Mathematics of Finance, Courant Institute, Fall 2015
http://www.math.nyu.edu/faculty/goodman/teaching/mathFin/index.html
Always check the class message board before doing any work on the assignment.

Getting started with R

Corrections: (check the class message board)

This is a rapid introduction to some aspects of computing and visualization in R. This class uses R because it seems to be the scripting language that is easiest to install and use for all the platforms people are likely to have. But experience shows that following the instructions will not give the desired outcomes for all people on all platforms. There is a special form of bad luck for newbies. We've all been there. It is likely others will have similar problems. Post your issue to the class message board or get some help another way.

Take the stuff about coding standards seriously, especially if you are a beginning programmer. If you spend 15 minutes making your code more readable and automatic, you will save days of pointless debugging. Follow the links on the class Resources page to read more about programming style.

1. If you do not have the `R` package on your computer, install it from the web. There are instructions for this on the class web page. It *should* be easy. When you start up `R`, you should get a `command window` with a prompt that looks like this;

   ```
   >
   ```

   Type the command `x = 2.3` . It creates an *object* (mathematicians say *variable*) called `x`, assigns it the value 2.3, then gives you a prompt for the next command. Now the R window should look like

   ```
   > x = 2.3
   >
   ```

   This is an *assignment* statement. You may use `<-` instead of `=` as the assignment `operator`. If you do that, it looks like this:

   ```
   > x <- 2.3
   >
   ```

   I use `=` because that's what it is in most other programming languages. Pure `R` programmers often use `<-` because it makes clear that you are not asserting that the two sides are equal, but changing the value on the left to be equal to the value on the right. You can use the one you prefer.

   If you just type the name of the object, `R` replies with `[1]` and then the value currently assigned to that object. So type `x` , and `R` will reply with `[1]` and then its current value:

```
> x = 2.3
> x
[1] 2.3
>
```

If you type a more complicated arithmetic expression, R will evaluate the expression and print the value. For example, typing x*x should give $2.3^2 = 5.29$. The R window should look like this:

```
> x
[1] 2.3
> x = 2.3
> x
[1] 2.3
> x*x
[1] 5.29
>
```

R will give an *error message* if you ask the impossible. For example, you can ask it to add a number it doesn't know, or perform an operation that is not defined

```
> x+z
Error: object 'z' not found
> x***3
Error: unexpected '*' in "x***"
>
```

This can be frustrating, but it does no harm to the computer. It's often a simple typo, for example z instead of x in the first one, or x***3 instead of x**3 in the second (x**3 means x^3 in R). Error messages try to be helpful, but are not always.

2. Every object in R has a *type*, also called its *class*. The object x above is a floating point number; its type is *float*. Two other important types are *vector* and *character string*. You assign x to the vector $(2, 1, -1, 0)$ using the c() function ("c" stands for "catenate", which means *concatenate*):

```
> x = c(2, 1, -1, 0)
> x
[1]  2  1 -1  0
> x+x
[1]  4  2 -2  0
> x*x
[1] 4 1 1 0
> 3*x
[1]  6  3 -3  0
```

Notice that addition and multiplication are done elementwise. Also, when you do an assignment, whatever value the object had before is forgotten. Even the type can change; here it goes from `float` to `vector`.

A `character string` (or just `string`) object is a sequence of characters in quotes:

```
> label = "time, in years"
> label
[1] "time, in years"
```

Here, `label` is the name of an object whose `type` is `string` and whose value is `"time, in years"`. Names are *case sensitive*, which means that `Label` and `label` are different names.

```
> Label = "time, in days"
> Label
[1] "time, in days"
> label
[1] "time, in years"
```

The `R` function `paste()` puts together two (or more) strings into a single string, while the `c()` function makes a "vector" (not a mathematical vector) whose "components" are the two strings:

```
> paste( Label, label)
[1] "time, in days time, in years"
> c( Label, label)
[1] "time, in days"  "time, in years"
```

