Introduction to R and RStudio

R Clay Wright

2017-03-11 16:52:07

Table of Contents

# Using this document

The goal of this particular 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/).

Keep in mind while navigating this tutorial that you are supposed to be experimenting with R, making mistakes, causing errors and figuring out why. Nothing you do in RStudio will break your computer. Coding is a trial-and-error process, especially as you are learning. Try anything! See what happens! Learn from everything!

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

Throughout the tutorial you will find several sections with this label followed by questions for you to answer. The answers to these questions or the methods by which you can find the answers have already been covered in the above sections. Replace the *answer here* text with your answer along with any code you may have used to arrive at that answer and or a plot you have produced. **Save this document and email it to yourself and/or save it on a flash drive or in your google drive.** You will be working on the same document on your own after class.

The text within this tutorial is formatted to differentiate code from dialog. text that looks like this is R code or keyboard commands. You can copy and paste this code into an R console and execute it. Within multiline blocks of code functions will be colored blue, comments will be colored green, strings will be teal and Booleans will be red. If you don't know what any of these terms mean yet, don't worry, we'll get to them.

In some cases the output from the code will be included in blocks of code. This output will be prefixed with #>.

Let's get started!

# 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 for your personal computer [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/C++ you will know each bit of code must be followed by a semicolon). Just one of the many great things about R is its beautiful integrated development environment (IDE, a program that provides a clean work space and numerous tools to assist you in coding). A good IDE can make learning a language much easier, and even if you are an experienced coder a good IDE can make you much more efficient.

## RStudio

This wonderful IDE for R is called RStudio. For your personal computer it can be downloaded [here](http://www.rstudio.com). Let's take a look around the RStudio window.

You will see several panes within the RStudio window. ###The Console pane We'll start with the **Console** pane. It is in the bottom-left corner labeled **Console** at the top left. (If the **Source** pane is closed, the **Console** pane may take up the whole left side of the window). The **Console** is essentially an R command line window. You can type in anything following the > then press return (or Enter) and whatever you typed will be interpreted by R and the appropriate output will be returned. R understands all basic algebra as well as logical expressions (aka Boolean expressions). Try running the following lines in the console!

2+2

#> [1] 4

9/3

#> [1] 3

3^3

#> [1] 27

2>3

#> [1] FALSE

2<3

#> [1] TRUE

### The Source pane

In the top left of the screen is the **Source** pane. (If you don't see this pane, create a new .R file by pressing shift + command(or ctrl) + N, or click the universal 'New Document' icon and select 'R script'). **Source** shows the '.R' files containing the scripts (multiple lines of related code) with which you are currently working. This is where you will spend most of your time building and testing lines of code and then combining them to make scripts for data analysis.

In addition to being a basic calculator, R interprets functions or variables that can be assigned to and represented by words. Variables (any peice of data of any variety) will be single words possibly followed by a $ or square brackets [] if the variable is a matrix (more on those symbols later). Functions are followed by parentheses containing the arguments passed to the function.

#### 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!

The big difference between using RStudio and running R from the command line is that this pane has an auto-complete feature. Try typing pri into R-script in the **Source** pane. RStudio automatically provides you with a list of all the available functions and variables beginning with 'pri'!

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. You will see that RStudio has completed print in the console and added a set of parentheses because print is a function, print(). Now we can add arguments that this function will operate on, within the parentheses. 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, or passing a function name to the help() function 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.

Using this pane you should be able to answer almost any question you have about any R function.

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.

### The Files/Plots/Packages/Help/Viewer pane

The **Help** pane contains additional tabs that can also be quite helpful. **Files** allows you to navigate through folders on your computer and open files. **Plots** shows you the most recent plot your code has produced and allows you to save it. **Packages** allows you to install and load packages into memory (more about packages later).

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

Choose any function (use tab complete to find one starting with your favorite letter) and open its documentation in the **Help** pane. What function did you choose? *answer here*

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

Additionally, there is a search engine specific to R resources including the documentation, blogs, books and questions users have asked on discussion boards. This invaluable resource is at [Rseek.org](http://rseek.org). This is especially helpful if you want to find a function to perform a specific task.

