

R for Beginners

C. Tobin Magle

February 8, 2016

This material was adapted from the following Data Carpentry lesson <http://www.datacarpentry.org/R-ecology/>

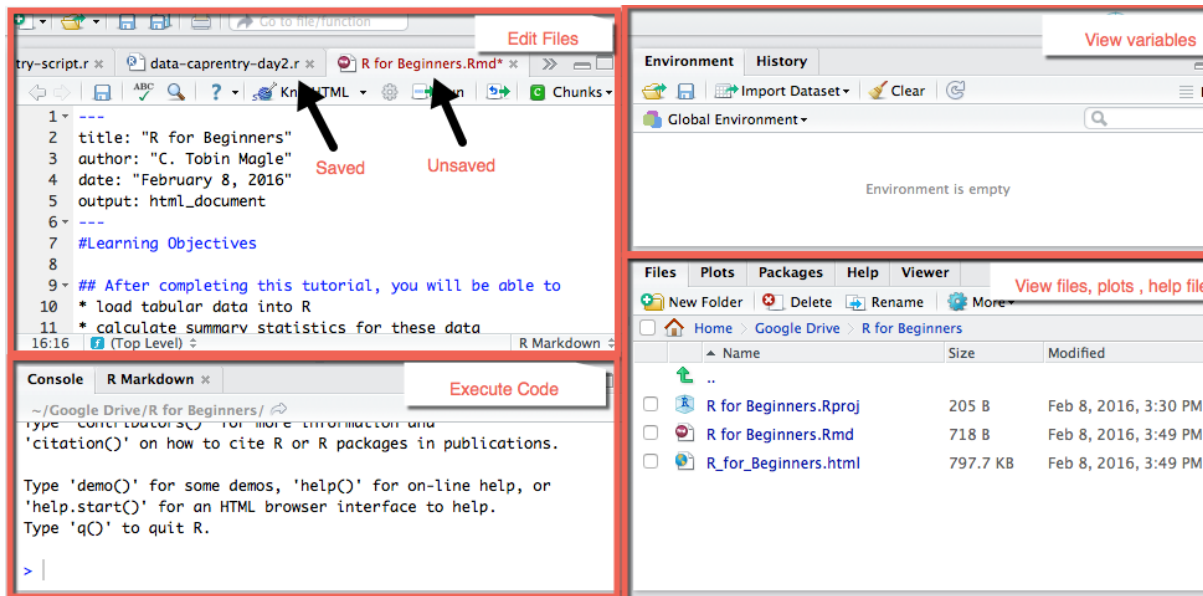
Learning Objectives

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

Introducing R Studio

R studio makes programming in R easier. See what the different windows represent below



Layout Overview:

Upper left: where you write your scripts

Upper right: view the variables

Lower left: where the code is executed

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

Getting ready to code

One of the hardest things about any programming language is remembering the syntax. Missing one semicolon or misspelling a variable or function name 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 **NewFile**
- 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!

Set your working directory

Since 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, use the following syntax:

```
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
```

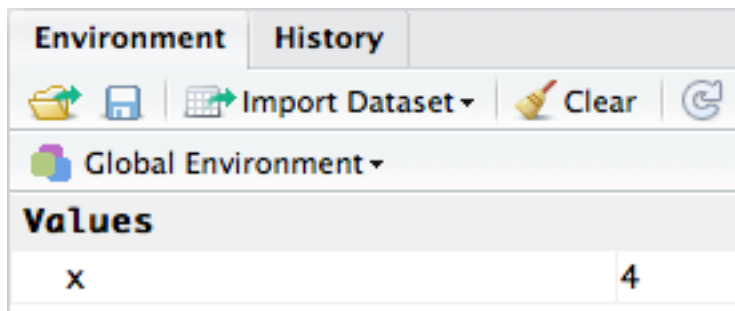
R outputs the number 4.

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

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.

