

R for Beginners

C. Tobin Magle

February 8, 2016

This material was adapted from [Data Carpentry](#) and [Software Carpentry](#) lessons. This document and associated script can be found at <https://github.com/maglet/r-for-beginners>

Learning Objectives

After completing this tutorial, you will be able to

- load tabular data into R
- calculate summary statistics for these data
- create a publication-quality graph

Software

This tutorial requires R and R studio to be installed. If you don't have these installed, please use [this tutorial](#).

Introducing R Studio

R studio makes programming in R easier. One of the hardest things about any programming language is remembering the syntax. Missing one semicolon or misspelling a variable or function name can cause your code to fail. Luckily, you can use scripts to save code that you have written, with notes to indicate what the code does.

Open a script file

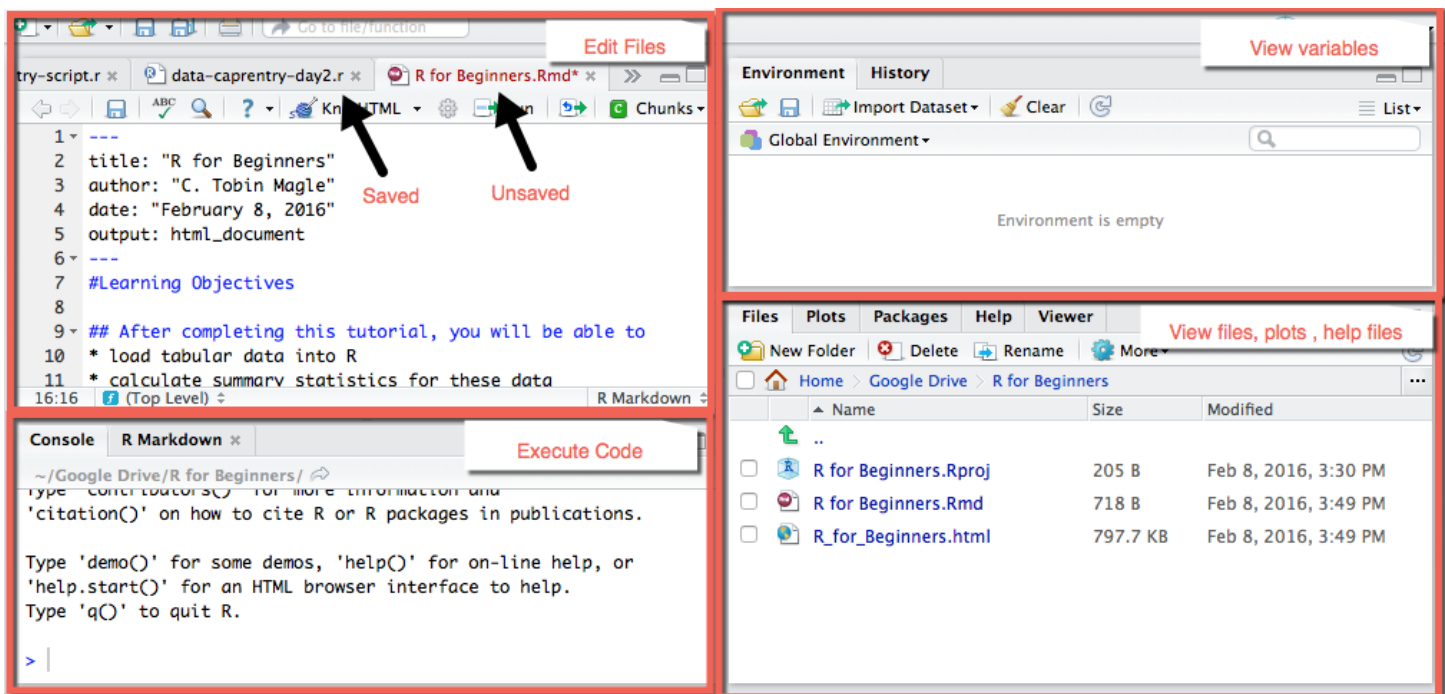
Scripts let you save your code to run whenever you want. To open a new script:

- Go to the **File** menu
- Select **New File**
- Select **R Script**

Now save the script to where you want to work. Every time you make changes to the script, the name of the script turns red, and a star appears after the name. Go to **File>Save** to save the file. You can also use the appropriate short key (**ctrl-s** for PC, **command-s** for MacOS).

Remember: Save early, save often!

See what the different windows represent below.



Upper left: Editor- where you write your scripts

Upper right: Environment- view the variables defined in your environment

Lower left: Console- where the code is executed

Lower Right: Help: where to view help files, plots, and loaded files

Set your working directory

Because we're going to be downloading and importing files, it's important to set your working directory so R knows where to look for the files. To set your working directory:

- Go to the **Session** menu
- Select **Set Working Directory**
- Select **Source File Location**

Now R will look where the script is located for your data files. If your script and your files are in a different location for some reason, you can choose a different working directory by selecting **Choose Directory**, but for the purposes of this tutorial, let's just keep it in the script location.

Operators

Operators are used to perform mathematical and logical operations. But more simply, they allow you to assign values to variables and combine/compare their values.

Operators can be broken into 4 categories: **assignment**, **logical**, **arithmetic**, **relational**.

You can find a more thorough discussion of these operators [here](#), but for now we'll focus on assignment operators.

Assignment operator

One of the most basic things to do in programming is to assign a **value** to a **variable**. Let's say you want to assign the value 4 to the variable x. To do this in R, type the following into your script file:

```
x<-4 #assigns the value 4 to the variable x
```

Now, check your work. When you type in 'x'

```
x #print x to the output screen
```

```
## [1] 4
```

R outputs the number 4.

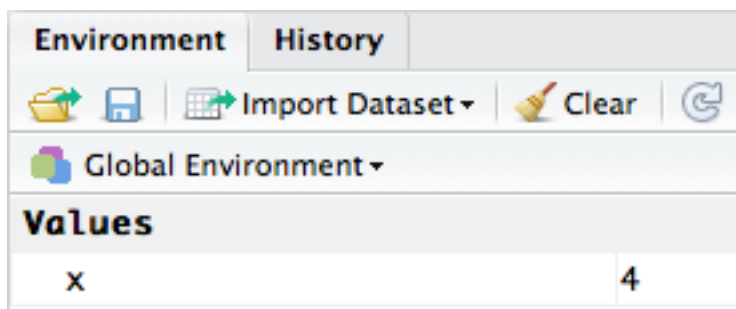
How do we send this code to the **Console** (lower left window) to actually run the code? There's a short key for that!

