Introduction to computational programming
Introductory Exercise
Loops and flow control in *R*

David M. Rosenberg
University of Chicago
Committee on Neurobiology

Version control information:
Last changed date: 2009-10-01 17:59:47 -0500 (Thu, 01 Oct 2009)
Last changes revision:68
Version: Revision 68
Last changed by: David M. Rosenberg

October 6, 2009

Overview

In this exercise we will explore the concepts of *flow control* and *loops*, two important tools for allowing the computer to do the "work." We will start by reviewing *R*'s facilities for boolean logic and how they enable simple control structures. Next we will explore the use of loops to minimize repetitive chunks of code and the conventions for their use.

Don't forget to complete and submit the exercises at the end of this document to your TA.

Part I
Tutorial

1 Flow control

1.1 Boolean Logic

One of the fundamental concepts of computer programming, and one often unfamiliar to non-programmers, is the concept of *boolean logic*. Boolean algebra is a simple mathematical system containing two values (**True**

and **False**) and three operations (**AND**, **OR** and **NOT**). The following table outlines how boolean logic is represented in *R* .

	R representation	context	meaning
true	`TRUE`	`a`	`a` is true
false	`FALSE`	`b`	`b` is true
not	`!`	`!a`	inverse of `a`
and	`&` and `&&`[1]	`a & b`	`a` and `b`
or	`|` and `||`[1]	`a | b`	`a` or `b`

The conventions for using these operations are bit unusual.

- **AND** (`&`) takes two arguments, `a` and `b`. One is placed before the `&` and one after. The expression `a & b` evaluates to `TRUE` if and only if `a` evaluates to `TRUE` and `b` evaluates to `TRUE`.
- **OR** (`|`) takes two arguments, `a` and `b`. One is placed before the `|` and one after. The expression `a | b` evaluates to `TRUE` if `a` evaluates to `TRUE`, if `b` evaluates to `TRUE`, or if both `a` and `b` evaluate to `TRUE`.
- **NOT** (`!`) takes one argument, `a`, placed directly after the `!`. The expression `!a` evaluates to `TRUE` if and only if `a` evaluates to `FALSE`.

Boolean logic is extremely important for *conditionals*, which we will explore below.

One final aspect of *boolean logic* which we must consider is that of *boolean indexing*. Recall that we can extract a subset of a vector by putting a vector of integer indexes in brackets.

```
> myVec <- letters
> myVec

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

> myVec[1:5]           # First five letters

[1] "a" "b" "c" "d" "e"
```

You can also use a *logical vector* (a vector of only `TRUE` or `FALSE` members) to index a vector as well. The result of such an operation is a vector composed of all the members of the original vector which were indexed by `TRUE`. An example may help to clarify this

```
> myVec <- letters
> myVec

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

> idx <- 1:26
> bool_idx <- idx < 10              # logical vector
> bool_idx
```

[1]The conjunction (*and*) and disjunction (*or*) each have two syntactical forms. The single (`&` and `|`) each return a logical `vector` equal in length to the the length of their arguments (pairwise comparisons performed). In contrast, the doubled forms (`&&` and `||`) return a single logical value which is *TRUE* if and only if all pairwise comparisons are *TRUE*.

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE

> myVec[bool_idx]                    # the first 10 letters of the alphabet

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

1.1.1 Conditionals

Conditionals are "tests" which return either `TRUE` or `FALSE` with respect to a particular variable. The most familiar conditionals are likely the *comparison operators*

comparison	*R* representation	context	notes
equality	`==`	`a == b`	True if `a` and `b` have the same value
inequality	`!=`	`a != b`	synonomous with `!(a == b)`
less than, greater than	`<`, `>`	`a < b`	true if `a` is less than `b`
less than or equal to	`<=`, `>=`	`a <= b`	synonomous with `(a < b) | (a == b)`
set-theoretic inclusion	`%in%`	`a %in% b`	true if vector `b` contains `a` as one of its members
vector union	`all()`	`all(a)`	true if every member of vector `a` is true
vector intersection	`any()`	`any(a)`	true if any member of vector `a` is true

