Introduction to R and RStudio

R Clay Wright

March 31, 2016

The goal of this paricular tutorial is to teach you how to efficiently and effectively use RStudio to teach yourself how to do anything you may want to do with R. If you would like a more in-depth R tutorial check out [Code School](http://tryr.codeschool.com/).

# The R programming language

R is a programming language for statistical computing and graphics generation. R is based on an older statistical computing language called S. It is freely available and open source. It can be downloaded [here](http://www.r-project.org).

R is a line-oriented language, meaning each line of code is run or interpreted independently (if you are familiar with C you will know each bit of code must be followed by a semicolon). You can run R scripts from the command line or terminal if you are really hardcore, but another benefit of R its beautiful integrated development environment (IDE, a program that provides a clean work space and numerous tools to assist you in coding).

## RStudio

This IDE is called RStudio and can be downloaded [here](http://www.rstudio.com). R and RStudio are already installed on the computers in the BSA, so go ahead and open up RStudio now.

You will see several panes within the RStudio window. First lets focus on the **Console** pane.

### The Console pane

This pane is essentially a R command line window. The only difference between this pane and running R from the terminal is that this pane has an auto-complete feature. Try typing pri into the **Console**. RStudio provides you with a list of all the available functions and variables beginning with 'pri'. What are functions and variables? Let's take a brief aside...

Variables and Functions

R is an object oriented language like most other modern programming languages. This means we can use R to create abstracted objects that contain data (of any type, shape or size) called variables or procedures/methods (individual blocks of code) commonly called functions. There are numerous functions and datasets included in the base R installation. Also, as an open source language countless programmers in the R community have written useful functions and created useful datasets that are freely available in the form of R-packages (more on these later). You can also write your own!

OK, back to the list next to pri in the console...

You can navigate this list using the arrow keys or your mouse. When you select a particular object RStudio also gives you some information about that object. Navigate down to print (if there are multiple, select the one that has {base} at the far right) and press tab and then return. You will see that RStudio has added parentheses to the right of print in the console. This is because print is a function, but what does this function do? To figure out type ?print in the console and press return. This opens the documentation for this function in the **Help** pane. A ? before any function name will do the same.

### The Help pane

This pane is essentially a browser window for R documentation. You can also search for functions or variables in R and all of the installed packages on your computer using the search box at the top. You can search within a documentation page using the *Find in Topic* box.

All R documentation follows standard formatting. **Description** is pretty self explanatory. **Usage** demonstrates how you use the function, sometimes with specifics for different variable types. For print this shows us that print takes the input argument x (an argument is just variable that is used in a function). If x is a 'factor' or a 'table', print will also take some additional arguments. In the **Usage** section the default value of each argument is listed (e.g. FALSE is the default value for argument quote). A description of each argument is listed below in the **Arguments** section. **Value** is the type of data returned by the function. There are a few other self-explanatory sections and finally **Examples**. This is often one of the most useful sections as it shows you how to use the function. The code in **Examples** can be copied and pasted into the console and run.

##### ???Questions???

Choose any function and open its documentation in the **Help** pane.

Copy the **Usage** section of your chosen function below.

#### OK, back to the Console...

> print() Put your cursor in the middle of the parentheses and press tab. RStudio will feed you all of the arguments of this function using auto-complete! Press tab again and x = will appear in the parentheses. Type "hello world" and press return. Amazing! Your first line of R code worked, hopefully...

Take note that if you are missing the quotes around hello world R will look for a variable named hello and return an error.

The console will allow you to execute one line of code at a time, but in order to do anything of value you will need tens if not hundreds of lines of code!

### The Source pane

Fortunately, multiple lines of code (scripts) can be edited, executed and saved as .R files in the **Source** pane. To create a new .R file press shift + command + N, or click the universal 'New Document' icon and select 'R script'. Copy and paste your first line of code from the console (the > isn't necessary in the source pane and will actually break your code). With your cursor in this line of code in the **Source** pane press command + return. This will execute the line in the **Console** pane. Now just select x = "hello world" and press command + return. So `command

### The Environment pane

You just created your first variable object in R! See over on the top right next to x is "hello world". You can now execute just print(x), and you will get [1] "hello world". The **Environment** pane shows all of the objects you have created or stored in memory. You can view data or functions by double clicking on them.

## Variables and data types

You can create objects (variables~values, large data structures~think spreadsheets and databases, and functions) using the =, <- or -> operators. You can see what type of data (or data type) a variable is using the class function. Go ahead, class(x). Data in R can be of several different basic types:

|  |  |  |
| --- | --- | --- |
| Data Type | aka | Example |
| Logical | boolean | TRUE, FALSE |
| Numeric | float | 12.3, 5, 999 |
| Character | string | 'a' , '"good", "TRUE", '23.4' |
| Integer |  | 2L, 34L, 0L |
| Complex |  | 3 + 2i |
| Raw |  | Hexedecimal values, "Hello" is stored as 48 65 6c 6c 6f |

### Vectors

Never fear we are not talking about vector calculus here. Vectors in R are simply ordered lists of values. These values can be of any type (strings, numerics, booleans, etc), but they must all be of the same type, or R will force them to be the same. We can construct vectors using the c() function.