Strings are important in numerical computing because they help us *format* the *output* (results of a calculation) in a way that is easy to read. Here is an example. The `R` function `sprintf()` (the "s" is for "string", "print" is for printing, and "f" is for "format": you format the numbers and text into a line string for printing) can produce a string that represents the value of a float. Note the different ways to make two-word variable names, the underscore (`x_string`) and capitalizing each word (`OutputLine`). The following creates strings that represent some numbers, does some computation, then uses `paste()` to put all the numbers into a sentence for printing. We will do this kind of thing a lot, particularly for making titles for plots.

```
> x = 2.3
> x_string = sprintf("\%5.3f",x)
> x_string
> y = 3.4
> y_string = sprintf("%5.3f",y)
> y_string
[1] "3.400"
> z = x+y
> z_string = sprintf("%5.3f",z)
> z
[1] 5.7
> OutputLine = paste( "Adding ", x_string, " to ", y_string, " gives ", z_string)
> OutputLine
[1] "Adding  2.300  to  3.400  gives  5.700"
```

3. You should do almost all of your `R` work using *scripts* rather than by typing the commands one by one as above. An `R` script is a file that contains a series of `R` commands. The `R` *interpreter* reads the lines from the file one by one and executes them in `R`. For example, suppose you create a file `AddNumbers.R` that contains the `R` commands we just used:

   ```
   # add two numbers and print the result

   x = 2.3
   y = 3.4
   z = x + y
   x_string = sprintf("%5.3f",x)
   y_string = sprintf("%5.3f",y)
   z_string = sprintf("%5.3f",z)

   OutputLine = paste( "Adding ", x_string, " to ", y_string, " gives ", z_string)

   cat(OutputLine)
   ```

   The `R` function `source()` reads an `R` script and uses the `R` interpreter to *execute* it. We execute `AddNumbers.R` by typing `source("AddNumbers.R")` at the command prompt. The interpreter prints whatever output is generated:

   ```
   > source("AddNumbers.R")
   Adding  2.300  to  3.400  gives  5.700
   ```

   Note a few differences between the lines in the script file `AddNumbers.R` and the corresponding lines typed directly at the command prompt as before. The first line is a *comment*. The interpreter ignores any character after the *comment* character, which is # in `R`. You put comments into scripts to help you, or someone else reading the file, understand what the script does. The script also has blank lines that visually separate different parts of the script. There are also blank spaces to separate things and make it easier to read. This is part of *programming style*, which is very very important for scripts that take more than a few minutes to write.

Finally, you see that the last line is not just `OutputLine`, which prints the result somewhere we can't see, but `cat(OutputLine)`. The `R` function `cat()` ("cat" is also for "catenate"; it's a long story.) prints out the output string to the command line where we typed `source()`.

One of the points of a script is that it is easy to change something and do it again. For example, if we change the line defining `x` to `x = -2.3` then we get:

```
> source("AddNumbers.R")
Adding  -2.300  to  3.400  gives  1.100
```

Here's a similar example. The file `QuadraticFormula.R` is:

```
# the quadratic formula

a = 1
b = 5
c = 2

root = ( -b + sqrt(b**2 - 4*a*c))/(2*a)

OutputLine = sprintf("a = %6.3f, b = %6.3f, c = %6.3f\n", a, b, c)
cat(OutputLine)

OutputLine = sprintf("The larger root is %6.3f\n",root)
cat(OutputLine)
```

And then we can do:

```
> source("QuadraticFormula.R")
a =  1.000, b =  5.000, c =  2.000
The larger root is -0.438
```

Quickly change the value of `c` and do it again (something goes wrong):

```
> source("QuadraticFormula.R")
a =  1.000, b =  5.000, c = 20.000
The larger root is    NaN
Warning message:
In sqrt(b^2 - 4 * a * c) : NaNs produced
```