- Put your cursor on the line of code you'd like to run
- Use ctrl-Enter to send that line of code to the console
- The cursor then jumps to the next line of code
- Repeat as necessary
- you can also highlight a block of code, and use ctrl-Enter to send it to the console at once.

Your **Console** should look like this now (lower left window)

```
> x<-4 #assigns the value 4 to the variable x
> x #print x to the output screen
[1] 4
> |
```

Also, you should see the variable x in your **Global Environment** (upper right window)



Why not use =?

The equal sign (=) works the same as the assignment operator in most cases.

It also serves a separate purpose in R. Instead of assigning values to variables, it's used to define arguments to functions. We'll show you what this means as we start to use functions later in this tutorial.

So, it's a matter of preference. This tutorial will use <- for clarity.

Using comments.

As you write your code, it's a great idea to document it in human-readable language. Any text after the '#' symbol will not be executed. Let's try it.

```
y<-"This is not in a comment" #This text will be assigned to the variable y  
#y<-"This is in a comment"      #This text will not  
y                               #prints y to the console
```

```
## [1] "This is not in a comment"
```

The system only prints out the value that was assigned in the un commented code. Use comments to take notes about what the code does.

Functions

Functions are discrete units of R code that allow you to perform specific tasks. You can find functions:

- **In the R base package:** no extra installation required.
- **In external packages:** because R is open source, many people create their own functions for specific purposes and make them available in the CRAN repository.
- **Make your own!:** you can write your own functions.

We'll be using functions from the base package in this tutorial.

Now we'll go over some really useful functions to use in R.

Loading files

You can use the assignment operator to load tabular data into R.

Let's use a data table called **inflammation.csv** that contains daily antibody levels for 30 days after the administration of a vaccine. The table also contains data about the gender of the subjects.

Downloading files You can download the file at this address: [here](https://raw.githubusercontent.com/maglet/r-for-beginners/master/inflammation.csv)

But why not use R to do the downloading for you? That's what the **download.file** function is for.

Here's the code to download a file in R.

```
download.file(url="https://raw.githubusercontent.com/maglet/r-for-beginners/master/inflammation.csv",  
             destfile = "inflammation.csv")
```

See what I mean about '=' being used for something different in R? It's used to give the function's **arguments** values. Arguments are information that the function needs to run. In this case, we need to know where the file is online and where you want to store it locally.

- The **url** argument specifies where the file is on the web

- The **destfile** argument specifies where you want the downloaded data to be stored. If you don't include a file path, the function saves the file in the working directory.
- Make sure both the url and the destination file are in quotes.

If you're not sure what arguments a function can take, type "help(FunctionName)". This command will open the help file for the function in the lower right of the R Studio console.

For example, this is how you'd pull up the help file for **download.file**

```
help(download.file) # get R documentation for download.file function
```



The screenshot shows the R Studio interface with the Help tab selected. The title bar reads "R: Download File from the Internet". Below the title bar is a search bar with the text "Find in Topic". The main content area displays the help page for `download.file` from the `utils` package. The page title is "Download File from the Internet". Under the "Description" section, it states: "This function can be used to download a file from the Internet." Under the "Usage" section, the function signature is shown: `download.file(url, destfile, method, quiet = FALSE, mode = "w", cacheOK = TRUE, extra = getOption("download.file.extra"))`. Under the "Arguments" section, the parameters are listed: `url` (A character string naming the URL of a resource to be downloaded.), `destfile` (A character string with the name where the downloaded file is saved. Tilde-expansion is performed.), `method` (Method to be used for downloading files. Current download methods are "internal", "wininet" (Windows only), "libcurl", "wget" and "curl", and there is a value "auto": see 'Details' and 'Note'. The method can also be set through the option "download.file.method": see `options()`).



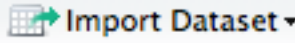



It takes a while to learn how to effectively read the R help files, but don't worry. We will explain all of the functions and arguments you need in this tutorial. You shouldn't have to rely much on the help files.

Loading data into R Now that you've downloaded the data you, you can load it into R. We're going to use the assignment operator and the **read.csv** function.

Below is the syntax to read the **inflammation.csv** file into R.

```
inflammation<-read.csv("inflammation.csv") #reads data from the csv file
```

Now your global environment looks something like this:

Environment		History
    Clear 		
Global Environment ▾		
Data		
 inflammation	720 obs. of 32 variables	
Values		
x	4	

There's a new dataset called **inflammation**. Descriptive variable names are good.

Inspecting your data

What you can do with the data set in `r` depends on the content and type of data in **inflammation**.

To find out the format that R is storing the data in, you can use the function **class**. Every variable in R has a class, including the variables **x** and **y** from earlier. Let's see what class these two items are.

```
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "character"
```

```
class(inflammation)
```

```
## [1] "data.frame"
```

x is numeric; **y** is character, as expected.

Inflammation is a **data frame**. This is a special R format that makes dealing with tabular data easier.

- **Each column is a variable.** In this case, it's patient id, gender, and 30 days of antibody levels.
- **Each row is an observation.** In this case, it's all of the patients in the study.
- **Each column must contain the same data type** (for example, all numeric)
- **The data set must be “rectangular”**, which means all variables (columns) must have the same number of observations (elements).