The most basic use of conditionals involves the *R* `if` command, which has the following syntax:

```
> if (CONDITION) {
+   BLOCK_1
+ } else if (CONDITION_2) {
+   BLOCK_2
+ } else {
+   ELSE_BLOCK
+ }
```

The capitalized expressions represent code which may be changed:

expression	meaning
`CONDITION`	a condition which evaluates to `TRUE` or `FALSE`
`BLOCK_1`	a sequence of commands evaluated if `CONDITION` evaluates to `TRUE`
`CONDITION_2` (optional)	a second condition which is evaluated only if `CONDITION` evaluates to `FALSE`. Note that as many `else if()` clauses may be used as one chooses.
`BLOCK_2` (optional)	sequence of commands to be evaluated if `CONDITION_2` is evaluated and `TRUE`
`ELSE_BLOCK` (optional)	sequence of commands to be evaluated if none of the `CONDITION` blocks evaluate to be true.

An example may help to clarify this syntax.

```
> x1 <- 2
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 16
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 128
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 1024
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
```

This example should produce the following output:

```
2  has only 1 digit.

16  has two digits.

128  has three digits.

1024  has more than three digits.
```

1.1.2 Flow control / loops

Flow control, the process by which commands are "selected" and evaluated is defined by two major constructs: *conditionals* (seen in the previous section) and *Loops*. A *loop* is a section of code which is repeatedly evaluated

(often with variables assuming different values). In *R* , there are two main types of loops which we will consider.

While loops are used to evaluate a set of commands based on the result of a *conditional* test. The syntax for a `while` loop is as follows:

```
> while(CONDITION) {
+   BLOCK
+ }
```

Here, ***CONDITION*** is a conditional statement (see above), and ***BLOCK*** is a series of commands which are evaluated, in sequence, until ***CONDITION*** evaluates to `FALSE`. As an example, consider the fibonacci sequence $f_n = 0, 1, 1, 2, 3, 5, \ldots$, defined by:

$$f_n = \begin{cases} 0 & \text{if } n = 1, \\ 1, & \text{if } n = 2, \\ f_{n-1} + f_{n-2}, & \text{if } n > 2 \end{cases} \tag{1}$$

Suppose that we wanted to find the first number in the fibonacci sequence greater than 1000.

```
> x0 <- 0
> x1 <- 1
> while(x1 < 1000) {       # x1 < 1000 is the condition
+   newX1 <- x0 + x1       # -\
+   x0 <- x1               # --+ -- these three lines make up the BLOCK
+   x1 <- newX1            # -/
+ }
> x1

[1] 1597
```

The second of our looping constructs, the `for` loop, is used to repeat a sequence of commands a predetermined number of times.

```
> for (COUNTER in VECTOR) {
+   BLOCK
+ }
```

Here ***COUNTER*** is a variable which changes from one iteration of the loop to the next and ***VECTOR*** is a vector of values. ***BLOCK*** is evaluated once for each value in ***VECTOR***. In each iteration, ***COUNTER*** is assigned a new value, taken from the members of ***VECTOR***.

Continuing with the previous example, suppose now you wanted to know the first 20 values of the fibonacci sequence. Then:

```
> xVec <- c(0, 1)
> for (ii in 3:20){
+   xminus1 <- xVec[ii-1]
+   xminus2 <- xVec[ii-2]
+   newVal <- xminus1 + xminus2
+   xVec <- c(xVec, newVal)
+ }
> xVec
```

```
 [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
[16] 610 987 1597 2584 4181
```