#### OK, back to the Source pane...

You should have print() in there now.  
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 command/ctrl + return. This will copy your line of code into the console and execute it. Amazing! Your first line of R code worked, hopefully...

print("hello world")

#> [1] "hello world"

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

If your code didn't work, try to fix it and run it again.

command + return (ctrl + enter on Windows machines) can be used to execute a whole line or any selected code. To prevent code from being executed, or take note about what your code does or how to use your code you can make "comments" by prepending any text with a #. Now just select x = "hello world" and press command + return (or ctrl + enter). You just created your first variable object in R!

### The Environment pane

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 sets or functions by clicking on them, but at the moment we only have the simple variable x. Don't worry, we'll practice this later.

## 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, try class(x). Data in R can be of several different, basic types:

|  |  |  |
| --- | --- | --- |
| Data Type | aka | Example |
| Logical | Boolean | TRUE, FALSE |
| Numeric | float | 42, 3.14, |
| Character | string | 'a' , "good", "TRUE", '23.4' |
| Integer |  | 2L, 34L, 0L |
| Complex |  | 3 + 2i |
| Raw | Hexadecimal | "Hello" is stored as 48 65 6c 6c 6f |

### Vectors

Vectors in R are simply ordered lists of values. These values can be of any type (strings, numerics, Boolean, 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.

Let's run through a quick example:

string\_vector <- c('plant', 'genotype')  
string\_vector

#> [1] "plant" "genotype"

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

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

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

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

string\_vector[1]

#> [1] "plant"

string\_vector[2]

#> [1] "genotype"

#Note that the indices begin at 1 in R!!!  
string\_vector[0]

#> character(0)

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

string\_vector[2] <- 'phenotype'  
string\_vector[3] <- 'root\_length'  
string\_vector

#> [1] "plant" "phenotype" "root\_length"

string\_vector[4] <- FALSE  
string\_vector

#> [1] "plant" "phenotype" "root\_length" "FALSE"

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

What happened to FALSE (is it a boolean)? *answer here*

Write a block of code to test what would happen if we instead added a character string to a vector of logical values (i.e. make a new variable containing a few Boolean values, then add a string to that vector)! What happens? *answer here*

### 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 structures used in R. ?data.frame provides a wealth of knowledge about data frames, but let's just go ahead and make one! Run the following code.

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 B  
#> 2 1 2 B  
#> 3 1 3 B  
#> 4 1 4 B  
#> 5 1 5 C  
#> 6 1 6 C  
#> 7 1 7 C  
#> 8 1 8 A  
#> 9 1 9 B  
#> 10 1 10 A

class(d)

#> [1] "data.frame"

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

What is LETTERS? What is L3? *answer here*

What does sample do? *answer here*

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? The columns of a data frame are actually just vectors.

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

What class of data is d$fac? *answer here*

#### 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 structures 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 containing factors you must be careful because some functions may interpret factors as their integer values! We could also avoid creating a factor in our data frame and just keep this column as characters 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. I'll suppress the output below and let you run these examples yourself.

#get the first row of d  
d[1,]  
#get the first column of d  
d[,1]  
#get the column named 'fac'  
d[,'fac']  
#or  
d[['fac']]  
#or (most efficient and readable)  
d$fac  
#get the element in the 5th row and 3rd column  
d[5,3]

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

d[,2] + 1  
d[[2]] + 1  
d[,2] \* 2  
#similarly for logical operations, note that logical 'is equivalent to' is '=='  
d[,3] == 'B'  
d$y <= 5  
  
#we can also use functions to perform complex calculations  
mean(d$y)  
median(d$y)  
sum(d$y)

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? *answer here*

What is the mean of your new column of d? Copy the code you used. *answer here*

What is the median or your new column of d? Copy the code you used. *answer here*

What is the sum of your new column? Copy the code you used. *answer here*

What fraction of the sum of your new column is each row's value? Make a new column for d showing this fraction. Copy the code you used. *answer here*

# Data analysis and visualization

## Importing and exporting data

Enough with these toy examples, let's look at some real data! Below we will analyze some real data from today's reading [Guseman et al. 2015](http://dx.doi.org/10.1242/dev.117234).