##### ???Question???

What is c abbreviating? (i.e. what is the title of the c() function?)

What are the arguments that you can pass to c()?

Let's run through a quick example:

strings <- c('hello', 'world')  
strings

## [1] "hello" "world"

Now we have a vector of strings. We can access the individual elements using the square bracket operator.

strings[1]

## [1] "hello"

strings[2]

## [1] "world"

#Note that the indices begin at 1 in R!!!  
strings[0]

## character(0)

We can also change elements or add elements to the vector using the bracket operator.

strings[2] <- 'good'  
strings[3] <- 'bye'  
strings

## [1] "hello" "good" "bye"

strings[4] <- FALSE  
strings

## [1] "hello" "good" "bye" "FALSE"

##### ???Question???

What happened to FALSE?

Write a block of code to test what would happen if we instead added a character string to a vector of logical values! What happens?

bools <- c(FALSE, TRUE)  
bools[3] <- 'hello'  
bools

## [1] "FALSE" "TRUE" "hello"

### Matrices, Arrays and Lists

**Matrices** are simple two dimensional data sets and **Arrays** are N-dimensional data sets. Like vectors these must be made of a single data type. For more info ?matrix and ?array.

Lists are more complex data structures that are similar to vectors but allow multiple data types. Lists can contain vectors as elements and even other lists! This makes them potentially N-dimensional but clunky to work with. You might encounter them if you use R in the future. For more info ?list.

### Data frames

Variables in R are not limited to just strings or integers or even matrices. You can store and operate on entire spreadsheets with fields of defined data types, using what R calls 'data frames'. The data frame is one of the most fundamental data structure used in R. ?data.frame provides a wealth of knowledge about data frames, but let's just go ahead and make one!

L3 <- LETTERS[1:3]  
fac <- sample(L3, 10, replace = TRUE)  
d <- data.frame(x = 1, y = 1:10, fac = fac)  
#notice how the columns of the data frame can be named using '=', just as if we were creating individual vectors  
d

## x y fac  
## 1 1 1 A  
## 2 1 2 A  
## 3 1 3 B  
## 4 1 4 C  
## 5 1 5 C  
## 6 1 6 A  
## 7 1 7 A  
## 8 1 8 A  
## 9 1 9 B  
## 10 1 10 B

class(d)

## [1] "data.frame"

##### ???Question???

What is LETTERS?  
What does sample do?

Now we have a data frame d with 10 rows and 3 columns. You can retrieve individual columns using the $ operator. Try it, d$fac!. Wait a minute, why is this no longer a column?

##### ???Question???

What data type is d$fac?

class(d$fac)

## [1] "factor"

### Factors

Factors used to be an efficient way of storing large vectors of repetitive discrete or categorical data. Factors do this by translating the potentially long individual pieces of data into integers, using a table called levels. Try levels(d$fac). So R will use this key with 1 = A, 2 = B, 3 = C, to read and write this factor. To see how R sees d$fac we can use as.integer(d$fac). R now stores large data strucutres by indexing values like this regardless of whether it's a factor of not. Despite this fact there are still some useful features of factors.

For one, factors can only take on values within the levels vector. This means that if you are manually entering things like genotype as you are collecting your data, R can help alert you to spelling mistakes.

Giving ?factor a look, you will see that we can also assign a particular order to the levels of a factor. This can be handy for ordering variables when plotting. We can also assign labels to the levels, just in case your level names are too abstracted to be understandable.

However when manipulating data frames contaning factors you must be careful because some functions may interpret factors as their integer values! We could also avoid creating factor in our data frame by including stringsAsFactors = F in our call to data.frame().

Going back to our data frame d, similar to vectors we can access rows, columns and elements of the data frame using the square bracket operator.

#get the first row of d  
d[1,]

## x y fac  
## 1 1 1 A

#get the first column of d  
d[,1]

## [1] 1 1 1 1 1 1 1 1 1 1

#get the column named 'fac'  
d[,'fac']

## [1] A A B C C A A A B B  
## Levels: A B C

#or  
d[['fac']]

## [1] A A B C C A A A B B  
## Levels: A B C

#or (most efficient and readable)  
d$fac

## [1] A A B C C A A A B B  
## Levels: A B C

#get the element in the 5th row and 3rd column  
d[5,3]

## [1] C  
## Levels: A B C

We can also perform calculations or other operations on the elements of a dataframe.

d[,2] + 1

## [1] 2 3 4 5 6 7 8 9 10 11

d[[2]] + 1

## [1] 2 3 4 5 6 7 8 9 10 11

d[,2] \* 2

## [1] 2 4 6 8 10 12 14 16 18 20

#similarly for logical operations, note that logical 'is equivelent to' is '=='  
d[,3] == 'B'

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

d$y <= 5

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

Just like with vectors we can change elements or add elements to a data frame.