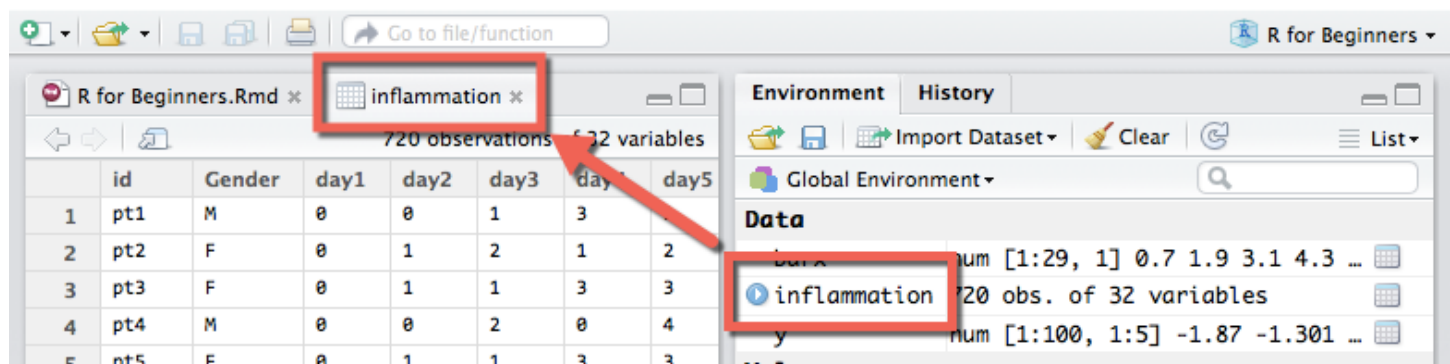
To see how the data are formatted in a data frame, you can use the **head** command. It takes one argument (the data frame).

```
head(inflammation) #displays the first 6 elements of all columns
```

```
##   id Gender day1 day2 day3 day4 day5 day6 day7 day8 day9 day10 day11
## 1 pt1     M    0    0    1    3    1    2    4    7    8    3    3
## 2 pt2     F    0    1    2    1    2    1    3    2    2    6   10
## 3 pt3     F    0    1    1    3    3    2    6    2    5    9    5
## 4 pt4     M    0    0    2    0    4    2    2    1    6    7   10
## 5 pt5     F    0    1    1    3    3    1    3    5    2    4    4
## 6 pt6     F    0    0    1    2    2    4    2    1    6    4    7
##   day12 day13 day14 day15 day16 day17 day18 day19 day20 day21 day22 day23
## 1     3     10     5     7     4     7     7    12    18     6    13    11
## 2    11     5     9     4     4     7    16     8     6    18     4    12
## 3     7     4     5     4    15     5    11     9    10    19    14    12
## 4     7     9    13     8     8    15    10    10     7    17     4     4
## 5     7     6     5     3    10     8    10     6    17     9    14     9
## 6     6     6     9     9    15     4    16    18    12    12     5    18
##   day24 day25 day26 day27 day28 day29 day30
## 1    11     7     7     4     6     8     8
## 2     5    12     7    11     5    11     3
## 3    17     7    12    11     7     4     2
## 4     7     6    15     6     4     9    11
## 5     7    13     9    12     6     7     7
## 6     9     5     3    10     3    12     7
```

By default, **head** will display the first 6 values of each column in the data frame.

If you want to see the entire data frame in a new tab, click on the name of the variable in the **Environment** window. A spreadsheet will open as a new tab in the editor (upper left).



To find out what data type is contained in each column, use the **str** command (which stands for structure) to get more information.

```
str(inflammation)
```

```
## 'data.frame':   720 obs. of  32 variables:
## $ id      : Factor w/ 720 levels "pt1","pt10","pt100",...: 1 112 223 334 445 556 667 699 710
## $ Gender  : Factor w/ 2 levels "F","M": 2 1 1 2 1 1 1 2 1 ...
## $ day1    : int  0 0 0 0 0 0 0 0 0 0 ...
## $ day2    : int  0 1 1 0 1 0 0 0 0 1 ...
```

```
## $ day3 : int 1 2 1 2 1 1 2 1 0 1 ...
## $ day4 : int 3 1 3 0 3 2 2 2 3 2 ...
## $ day5 : int 1 2 3 4 3 2 4 3 1 1 ...
## $ day6 : int 2 1 2 2 1 4 2 1 5 3 ...
## $ day7 : int 4 3 6 2 3 2 2 2 6 5 ...
## $ day8 : int 7 2 2 1 5 1 5 3 5 3 ...
## $ day9 : int 8 2 5 6 2 6 5 5 5 5 ...
## $ day10 : int 3 6 9 7 4 4 8 3 8 8 ...
## $ day11 : int 3 10 5 10 4 7 6 7 2 6 ...
## $ day12 : int 3 11 7 7 7 6 5 8 4 8 ...
## $ day13 : int 10 5 4 9 6 6 11 8 11 12 ...
## $ day14 : int 5 9 5 13 5 9 9 5 12 5 ...
## $ day15 : int 7 4 4 8 3 9 4 10 10 13 ...
## $ day16 : int 4 4 15 8 10 15 13 9 11 6 ...
## $ day17 : int 7 7 5 15 8 4 5 15 9 13 ...
## $ day18 : int 7 16 11 10 10 16 12 11 10 8 ...
## $ day19 : int 12 8 9 10 6 18 10 18 17 16 ...
## $ day20 : int 18 6 10 7 17 12 6 19 11 8 ...
## $ day21 : int 6 18 19 17 9 12 9 20 6 18 ...
## $ day22 : int 13 4 14 4 14 5 17 8 16 15 ...
## $ day23 : int 11 12 12 4 9 18 15 5 12 16 ...
## $ day24 : int 11 5 17 7 7 9 8 13 6 14 ...
## $ day25 : int 7 12 7 6 13 5 9 15 8 12 ...
## $ day26 : int 7 7 12 15 9 3 3 10 14 7 ...
## $ day27 : int 4 11 11 6 12 10 13 6 6 3 ...
## $ day28 : int 6 5 7 4 6 3 7 10 13 8 ...
## $ day29 : int 8 11 4 9 7 12 8 6 10 9 ...
## $ day30 : int 8 3 2 11 7 7 2 7 11 11 ...
```

By default, **str** displays the dimensions of the data frame (720 observations x 32 variables) and the name of each variable (ID, Gender, day1...) followed by the type of data (**Factor** or **integer**) and the first 10 elements in the column.

In this dataset, each day column hold a list of integers that represent antibody levels.

But what does it mean to say that **id** and **Gender** are factors? **Factors** are categorical data that describe the observation.

- **id** is a factor with 720 levels, the same as the number of observations. This is good because **id** is meant to be a unique identifier for each patient. This is not a very useful factor though, because you can't group the data by this variable.
- **Gender** is a factor with 2 levels: "M" or "F". You can use this factor to split the data into male and female subgroups. We'll come back to this point later.