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 analyzing flow cytometry data. 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 reading, [Havens et al. 2012](http://dx.doi.org/10.1104/pp.112.202184), flow cytometry measures light scattering, absorption, and emission 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 lens 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 perpendicular, 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'.

First, 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('ggplot2', repos = 'http://cran.fhcrc.org/')  
#before using the package we must load it into our workspace  
library('ggplot2')  
  
#we will also need the ‘dplyr’ package  
install.packages('dplyr', repos = 'http://cran.fhcrc.org/')  
library('dplyr')   
  
#installing from bioConductor is a little more complicated  
source("http://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')  
#If this asks you to update packages, you don't need to. Type 'n' and press 'return'.  
  
#You need some development tools to use install\_github, which will take to long to download and install in class, so let's instead download the flowTime package from course page on catalyst and install it.  
install.packages("~/Downloads/flowTime\_0.99.1.tgz", repos = NULL, type = "source")  
# !!!You may have to modify the path depending on where you have saved flowTime\_0.99.1.tgz!!!

Make sure flowTime\_0.99.1.tgz is in the Downloads folder before running this line, you might also want to make sure that your R session is in your user home folder using getwd()

Just in case you want to use github in the future I'll include some instructions, but don't run these now.

#installing from a code repository requires the 'devtools' package from CRAN  
install.packages('devtools')  
#you will also need to install xcode from the app store  
#load this package  
library(devtools)  
#install a package from github  
devtools::install\_github(repo = 'flowTime', user = 'wrightrc')

Now that we have installed these packages we can load some flow cytometry data using the read.flowSet function. Download flowSet.zip from the catalyst webpage and unzip it into the Downloads folder.

library(flowTime)  
flowSet <- read.flowSet(path = 'flowSet', alter.names = T)  
#!!! Again make sure the path variable here is correct !!!  
# This line assumes the flowSet folder is in your working directory

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

What does alter.names = T do? *answer here*

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.

Download annotation.csv to your Dowloads folder.

annotation <- read.csv('~/Downloads/annotation.csv')

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

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

## Very basic data wrangling

In this annotation data frame strain includes IAA14 and several mutant variants of IAA14 each fused to a fluorescent protein. We can see the list of possible strains using the code below.

unique(annotation$strain)

#> [1] 14 14 G79E 14 P81A 14 P81S 14 3A   
#> Levels: 14 14 3A 14 G79E 14 P81A 14 P81S

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

What is the AFB used in this experiment? *answer here*

What are the treatments used? What information is missing from this annotation? *answer here*

In this experiment a culture of each strain was treated with 10 uM auxin and another identical culture was treated with 0 uM auxin as the control. This treatment was done at the time that well 2\_A01 was being read. We typically set the time at which the cultures were treated with hormone as T=0. We'll adjust this later.

The auxin here as you know induces the binding of the F-box protein, TIR1 along with it's associated SCF ubiquitin ligase complex to the IAA proteins. The binding of IAA proteins to SCFTIR1 allows ubiquitin to be transferred to the IAA. The ubiquitinated IAA is then degraded by the proteasome along with it's fluorescent label. Using flow cytometry, the amount of fluorescently-labelled IAA in each of the cultures was measured once every 10-15 minutes. What culture was measured in each well is recorded in the annotation.csv file. The time at which each well was read is recorded by the cytometer in the flowSet files.

Before attaching annotation 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, as these could have been ordered differently in annotation and flowSet. 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. mergeBy is the name of that unique identifier column that we created.

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

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

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

## Generating graphs

### Basic graphs

Now 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 generally short 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)

The histogram shows the number of yeast cells that have Fl1.A values within each bin of a set size across the entire range of FL1.A values. This gives us an idea of the distribution of FL1.A values.

