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.

Loops with R

Corrections: (check the class message board)

A `loop` in a program is a piece of code that gets *executed* many times. Much of the numerical computing applied mathematicians do consists of loops. For example, consider the rectangle rule approximation to an integral

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} \Delta x f(x_i)\ . \tag{1}$$

Here $\Delta x f(x_i)$ is the area of a little rectangle whose length is Δx and height is $f(x_i)$. The evaluation points are $x_0 = a$, $x_1 = a + \Delta x$, $x_2 = a + 2\Delta x$, etc. The length, Δx, is chosen so that n intervals of length Δx exactly fill the integration interval $[a, b]$. That means $\Delta x = (b - a)/n$. We would use a formula like (1) because we want to know the answer, the number, and we cannot find a formula for the it. All those integrals you did in calculus were exceptions to the general rule that most integrals cannot be done by formula. An `R` script to evaluate the sum in (1) probably uses a *loop*, which is some lines of code that evaluate the rectangle area $\Delta x f(x_i)$, add this area to the total, and increment x_i.

Download the code `Loops.R` and run it: `> source("Loop.R")`. Open `Source.R` in your editor. It has some examples of loops. Each example loop has a *code block* that is to be executed many times, and *control statement* that determines how is code block is executed. A code block in `R` is a sequence of statements contained in *curley braces*, often called *curlies*. The code block in the first example is

```
{
   sum = sum + i
   cat("sum is ", sum, " and i is ", i, "\n")
 }
```

It starts with the *open* curley brace "{", and ends with the *close* curley "}". These open and close the code block. There are two lines of code that form the *body* of the code block. These are indented three spaces to make the code block easy to spot in the code. Indention is not required by the language `R`, but the rules of programming style require it. You will lose points on coding assignments if you do not indent code blocks. The programming style links on the course web page have more discussion of this.

The control statement for this loop comes before the code block that is the body of the loop. It is `for( i in 1:n)`. The open curley of the loop body is on the same line as the control statement. Some people say this is bad

programming style. Other people do it, as I do. You have to decide which style you will use, it's a fashion choice. This control statement defines a *for loop*. The body of the for loop will be executed with the variable `i` having values `1`, `2`, ..., `n`. More precisely, the expression "`1:n`" defines an `R` *list*, which is the sequence $(1, 2, \ldots, n)$. The body of the loop will be executed once for every element in the list, with `i` equal to that element. The order is the order of the elements in the list. The output shows what happens. Also pay attention to the comments in the `R` code. A common bug is to forget to *initialize* the variable `sum`. The statement `cat("sum is ", s...` is a *print statement*. Good programmers put print statements into the body of a loop to find programming mistakes, *bugs*. Print statements let you "watch" the program execute and check that it does what you wanted it to do. Once the program is correct, you can remove print statements you used for debugging. It is common to *comment out* print statements, which means putting a hashtag # before the statement, which makes it a comment. If you want the print statement back, you just edit away the hashtag.

The second example concerns lists in `R`. The first print `cat("lis...` shows that `1:n` produces a list. The second one shows that the list has changed. These are lists of numbers, but any `R` objects can be in lists.

Example 3 has an *if test* and a `break` out of the loop. Inside the `if` is a *logical expression* that evaluates to `TRUE` or `FALSE`. Here, the conditional is `sum > target`. If the value of `sum` is greater than the value of `target`, this evaluates to `TRUE`. Otherwise it evaluates to `FALSE`. If it evaluates to `TRUE`, then the next statement is executed. Otherwise it is not. In this case, the next statement is `break`, which means: stop executing the loop and start executing at the statement after the end of the code block, the statement after the close curley. Here, that is `cat( i, " t...` .

Example 4 has `if` and `else`. The `else` part is executed if the conditional expression evaluates to `FALSE`. These are two *branches* of the if test. Here, the `else` branch is executed if the value of `sum` is not larger than the value of `target`. The output shows this in action. Notice that Examples 3 and 4 compute the same thing, the number `36`. But the code of Example 3 is simpler, shorter and easier to understand, therefore better.