Finally, to get the min, max, median, mean, and quartiles for the numerica data, and the counts for the factor variables, you can use the **summary** command.

```
summary(inflammation)
```



```

##          id      Gender      day1      day2      day3
## pt1      : 1    F:365    Min.      :0    Min.      :0.000    Min.      :0.0000
## pt10     : 1    M:355    1st Qu.:0    1st Qu.:0.000    1st Qu.:0.0000
## pt100    : 1                Median :0    Median :0.000    Median :1.0000
## pt101    : 1                Mean   :0    Mean   :0.375    Mean   :0.8625
## pt102    : 1                3rd Qu.:0    3rd Qu.:1.000    3rd Qu.:1.0000
## pt103    : 1                Max.    :0    Max.    :1.000    Max.    :2.0000
## (Other):714
##          day4      day5      day6      day7
## Min.      :0.000    Min.      :0.000    Min.      :0.00    Min.      :0.000
## 1st Qu.:1.000    1st Qu.:1.000    1st Qu.:2.00    1st Qu.:2.000
## Median :2.000    Median :2.000    Median :3.00    Median :4.000
## Mean   :1.578    Mean   :2.032    Mean   :2.81    Mean   :3.522
## 3rd Qu.:3.000    3rd Qu.:3.000    3rd Qu.:4.00    3rd Qu.:5.000
## Max.    :3.000    Max.    :4.000    Max.    :5.00    Max.    :7.000
##
##          day8      day9      day10     day11
## Min.      :0.000    Min.      :0.000    Min.      : 0.000    Min.      : 0.000
## 1st Qu.:2.000    1st Qu.:3.000    1st Qu.: 4.000    1st Qu.: 3.000
## Median :4.000    Median :5.000    Median : 6.000    Median : 5.000
## Mean   :3.904    Mean   :4.467    Mean   : 5.528    Mean   : 5.382
## 3rd Qu.:5.000    3rd Qu.:6.000    3rd Qu.: 8.000    3rd Qu.: 8.000
## Max.    :7.000    Max.    :8.000    Max.    :12.000    Max.    :10.000
##
##          day12     day13     day14     day15
## Min.      : 0.000    Min.      : 0.000    Min.      : 0.000    Min.      : 0.000
## 1st Qu.: 4.000    1st Qu.: 4.000    1st Qu.: 4.000    1st Qu.: 5.000
## Median : 7.000    Median : 7.000    Median : 6.000    Median : 9.000
## Mean   : 6.758    Mean   : 6.782    Mean   : 6.746    Mean   : 8.525
## 3rd Qu.: 9.000    3rd Qu.: 9.000    3rd Qu.:10.000    3rd Qu.:12.000
## Max.    :14.000    Max.    :12.000    Max.    :13.000    Max.    :17.000
##
##          day16     day17     day18     day19
## Min.      : 0.000    Min.      : 0.000    Min.      : 0.00    Min.      : 0.00
## 1st Qu.: 5.000    1st Qu.: 5.000    1st Qu.: 6.00    1st Qu.: 7.00
## Median : 9.000    Median : 8.000    Median :10.00    Median :11.00
## Mean   : 8.817    Mean   : 8.518    Mean   :10.11    Mean   :10.51
## 3rd Qu.:13.000    3rd Qu.:12.000    3rd Qu.:14.00    3rd Qu.:15.00
## Max.    :17.000    Max.    :16.000    Max.    :20.00    Max.    :18.00
##
##          day20     day21     day22     day23
## Min.      : 0.00    Min.      : 0.00    Min.      : 0.00    Min.      : 0.000
## 1st Qu.: 6.00    1st Qu.: 7.00    1st Qu.: 6.00    1st Qu.: 5.000
## Median :11.00    Median :10.00    Median :11.00    Median : 9.000
## Mean   :10.53    Mean   :10.95    Mean   :10.48    Mean   : 9.314
## 3rd Qu.:15.00    3rd Qu.:15.00    3rd Qu.:15.00    3rd Qu.:13.000
## Max.    :19.00    Max.    :20.00    Max.    :19.00    Max.    :18.000
##
##          day24     day25     day26     day27
## Min.      : 0.000    Min.      : 0.000    Min.      : 0.000    Min.      : 0.000

```

```
## 1st Qu.: 6.000 1st Qu.: 6.000 1st Qu.: 5.000 1st Qu.: 5.000
## Median :10.000 Median :10.000 Median : 7.000 Median : 8.000
## Mean : 9.983 Mean : 9.628 Mean : 7.996 Mean : 8.219
## 3rd Qu.:14.000 3rd Qu.:13.000 3rd Qu.:12.000 3rd Qu.:11.000
## Max. :19.000 Max. :16.000 Max. :15.000 Max. :16.000
##
## day28 day29 day30
## Min. : 0.000 Min. : 0.000 Min. : 0.000
## 1st Qu.: 4.000 1st Qu.: 5.000 1st Qu.: 3.000
## Median : 7.000 Median : 8.000 Median : 5.000
## Mean : 7.018 Mean : 7.542 Mean : 5.618
## 3rd Qu.:10.000 3rd Qu.:10.000 3rd Qu.: 8.000
## Max. :13.000 Max. :14.000 Max. :11.000
##
```

Subsetting Especially with large data sets, sometimes you need to assess a particular part of the data. Depending on what you want to do, you'll either want to work with a subset of rows or columns. We'll look at how to subset by both.

Subsetting Columns Let's say that you're only interested in the inflammation on day 30. Use the following code to extract only the day 30 data and store it in a variable called **day30inflammation**

```
day30inflammation<-inflammation$day30
```

- You already know about the assignment operator (<-)
- The \$ operator lets you subset the data fram based on column name
- **Syntax:** (variable name)\$ (column name)

Now you should see a new variable called **30dayinflammation** with 720 elements in your global environment.

You can get the same subset using the “[row,column]” format:

```
day30inflammation2<-inflammation[,32]
```

- **Syntax:** the row you want and the column you want, in brackets, separated by a comma
- Leaving either field blank, will select ALL elements, in this case all rows
- the day 30 data are stored in column 32 of the data frame.

Prove to yourself that these data sets are equal using a **Boolean operator** “==”. Make sure to use the double equal sign when you're evaluating whether 2 values are equivalent. A single “=” will try to perform an assignment operation.

Running **day30inflammation == day30inflammation2** would give you so we'll

```
day30inflammation == day30inflammation2
```



```
## [701] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [715] TRUE TRUE TRUE TRUE TRUE TRUE
```

The above code returns a list of TRUE/FALSE values that indicate whether the values in each position of the list match. Scanning through that list could get tedious, so instead we can use the fact that **TRUE** in R is equivalent to **1**, and **FALSE** is the equivalent of **0**. Thus, if all elements are equal to each other, the sum of that list of **TRUE**s and **FALSE**s would be equal to the number of rows.

```
nrow(inflammation)==sum(day30inflammation == day30inflammation2)
```

```
## [1] TRUE
```

It checks out! And now you only have to look at one result.