There are two special commands for dealing with loops

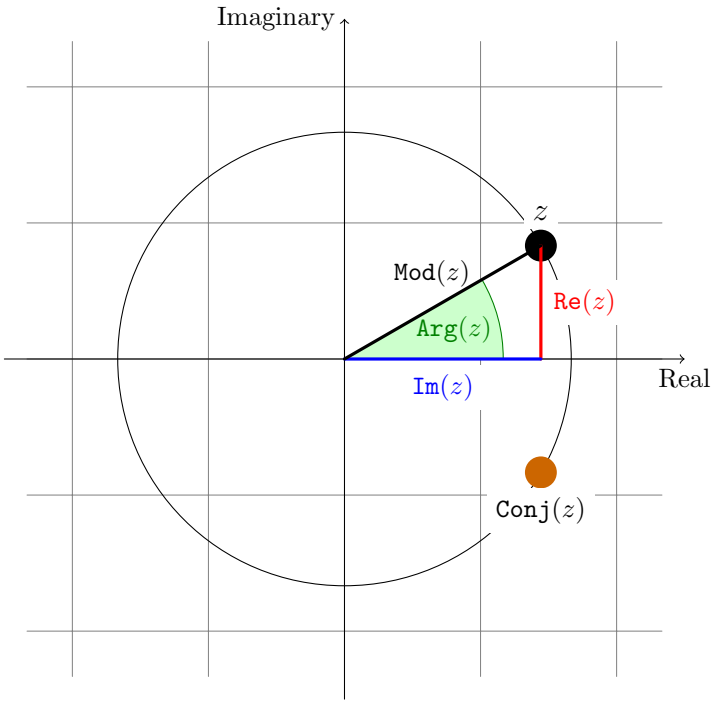
- `break` immediately terminates the loop and continues evaluation at the first command following the end brace of the loop.
- `next` ends the current loop iteration and starts the next one.

2 Complex numbers

2.1 Syntax

As you (may) recall from the **Guide to Using *R***, the *R* interpreter has built-in support for complex numbers. Shown below are example code for generating complex values in *R* and a summary of the five major complex operations supported by *R*.

```
> myVar <- 3 + 4i
> myVar

[1] 3+4i

> is.complex(3 + 4i)

[1] TRUE

> is.complex(3)

[1] FALSE

> myVar2 <- complex(real=3, imaginary=4)
> myVar3 <- complex(argument=0.9272952, modulus=5)
> myVar2

[1] 3+4i

> myVar3

[1] 3+4i
```



These functions assume that z is the complex number $z = a + bi$.

Function	Syntax	Value
real part	`Re(z)`	a
imaginary part	`Im(z)`	b
argument	`Arg(z)`	$\arctan(\frac{b}{a})$
modulus	`Mod(z)`	$\sqrt{a^2 + b^2}$
complex conjugate	`Conj(z)`	$a - b\,i$

Part II

Exercises

1. (a) Starting with the fibonacci sequence examples from the tutorial, write a code chunk which takes as input a number `k` and finds the largest fibonacci number less than or equal to `k`

   (b) Write a code chunk which takes as input three numbers (say, `k`, `x`, and `y`) and prints either the second number (`x`) or the third number (`y`) depending on which is closer to `k`. (*Hint:* the functions `min()`, `max()`, and `abs()` may be helpful. See the online documentation for their usage.)

   (c) Write a code chunk which takes as input a number `k` and returns the fibonacci number with is closest to `k`.

2. (a) Write a code chunk which takes a "sorted" numeric vector of length 2 and another numeric vector of length 1 and prints a single "sorted" vector of length 3.

   (b) Write a code chunk which takes an "unsorted" numeric vector of length 2 and prints the values of that vector, "sorted."

   (c) Write a code chunk that takes an "unsorted" numeric vector of length 10 and prints the sorted values to the screen.

3. Consider the equation

$$x^k - 1 = 0, \quad x \in \mathbb{C},\, k \in \mathbb{N} \tag{2}$$

Write a code chunk that takes in integer (k) as input and prints all values x which satisfy the above equation.