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 also 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")
```

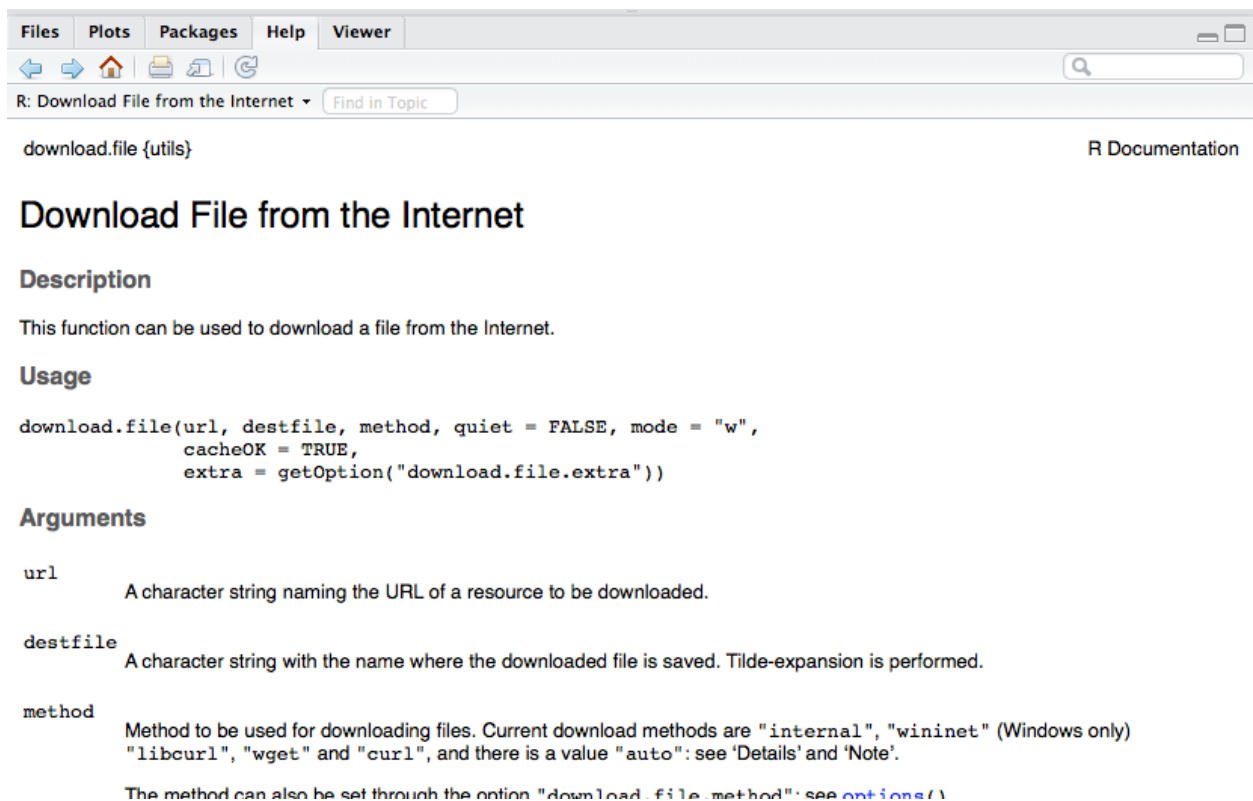
See what I mean about '=' being used for something different in R? It's used to give the function's **arguments** values. Arguments are information that is necessary to run the function. In this case, we need to know where the file is online and where you want to store it locally.

- The **url** argument specifies where the file is on the web
- The **destfile** argument specifies where you want the downloaded data to be stored. We're not including a file path, so it defaults to the working directory. You can name the file whatever you want.
- Make sure both the url and the destination file are in quotes.

If you're not sure what arguments a function can take, type `?`. 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**

```
?download.file # get R documentation for download.file function
```



The screenshot shows the R Studio interface with the Help pane open. The title bar of the Help pane reads "R: Download File from the Internet". The content area displays the documentation for the `download.file` function, including its description, usage, and arguments.

download.file {utils} R Documentation

Download File from the Internet

Description

This function can be used to download a file from the Internet.

Usage

```
download.file(url, destfile, method, quiet = FALSE, mode = "w",
             cacheOK = TRUE,
             extra = getOption("download.file.extra"))
```

Arguments

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 the data into R Now that you've downloaded the data you want to work with, it's time to load it into R. We're going to use the assignment operator in conjunction with a function called **read.csv**.

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:

~

There's a new dataset called **inflammation**. You can call it (mostly) whatever you want, but descriptive names are better. What's a better name for this dataset?

Inspecting your data

Now, 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 variable **x** from earlier. Let's see what class these two items are.

```
class(x)
```

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

```
class(inflammation)
```

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

So, **x** is numeric, which makes sense because it's a number. Inflammation is a data frame. This is a special R format that makes dealing with tabular data easier.

- Each column represents a variable that is being measured in the experiment. In this case, it's patient number, gender, and 30 day readings for antibody levels.
- Each row is a separate observation. In this case, it's all of the patients in the study.
- Each column must have data all of the same 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, use the **View** command.

```
View(inflammation) #opens the data frame in a new tab
```

```
## Warning: running command '/usr/bin/otool' -L '/Library/Frameworks/
## R.framework/Resources/modules/R_de.so' had status 1
```

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 2 ...
## $ Gender: Factor w/ 2 levels "F","M": 2 1 1 2 1 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 X 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 X is meant to be a unique identifier for each patient. This is not a very useful factor.
- **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.

```
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 part of the data rather than the whole. Depending on what you want to do, you'll either want to work with a subset of rows or columns. Let's look at 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 **30dayinflammation**

- You already know about the assignment operator
- The **\$** operator lets you subset the data frame based on column name
- Syntax: **\$**


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

Now you should see a new variable in your global environment with 720 elements.