Now let's calculate the mean of this column:

```
mean(day30inflammation)
```

```
## [1] 5.618056
```

But what if you want to find the mean for all of the columns? That's where the **apply** function comes in. This function applies a function (**FUN**) to a chunk of data (**X**) on either the rows or columns (**MARGIN**).

Here's how you get the mean and standard deviation for all of the day columns:

```
daymeans<-apply(X=inflammation[,3:32], #the data you want to use
                MARGIN=2,               #apply the function to the columns
                FUN=mean)               #take the mean of the values

daysd<-apply(X=inflammation[,3:32],   #the data you want to use
              MARGIN=2,                 #apply the function to the columns
              FUN=sd)                  #take the standard deviation of the columns
```

- **X=inflammation[,3:32]**: grabs the 3rd to 32nd columns of the data frame
- **MARGIN = 2**: tells apply to run the function on each column (MARGIN=1 would indicate rows)
- **FUN= mean**: tells apply to run the mean function

You can get the mean for each patient as well by changing the **margin**.

```
ptmeans<-apply(X=inflammation[,3:32], #the data you want to use
               MARGIN=1,               #apply the function to rows
               FUN=mean)               #take the mean of the rows
```

Subsetting Rows. But what if you only want to see values for a certain subset of patients? That's where the **subset** function comes in. Suppose you want to get inflammation values only for female subjects and store that subset in a variable called **inflammationF**.

```
inflammationF <- subset(x= inflammation,      #the data frame to subset
                        subset = (Gender == "F")) #the factor to subset on
```

- **x = inflammation:** tells subset the data set we want to use
- **subset = (Gender == "F"):** a boolean expression that tells subset to only select observations (rows) with the value "F" in the Gender variable (column).

Let's do the same thing for male subjects

```
inflammationM <- subset(x= inflammation,      #the data frame to subset
                        subset = (Gender == "M")) #the factor to subset on
```

Now we can compare the data for the two groups using a t-test:

```
t.test(inflammationF$day30, inflammationM$day30)

##
##  Welch Two Sample t-test
##
## data:  inflammationF$day30 and inflammationM$day30
## t = -0.38316, df = 717.33, p-value = 0.7017
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.5305091  0.3572504
## sample estimates:
## mean of x mean of y
##  5.575342  5.661972
```

That's a pretty high p value.

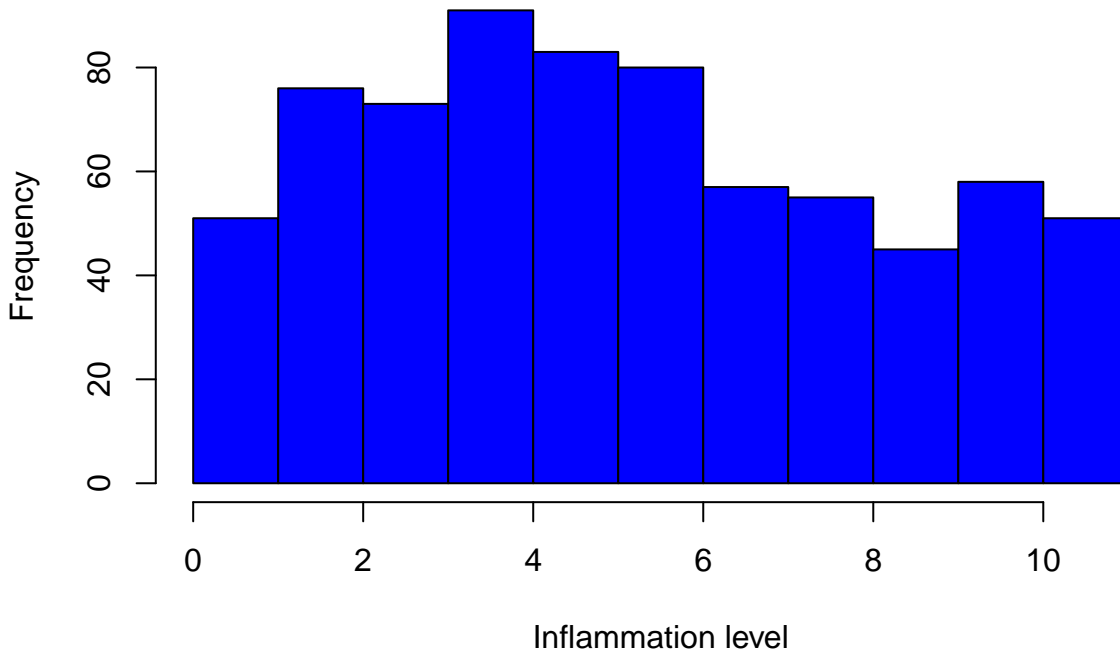
Plotting data Lastly, we're going to look at how to plot these data. The R base graphics package has many options for plotting your data. Many people prefer more advanced packages, such as [ggplot2](#), but we will stick to the base package here for simplicity's sake.

We'll be working with **histograms**, **bar plots**, **box plots**, **scatter plots**, and a **basic plot with colorization by factors**.

First, let's look at the distribution of values of day 30 inflammation, and just for kicks, we'll make the bars blue. Make sure to label the axes!

```
hist(x=inflammation$day30,      #plots inflammation at day 30 for all patients
     main="Histogram of Inflammation level on day 30", #main title
     xlab="Inflammation level",      #x axis label
     ylab="Frequency",      #y axis label
     col="blue")      #color
```

