, 4820, 4928, 4605, 44
57, 4952…
```

We can see that there are 82 observations (aka "cases") and 3 variables in this dataset. The variable names are `year` , `boys` , and `girls` .

At this point, you might notice that many of the commands in R look a lot like functions from math class; that is, invoking R commands means supplying a function with some number of arguments. The `glimpse()` command, for example, took a single argument, the name of a data frame, and returned some output, much as the `sin()` function would take a number and return a number corresponding to its trigonometric sine.

Some Exploration

Let's start to examine the data a little more closely. We can access the data in a single column of a data frame separately using a command like

```
pull(arbuthnot, boys)
```

This command will only show the number of boys baptized each year. In a sense, `pull()` extracts the variable `boys` from the data frame `arbuthnot`.

Exercise 1 What command would you use to extract just the counts of girls baptized? Try it!

Notice that the way R has printed these data is different. When we looked at the complete data frame, we saw 82 rows, one on each line of the display. This data is no longer structured in a table with other variables, so the values are displayed one right after another.

Objects that print out in this way are called *vectors*; they represent an ordered collection of numbers.

R has added numbers in [brackets] along the left side of the printout to indicate locations within the vector. For example, `5218` follows `[1]`, indicating that `5218` is the first entry in the vector. And if `[43]` starts a line, then that would mean the first number on that line would represent the 43rd entry in the vector.

Data visualization

R has some powerful functions for making graphics. We can create a simple plot of the number of girls baptized per year with the command

```
qplot(x = year, y = girls, data = arbuthnot)
```

The `qplot()` function (meaning "quick plot") considers the type of data you have provided it and makes the decision to visualize it with a scatterplot. The plot should appear under the Plots tab of the lower right panel of RStudio. Notice that the command above again looks like a function, this time with three arguments separated by commas. The first two arguments in the `qplot()` function specify the variables for the x-axis and the y-axis and the third provides

the name of the data set where they can be found. If we wanted to connect the data points with lines, we could add a fourth argument to specify the geometry that we'd like.

```
qplot(x = year, y = girls, data = arbuthnot, geom = "line")
```

You might wonder how you are supposed to know that it was possible to add that fourth argument. Thankfully, R documents all of its functions extensively. To read what a function does and learn the arguments that are available to you, just type in a question mark followed by the name of the function that you're interested in. Try the following.

```
?qplot
```

Notice that the help file replaces the plot in the lower right panel. You can toggle between plots and help files using the tabs at the top of that panel.

Exercise 2 Is there an apparent trend in the number of girls baptized over the years? How would you describe it?

Functions, arguments and commands

Most of what we do in R consists of applying **functions** to data objects, specifying some options for the function, which are called **arguments**. Together, the application of a function, together with its arguments, is called a **command**.

A useful analogy is that commands are like sentences, where the function is the verb, and the arguments (one of which usually specifies the data object) are the nouns. There is often a special argument that comes first. This is like the direct object of the command.

For example, in the English command, "Draw a picture for me with some paint", the verb "draw" acts like the function (what is the listener supposed to do?); the noun "picture" is the direct object (draw what?), and "me" and "paint" are extra (in this case, optional) details, that we might call the "recipient" and the "instrument".

In the grammar of R, I could write this sentence like:

```
## Note: this is not real R code
draw("picture", recipient = "me", material = "paint")
```

We are applying the function `draw()` to the object `"picture"`, and adding some additional detail about the recipient and material. Here the function is called `draw`, and we have a main argument with the value `"picture"`, and

additional arguments `recipient` and `material` with the values `"me"`, and `"paint"`, respectively.

Technically speaking, `"picture"` is the value of an argument too; we might have written

```
### Note: this is not real R code
draw(object = "picture", recipient = "me", material = "paint")
```

However, in practice, there is often a required first "main" argument whose name is left out of the command.

In R, arguments always go inside parentheses, and are separated by commas when there is more than one. For arguments whose names are explicitly given, the name goes to the left of the `=`, and the value goes to the right.

Creating new variables

Now, suppose we want to plot the total number of baptisms. To compute this, we could use the fact that R is really just a big calculator. We can type in mathematical expressions like

```
5218 + 4683
```

to see the total number of baptisms in 1629. We could repeat this once for each year, but there is a faster way. If we add the vector for baptisms for boys and girls, R will compute all sums simultaneously.

```
pull(arbuthnot, boys) + pull(arbuthnot, girls)
```

What you will see are 82 numbers (in that packed display, because we aren't looking at a data frame here), each one representing the sum we're after. Take a look at a few of them and verify that they are right.

We can make a plot of the total number of baptisms per year with the command

```
qplot(x = year, y = boys + girls, data = arbuthnot)
```

To keep our data organized, and to facilitate plotting, we want to save this variable as a column in our data frame. We can do this as follows:

```
arbuthnot <- arbuthnot %>% mutate(total = boys + girls)
```

The `%>%` operator is called the "piping" operator. It takes the output of the previous expression and "pipes" it into the first argument of the function in the following one. To continue our analogy with mathematical functions, `x %>% f(y)` is equivalent to `f(x, y)`.

We can read this command as the following:

1. "Take the `arbuthnot` dataset and **pipe** it into the `mutate()` function.
2. Mutate the `arbuthnot` data set by creating a new variable called `total` that is the sum of the variables called `boys` and `girls`.
3. Then **assign** the resulting dataset to the object called `arbuthnot`, i.e. overwrite the old `arbuthnot` dataset with the new one containing the new variable.

When you make changes to variables in your dataset, click on the name of the dataset again to update it in the data viewer.

The special symbol `<-` performs an **assignment**, taking the output of one line of code and saving it into an object in your workspace. In this case, you already have an object called `arbuthnot`, so this command updates that data set with the new mutated column.

We can make a plot of the total number of baptisms per year with the command

```
qplot(x = year, y = total, data = arbuthnot, geom = "line")
```

Exercise 3 Combine the `mutate()` idea and the `qplot()` syntax to create a plot of the *proportion* of boys baptized over time. What do you see?

Getting credit

To get credit for this lab, respond to the following two questions by Direct Message to me on Slack by Tuesday 9/10 at 3:00 P.M. Tag your answers somehow as being about lab1, and also include the Honor Pledge with your message.

1. What do you take away from the plot you made of the proportion of boys born over time?
2. In no more than a sentence or two, what were the main ideas you took away from this lab? As you worked through the lab, where did you get stuck? What is still giving you trouble?