Now let's look at the mean of this column:

```
mean(day30inflammation)
```

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

The mean of the inflammation scores is ~5.6.

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 for all of the day columns:

```
daymeans<-apply(X=inflammation[,3:32], MARGIN=2, FUN=mean)
```

- **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

```
ptmeans<-apply(X=inflammation[,3:32], MARGIN=1, FUN=mean)
```

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,  
                        subset = (Gender == "F"))
```

- **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).
- note the use of "==" in the Boolean expression. This is an example of a logical operator.

Let's do the same thing for male subjects

```
inflammationM <- subset(x= inflammation,  
                        subset = (Gender == "M"))
```

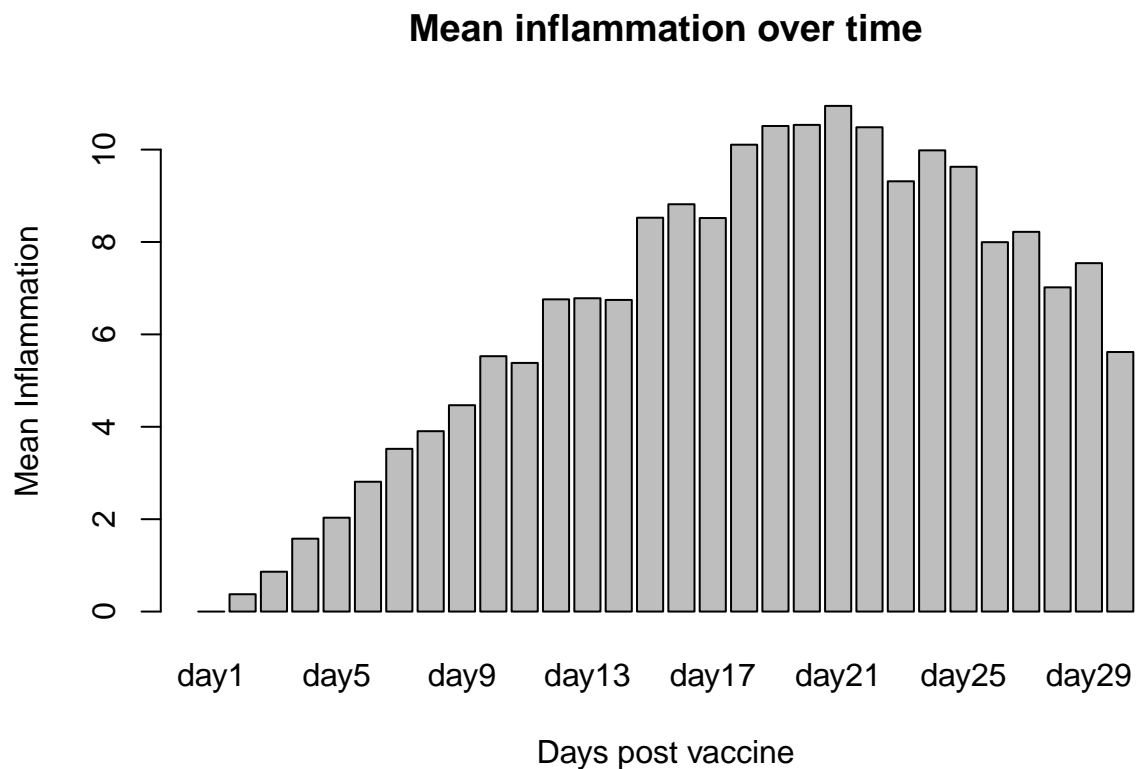
Now we can compare the data for the two groups using a t-test:

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

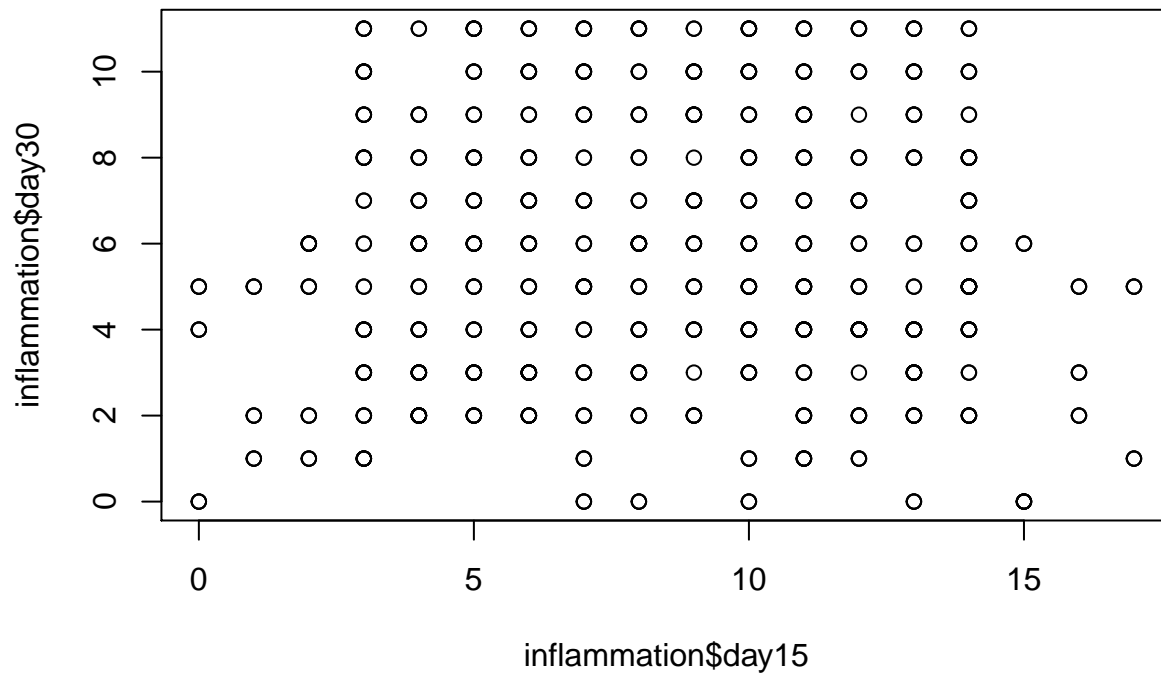
```
##
## Welch Two Sample t-test
##
## data: inflammationF$day30 and inflammationM$day30
## t = -0.38316, df = 717.33, p-value = 0.7017
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -0.5305091 0.3572504
## sample estimates:
## mean of x mean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