Histogram of Inflammation level on day 30

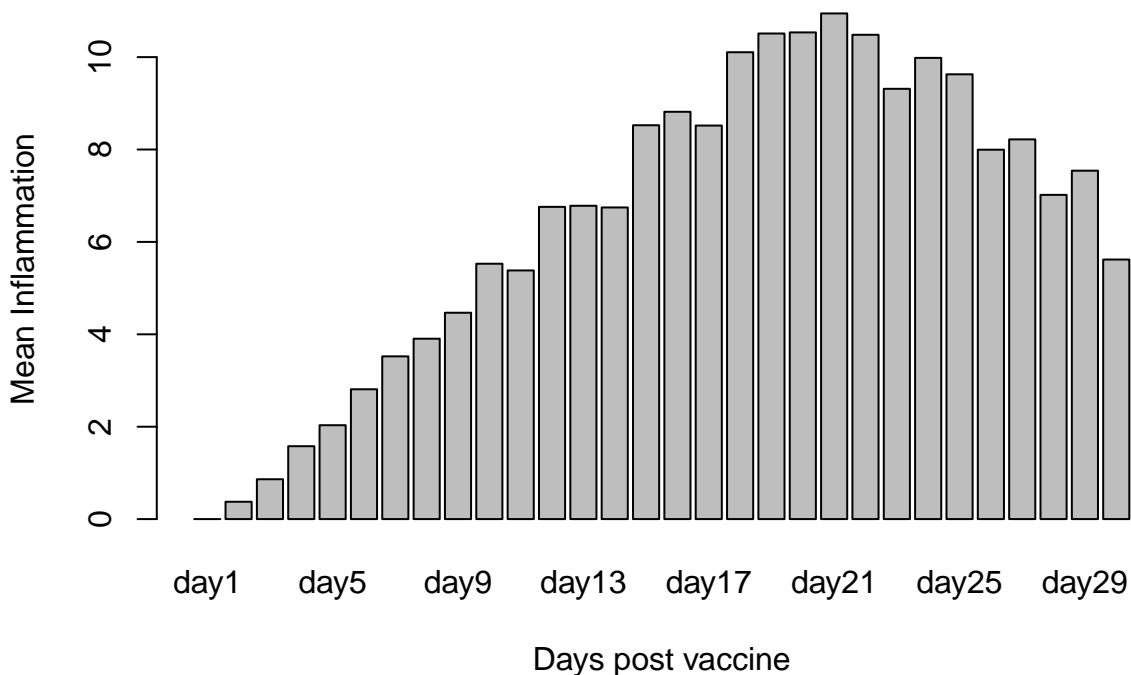


This allows you to see how your data are distributed. Remember how hard it is to [make a histogram in Excel](#)? That's a thing of the past for you now.

Now let's look at making a bar chart of the averages for each day using our variable `daymeans`.

```
barplot(height=daymeans,                                #height of the bars
        main="Mean inflammation over time",              #main graph title
        xlab="Days post vaccine",                        #x axis label
        ylab="Mean Inflammation")                       #y axis label
```

Mean inflammation over time



But wait, where are the error bars? This requires some extra code.

A great way to find code is to look at R blogs. I found some code to make box plots [here](#) and adapted it to our purposes.

Here's a function to plot the error bars. I didn't need to change anything here, because we'll provide our arguments during the function call.

```
error.bar <- function(x, # the bar plot
                     y, #daily means
                     upper, #standard deviation
                     lower=upper, #error bars in both directions
                     length=0.1,...) #length of the arrow head (in inches)
{
  if(length(x) != length(y) | length(y) !=length(lower) | length(lower) != length(upper))
    stop("vectors must be same length")
  arrows(x0=x,          #point to draw the arrow from (x)
         y0=y+upper,    #                               (y)
         x1=x,          #                               to (x)
         y1=y-lower,    #                               to (y)
         angle=90,      #draw it at a 90 degree angle
         code=3,        #choose type of arrow
         length=length, #length of the arrow head (inches)
         ...)          #sends graphics parameters from the function call
}
```

Now there is a function called **error.bar** in your environment.

Functions

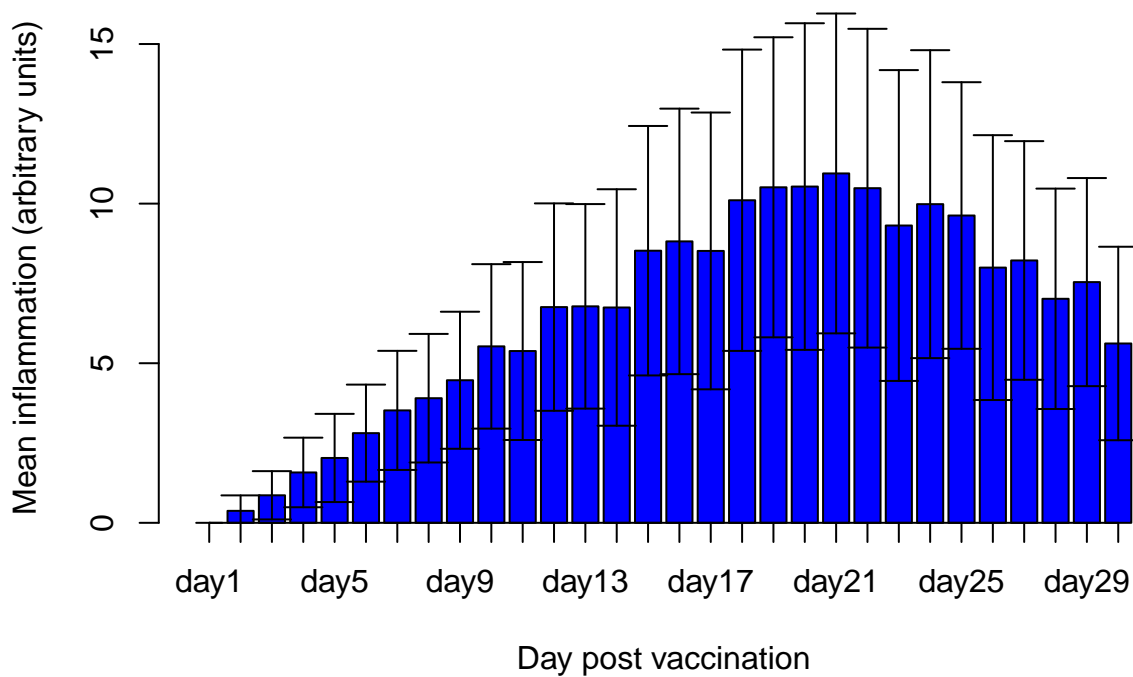
error.bar	function (x, y, upper, lower = upper, length = 0.1, ...)
-----------	--