##### ???Question???

How would you add a column to d with the integer values representing d$fac?

d$faci <- as.integer(d$fac)

## Importing and exporting data

Enough with these toy examples, let's look at some real data! Below we will analyze some real data from the assigned paper this week.

R provides convenient functions for reading data of many types into memory. These functions include read.csv and read.table along with many others. R also has several packages of functions for analysing flow cytometry data. As you discussed last week the Nemhauser lab has pioneered a technique by which plant signaling network dynamics can be measured using flow cytometry.

### Flow Cytometry

As you might remember from digging deep into the methods of last weeks paper (Havens et al.), flow cytometry measures light scattering, absorption, and emmision of single cells, by passing a stream of cells through the path of a laser in rapid succession. Light scattered from the laser beam is detected as a pulse beginning when the cell enters the laser beam peaking when the cell is centered in the beam, and ending when the cell leaves the beam. The cytometer collects the maximum height of the signal from this pulse (H), the total area of the pulse (A) and the width of the pulse (W). Light is scattered in all directions as the cell passes through the beam and how much light is scattered and in what direction can tell us a lot about a cell. Typically, light that passes through the cell is measured as forward-scattered light or 'FSC' and this is considered proportional to a cells size. The cell actually acts as a lense bending the light, therefore the smaller the cell the more light is bent away from the forward-scatter detector, whereas larger cells allow more light to pass straight through. Dead cells appear very low on the FSC axis.

Side-scattered light or 'SSC' is the light that is reflected perpedicular, or at a wide angle, relative to the laser beam. SSC is proportional to the internal complexity, or granularity, of the cell. Imagine all of those tiny, circuitous, membrane-bound organelles acting as little lenses bending light in all sorts of directions.

Similarly side-scattered light at different wavelengths from the laser is collected as a measure of the fluorescence of the cell. Because light is scattered at all angles you can measure fluorescence at many wavelengths. For our example experiment below, the fluorescence of the Aux/IAA-fluorescent-protein fusion is detected as 'FL1'.

Let's install a few packages so we can analyze some flow cytometry data.

### Packages

There are three places where R packages are available:  
1. [CRAN](http://cran.r-project.org/web/packages/) contains a huge variety of general packages,  
2. [bioConductor](http://bioconductor.org/) contains packages related to high throughput biological (mostly -omics) data,  
3. Packages in development may be available from code repositories such as [GitHub](http://github.com), [Bitbucket](http://bitbucket.org) or others.

#install a package from CRAN (if you exclude repos it will give you a list and ask you to select a repository server)  
install.packages('plyr', repos = 'https://cran.fhcrc.org/')  
  
#installing from bioConductor is a little more complicated  
source("https://bioconductor.org/biocLite.R")  
biocLite()  
#you only have to run the above 2 lines the first time you use bioConductor during an R session  
#install the package 'flowCore'  
biocLite('flowCore')  
  
#installing from a code repository requires the 'devtools' package from CRAN  
install.packages('devtools')  
#install a package from github  
devtools::install\_github('wrightrc/flowTime')

Now that we have installed these packages we can load some flow cytometry data using the read.flowSet function.

library(flowTime)  
flowSet <- read.flowSet(path = 'flowSet', alter.names = T)

##### ???Question???

What does alter.names = T do?

?read.flowSet  
*It renames the columns to valid R names using the make.names function.*

At the moment all of the information that we have about this data is the fluorescence measurements and the time at which each well of the plate was read. We need to annotate this data with the information about the strain and treatment that was loaded into each well. We have recorded this information in a csv file (made using Excel). We need attach this spreadsheet to our flowSet so that in the future we (and our collaborators or lab mates) can simply load this information in with the flowSet.

annotation <- read.csv('annotation.csv')

##### ???Questions???

Write a line of code to figure out the column names of annotation.