Plotting data Lastly, we're going to look at how to plot these data.

```
barplot(height=daymeans, main="Mean inflammation over time",
        xlab="Days post vaccine",
        ylab="Mean Inflammation")
```

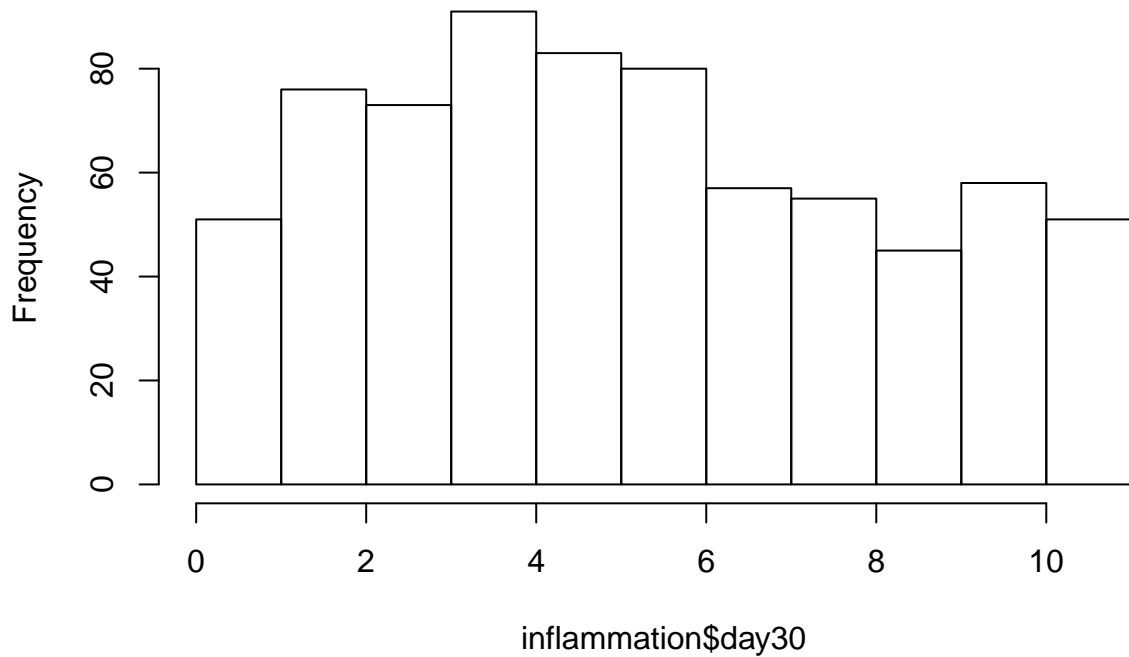


```
plot(inflammation$day15, inflammation$day30)
```



```
hist(inflammation$day30)
```

Histogram of inflammation\$day30



Resources

Congrats! Now you know enough R programming to be dangerous! If want to learn more, see these resources:
 * Loading packages * swirl