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

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

Just by eye, what would you estimate as the mean (the average value) of the above FL1.A distribution? *answer here*

Make a histogram plot of the same strain we plotted above, but after it has been treated with 10 uM auxin (i.e. use a different well). *answer here*

Again just by eye, what is the mean of this distribution? How has the distribution changed? Is this what you would expect? *answer here*

### Plotting with ggplot2

These plots use the default graphics package in R. Below we will use the much more powerful and intuitive (once you get the hang of it) ggplot2 package. This can help us present this data in the most clear and understandable way.

Before we start exploring this data using ggplot2 let's summarize the distribution of fluorescence in each well using the summarizeFlow function. summarizeFlow will calculate the mean, median and standard deviation of the FL1.A distributions for each well. It will only include healthy single cells in this calculation based on the FSC and SSC values (this is what the ploidy and only arguments are for). These FSC and SSC values have been set by earlier experiments and are called 'gates'. We will load these gates first.

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

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

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

What data type is dat\_sum? *answer here*

What are the column names of dat\_sum? *answer here*

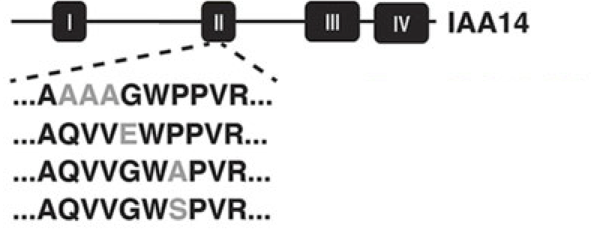
Now we can set time to zero when auxin was added, the time when well 2\_A01 was read.

#get the time when 2\_A01 was read  
T0 <- dat\_sum[which(dat\_sum$file == '2\_A01'), "time"]  
#subtract this time from the time column of dat\_sum  
dat\_sum$time <- dat\_sum$time - T0

Finally, before we graph this data let's make a hypothesis about what effect these mutations will have. Going back to our earlier code:

unique(dat\_sum$strain)

#> [1] 14 14 G79E 14 P81A 14 P81S 14 3A   
#> Levels: 14 14 3A 14 G79E 14 P81A 14 P81S

This experiment is comparing the degradation rate of IAA14, IAA14 G79E (this means that the glycine residue at position 79 has been mutated to glutamate), IAA14 P81A, IAA14 P81S, and IAA14 3A (which has three alanines replacing residues 76-78) in the presence of TIR1 and 10 uM auxin.  


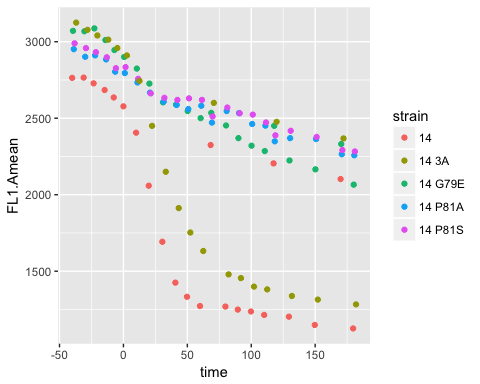
##### ???Questions???

Based on your knowledge of auxin signaling and amino acid structures hypothesize the order of their degradation rates. *answer here*

The 'gg' in ggplot2 stands for 'grammar of graphics'. This grammar allows us to map data to visual representations in a manner similar to how language maps words to abstract concepts. The grammar of graphics allows precise control over each item in a graph, and also provides convenient defaults. The qplot function uses these default values to make quick plots.

Let's explore this data a bit using the qplot function.