What are data is in annotation?

names(annotation)

## [1] "file" "strain" "treatment" "grp" "afb" "rep"

Before attaching this to our flowSet, we need to make sure that we have a unique identifier (a column that is identical in the flowSet and in the annotation). The easiest way to do this is to pull the sampleNames from flowSet and add them to annotation.

annotation$name <- sampleNames(flowSet)

We should also make sure that the order of sampleNames matches the order of annotation. Double click on annotation over in the **Environment** pane. Quickly scroll through to make sure that the file and name fields match.  
Now we can attach this annotation to the flowSet using annotateFlowSet.

flowSet <- annotateFlowSet(flowSet, annotation\_df = annotation, mergeBy = 'name')

We can then save this flowSet so that we and others can easily re-analyze it later. (You don't need to run the below line)

write.flowSet(flowSet, outdir = 'flowSet')  
#To read in the annotated flowSet used  
read.flowSet(path = 'flowSet', phenoData = 'annotation.txt', alter.names = T)

Let's take a quick look at what this raw flow cytometry data looks like using the flowViz package.

bioClite('flowViz')  
library(flowViz)

One of the easiest ways to learn how to use a new package is to go through the vignettes. These are little examples of how to use the package written by the package author.

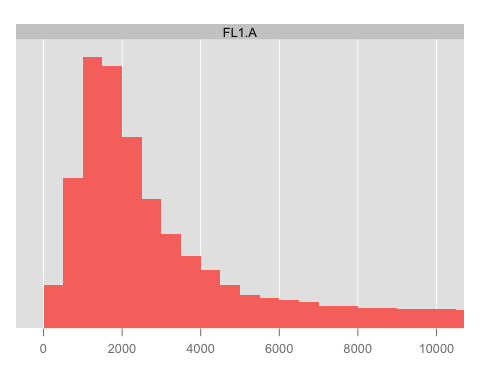
#what vignettes are available for flowViz?  
vignette(package = 'flowViz')  
#Let's give the 'filters' vignette a look  
vignette('filters', package = 'flowViz')  
#Make a couple of plots of a single flowFrame  
xyplot(SSC.H~FSC.H, data = flowSet[['0\_A01.fcs']], xlim = c(0, 6E6), ylim = c(0, 6E5), smooth = F)  
xyplot(FL1.A~FSC.H, data = flowSet[['0\_A01.fcs']], xlim = c(0, 6E6), ylim = c(0,1E4), smooth = F)  
histogram(~FL1.A, data = flowSet[['0\_A01.fcs']], xlim = c(0,1E4), margin = F, type = 'count', breaks = 500)

You can save plots by clicking export from the **Plots** pane.

##### ???Question???

Make a histogram plot of the same strain we plotted above, but after it has been treated with 10 uM auxin.

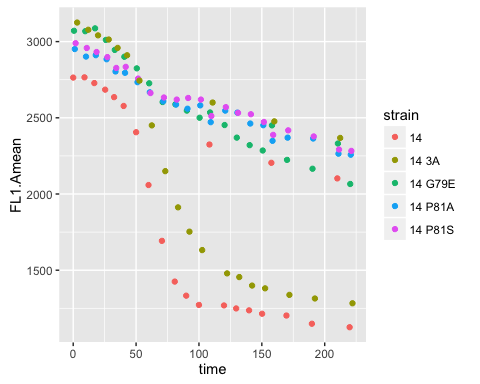
histogram(~FL1.A, data = flowSet[['3\_F01.fcs']], xlim = c(0,1E4), margin = F, type = 'count', breaks = 500)



loadGates(gatesFile = 'C6Gates.RData')  
dat\_sum <- summarizeFlow(flowSet, ploidy = 'diploid', only = 'singlets',channel = 'FL1.A')

## [1] "Gating with diploid gates..."  
## [1] "Summarizing singlets events..."

qplot(time, y = FL1.Amean, data = dat\_sum, color = strain)

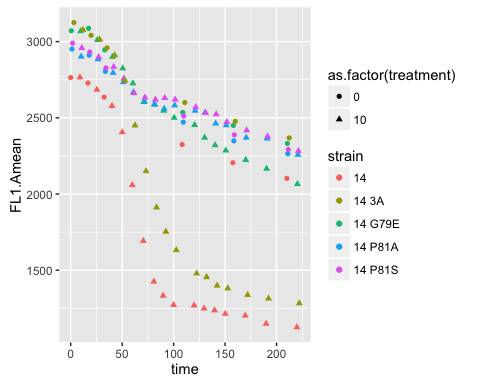
 \*\*\* #####???Questions???

qplot(time, y = FL1.Amean, data = dat\_sum, color = strain, shape = treatment)

What is causing the error returned by running the above line of code? What does this mean?

*The continuous variable treatment cannot be mapped to shape, which is a discrete set of shapes. A continuous variable can take on an infinite number of values whereas a discrete variable can only take on a set number of possible values. Because treatment is a numeric variable here it is continuous and therefore cannot be mapped by the discrete set shape.* \*\*\*

qplot(time, y = FL1.Amean, data = dat\_sum, color = strain, shape = as.factor(treatment))



##### ???Question???

What is the default geom argument in the above call to qplot?

Try adding a different geom argument in the above line.

Does this improve how the data is visualized? Why or why not?