Now I replaced their randomly generated variables with the ones that we want to plot in the function call.

```
#This command draws the plot w/o error bars
barx <- barplot(height=daymeans,      #height of the bars
               ylim=c(0,16),         #y axis range, determined by tweaking
               col="blue",           #blue bars
               axis.lty=1,           #choose line type (opaque) for y axis
               xlab="Day post vaccination", #x axis label
               ylab="Mean inflammation (arbitrary units)") #y axis label

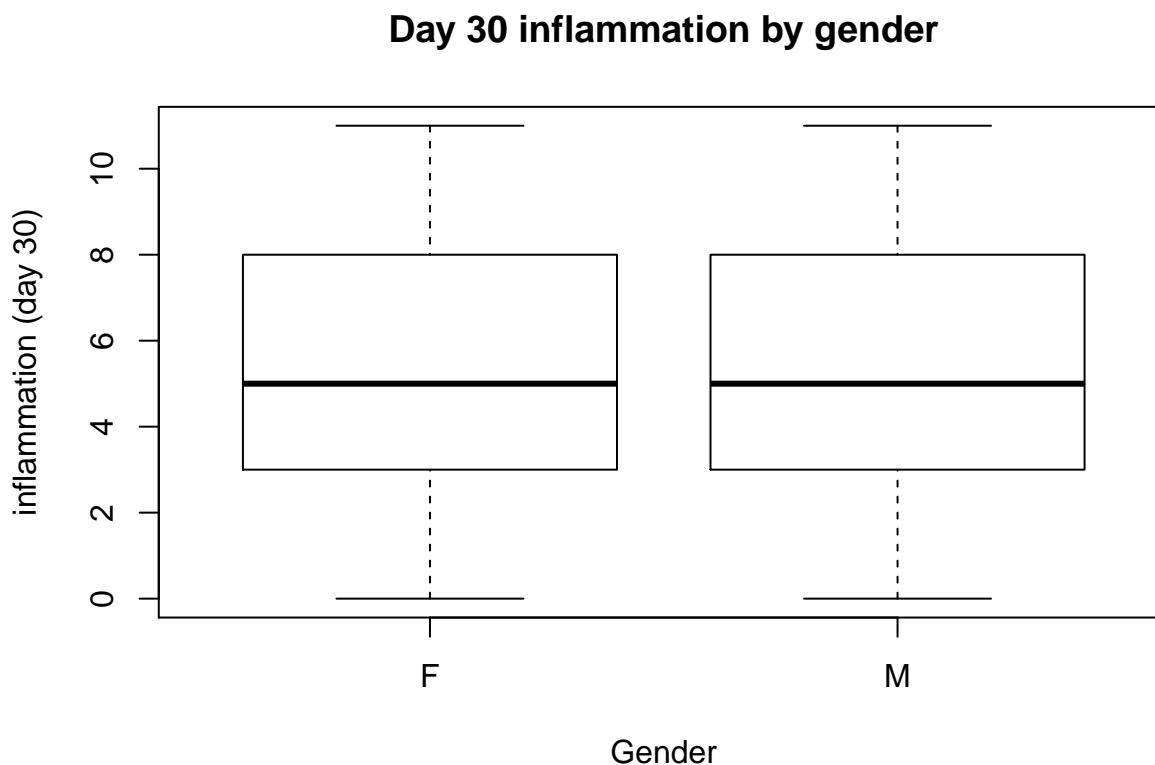
#this function draws the error bars
error.bar(barx, daymeans, daysd)
```

```
## Warning in arrows(x0 = x, y0 = y + upper, x1 = x, y1 = y - lower, angle =
## 90, : zero-length arrow is of indeterminate angle and so skipped
```



(The error is appearing because the error bar for day 1 is length zero, because all values on day 1 are 0.)
 Now let's draw a box plot, separating the data based on gender.

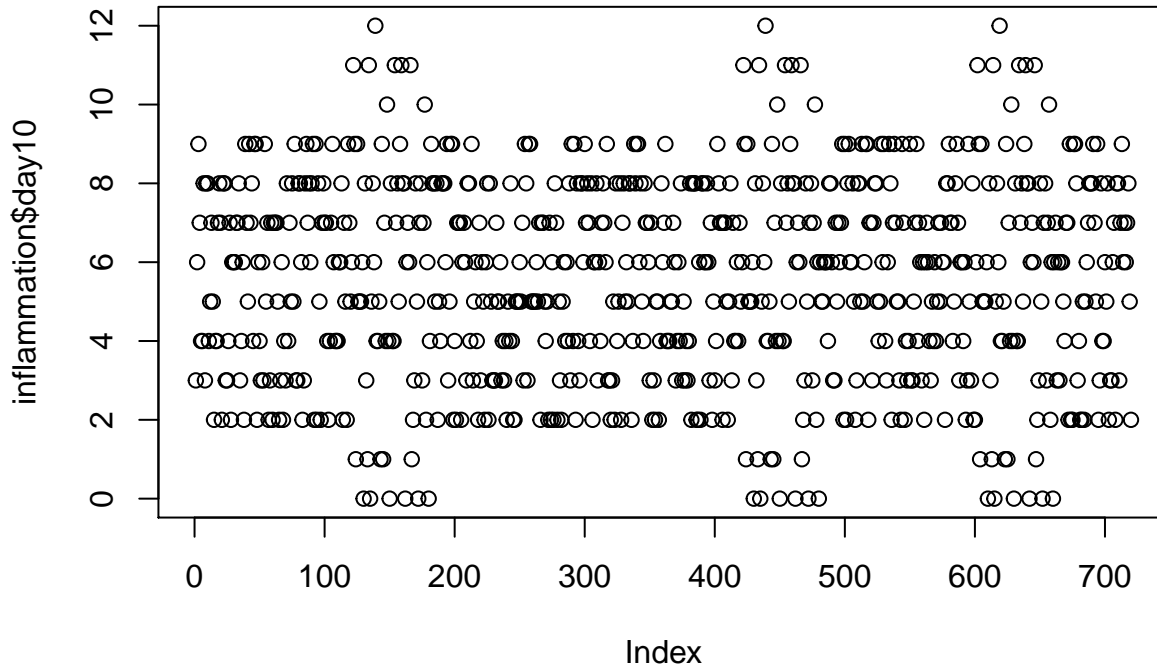
```
#Data to plot      #factor to separate on
boxplot(inflammation$day30 ~ inflammation$Gender,
        main="Day 30 inflammation by gender",
        ylab="inflammation (day 30)",
        xlab="Gender")
```



Not much of a gender difference, which was expected because of the T-test results.