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



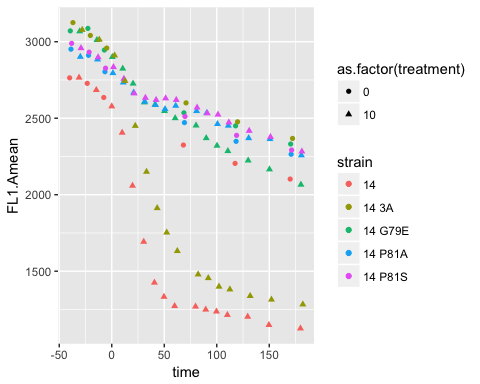
##### ???Questions???

What key piece of data is left out of the above graph? *answer here*

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? *answer here*

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



Within the grammar of graphics aesthetics are visual parameters to which you can map a variable. So for example here, x, y, color and shape are aesthetics to which we have mapped the variables time, FL1.Amean, strain and treatment. If we extend the language grammar metaphor, aesthetics are like nouns, you can have several nouns in a sentence. But we also need a verb to make a sentence, right? The verb-equivalent in the grammar of graphics is called a geometries. These geometries are the methods of representing data, e.g. bar plots, scatter plots, and box-and-whisker plots. In ggplot2 these geometries are created using which always begin with geom\_, so you can type 'geom' into the **Help** pane and see a list of all the possibilities.

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

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

Try using a different geom argument in the above line. *answer here*

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

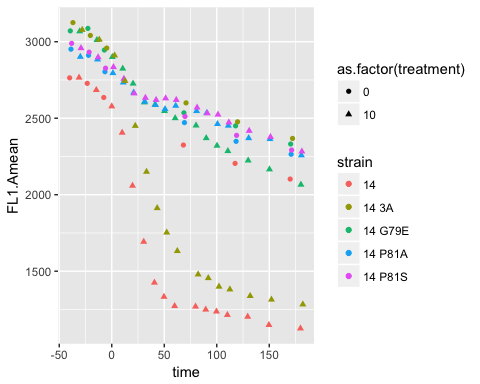
The more advanced plotting function in ggplot2 is ggplot(). ?ggplot

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

What are the arguments of ggplot? *answer here*

So ggplot sets up the aesthetic mapping, then we can add geoms to this ggplot aesthetic. Let's try to recreate the plot from above to help solidify this idea.

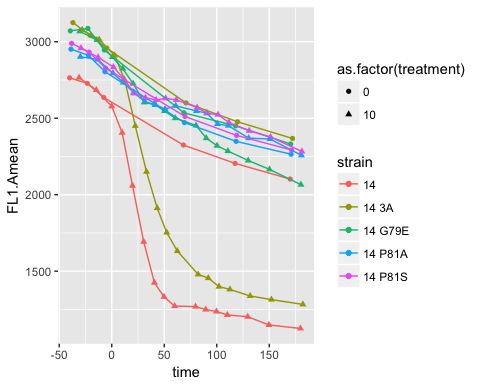
#set up the aesthetic mapping, note that we have to put this within an aes function  
plot <- ggplot(data = dat\_sum, mapping = aes(x = time, y = FL1.Amean, color = strain, shape = as.factor(treatment)))  
#now we can add to this a point geom  
plot <- plot + geom\_point()  
#and finally visualize the plot  
plot



### Making your plots easily understandable

All of the dots in the figure make it quite busy, perhaps a line connecting the dots would help show the trends. We can simply add a line geometry to the plot object, and R will add this layer to the graph.

plot <- plot + geom\_line()  
plot

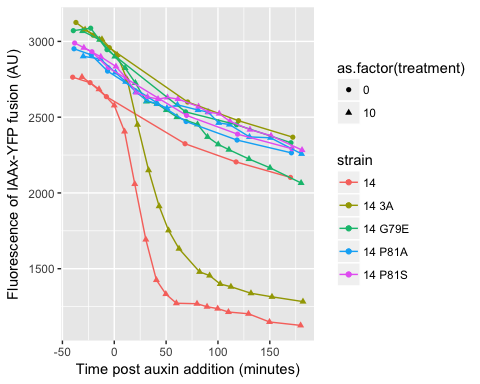
 This is easier to visualize, but our graph is still not very easy to understand for anyone who doesn't know our experimental methods.