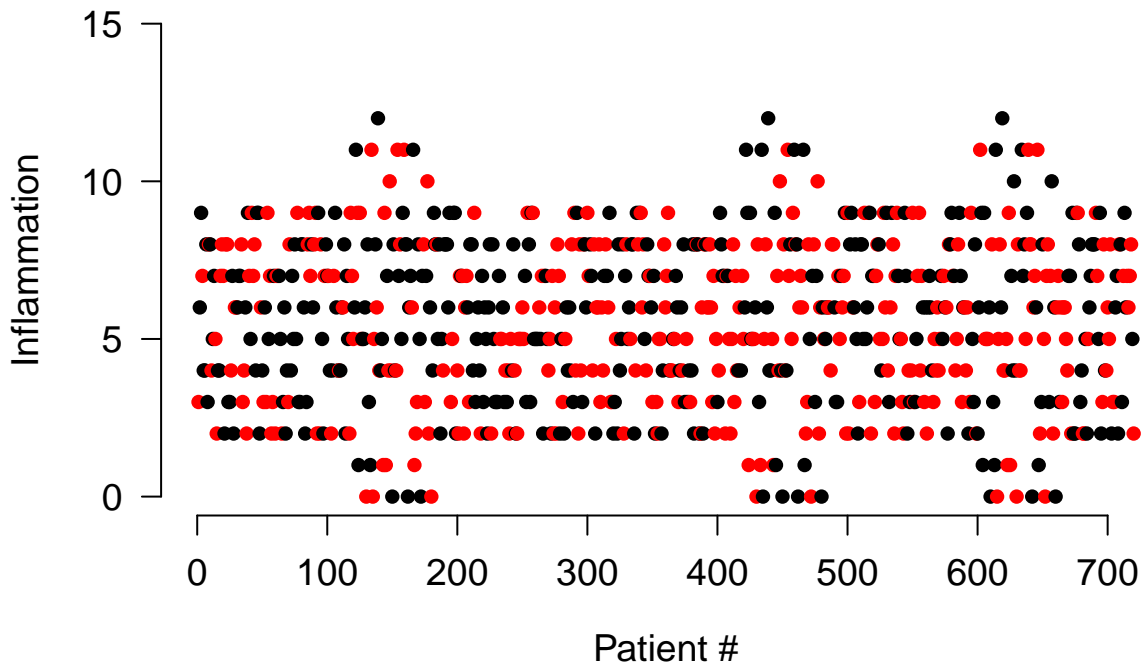
We can also use colors to differentiate data points based on gender. First, print out the inflammation on day 10 for all patients:


```
plot(inflammation$day10)
```



Let's see which of these data points are from females and which are from males. I also altered some other useful base graphics packages to get the image ready for publication:

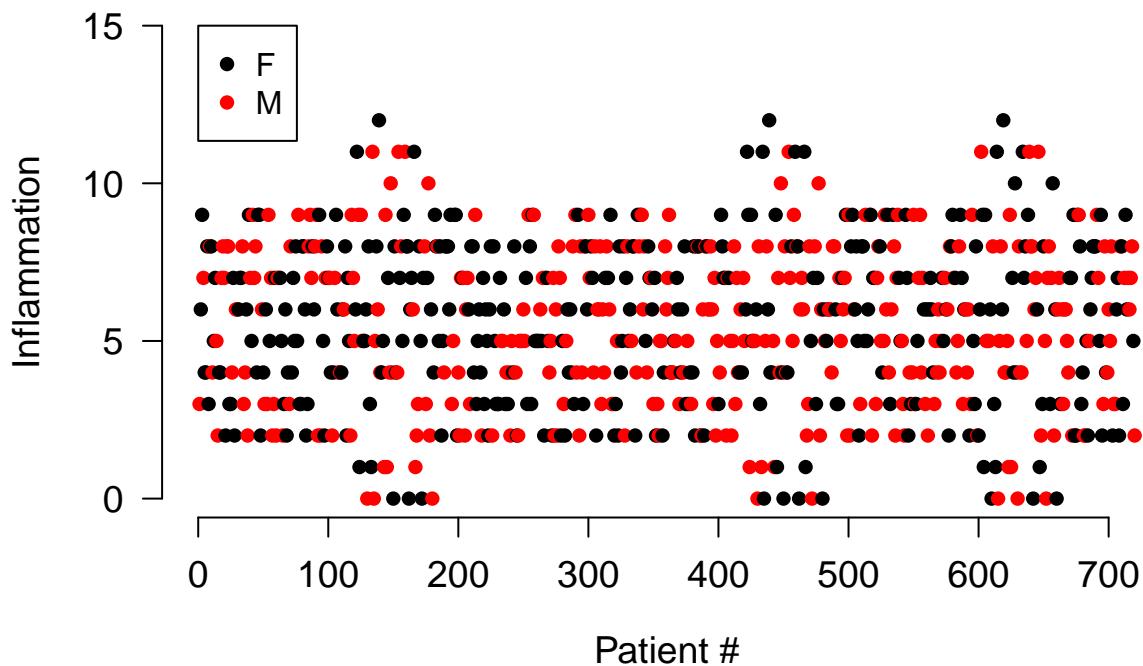
```
plot(inflammation$day10,
     xlab="Patient #",           #X axis label
     ylab="Inflammation",       #Y axis label
     las= 1,                     #Make numbers horizontal on the x axis,
     cex.lab=1.2,                #Increase axis labels font size
     cex.axis=1.2,              #Increase axis label number size
     lwd=2,                      #Increase line/dot thickness
     bty = "n",                 #removes black border around the plot
     pch=16,                    #Specifies symbols to draw
     ylim= c(0,15),             #set y axis limits
     col = inflammation$Gender) #Set Color based on Gender
```



But how do we know which are male and which are female? A figure legend would help:

```
plot(inflammation$day10,
     xlab="Patient #",          #X axis label
     ylab="Inflammation",      #Y axis label
     las= 1,                   #Make numbers horizontal on the x axis,
     cex.lab=1.2,              #Increase axis labels font size
     cex.axis=1.2,            #Increase axis label number size
     lwd=2,                    #Increase line/dot thickness
     bty = "n",                #removes black border around the plot
     pch=16,                   #Specifies symbols to draw
     ylim= c(0,15),            #set y axis limits
     col = inflammation$Gender) #Set Color based on Gender

legend(x= 0,                   #x position of legend
       y=15,                   #y position of legend
       legend=levels(inflammation$Gender), #Legend text
       col = c(1:2),           #colors to use in legend
       pch=16)                 #specifies how to draw symbols
```



Resources

Congrats! Now you can plot tabular data in R! If you want to learn more, see these resources:

- [Installing packages](#)
- [swirl: Learn R in R](#)
- [R base graphics: an idiot's guide](#)
- [ggplot2](#)
- [Elementary statistics with R](#)
- [Software Carpentry](#)
- [Data Carpentry](#)