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

What would you change about this graph or add to it, to make it more immediately understandable? *answer here*

One issue is that all of the titles are taken directly from our data frame. It would be much better to translate these variable names into plain English. Let's start with the axis titles.

plot <- plot + labs(x = 'Time post auxin addition (minutes)', y = 'Fluorescence of IAAx-YFP fusion (AU)')  
plot



Similarly, we can change the titles of the legends using the aesthetics we have mapped the variables to as parameters to labels.

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

Change the legend titles such that they provide full information about the variables they are showing (don't forget units). Copy the code and the final graph below.  
*answer here*

What would you title this graph? Try to fully describe the experiment as concisely as possible.  
*answer here*

Use ggtitle() to add a main title to the graph. Copy the code and graph below.  
*answer here*

Wow, that looks great! But there is still one last thing. Remember from the **Factors** section how we could add a label to each level of our factors? Let's use this to make the factor dat\_sum$strain more descriptive.

levels(dat\_sum$strain)

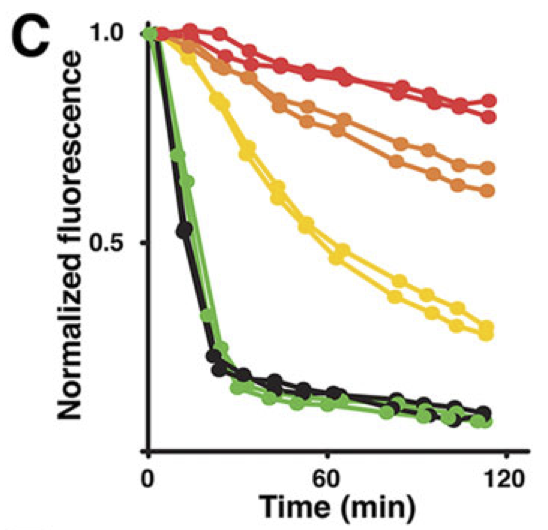
#> [1] "14" "14 3A" "14 G79E" "14 P81A" "14 P81S"

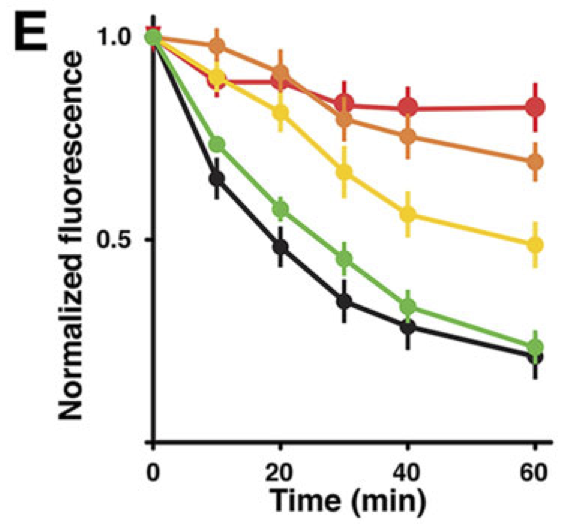
dat\_sum$strain <- factor(dat\_sum$strain, levels = levels(dat\_sum$strain), labels = c('IAA14', 'IAA14-3A', "IAA14-G79E","IAA14-P81A","IAA14-P81S"))  
#note that the labels vector must be in the same order as the levels vector

Now that we have changed the data we need to reassemble the plot from scratch.

plot <- ggplot(data = dat\_sum, mapping = aes(x = time, y = FL1.Amean, color = strain, shape = as.factor(treatment))) + geom\_point() + geom\_line() + labs(x = 'Time (minutes)', y = 'Fluorescence of IAAx-YFP fusion (AU)') #add your legend labels in here

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

What is the difference between our graph and figure 1C from Guseman et al.?  
*answer here*

What is the difference between figures 1C and 1E from Guseman et al.?  
*answer here*

What do the authors conclude based on these two figures?  
*answer here*