Programming Tools

I) Iteration

Iteration is a loop or repeatedly executed instruction cycle, with only a few changes in each cycle. In programming language that are not matrix or array-oriented, like C, Pascal, or FORTRAN, even a simple matrix multiplication needs three nested loops (over rows, columns, and the indices). Since R is matrix-oriented, these operations are much more efficient and easy to formulate in mathematical terms. This means they are faster than loops and the code is much easier to read and write.

The following table contains the different forms of loops.

Forms of loop	Syntax
for loop	for (index in range) { expressions to be executed }
while loop	while (condition) { expressions to be executed }
repeat loop	repeat { expressions to be executed if (condition) break}

For loops

for (variable in sequence) expression

The *expression* can be a single R command - or several lines of commands wrapped in curly brackets:

```
for (variable in sequence) {
    expression
    expression
    expression
}
```

Here is a quick trivial example, printing the square root of the integers one to ten:

```
> for (x in c(1:10)) print(sqrt(x))
[1] 1
[1] 1.414214
[1] 1.732051
[1] 2
[1] 2.236068
[1] 2.449490
[1] 2.645751
[1] 2.828427
[1] 3
[1] 3.162278
```

While loops

In R a while takes this form, where *condition* evaluates to a boolean (True/False) and must be wrapped in ordinary brackets:

```
while (condition) expression
```

As with a for loop, *expression* can be a single R command - or several lines of commands wrapped in curly brackets:

```
while (condition) {
   expression
   expression
   expression
}
```

NOTES:

- there is no explicit to return argument in loops. Use a print or cat functions to print out results.
- The main different between while and repeat is that it is possible not to enter the while loop at all where the repeat is entered at least once.

while	repeat
• Wrapped the enter condition • It is possible not do any expression.	• Wrapped the exit condition • do at least one expression.

Example :

Calculate the sum over 1, 2, 3, . . . until the sum is larger than 100 by using different loops.

1. while loop:

```
n=0;sumn=0
while (sumn<=100)
{ n=n+1
  sumn=sumn+n
}
```

2. repeat loop

```
n=0;sumn=0
repeat
{ n=n+1
  sumn=sumn+n
  if (sumn>= 100) break}
```

3. for loop

```
n=0;sumn=0
for (i in 1:100)   sumn=sumn+i        # Is this command give the exact answer?
                                      # It is not flexible to use for here
```

If we want to print sumn each time

What is the difference between these two commands?

n=0;sumn=0
while (sumn<=100)
{ n=n+1
  sumn=sumn+n
print(sumn)
}
	n=0;sumn=0
	while (sumn<=100)
	{ n=n+1
	  sumn=sumn+n
	if(sumn<=100)
	{ # this Curly brackets is unnecessary here
	print(sumn)
	}
	}

Try if sumn start with 0, and try if sumn start with 101

Example:

Create this matrix by using loop

$$\begin{bmatrix} 1 & 4 & 7 \\ 8 & 2 & 9 \end{bmatrix}$$

1- for loop

```
x=c(1,4,7,8,2,9)
n=1
m=matrix(,2,3)
for(i in 1:2)
{
for (j in 1:3)
{
m[i,j]=x[n]
n=n+1
}
}
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    8    2    9
```

2- while loop:

```
x=c(1,4,7,8,2,9)
n=1
i=1
j=1
m=matrix(,2,3)
while(i<=2)
{
while (j<= 3)
{
m[i,j]=x[n]
n=n+1
j=j+1
}
j=1
i=i+1
}
m
```

3- repeat loop

```
x=c(1,4,7,8,2,9)
n=1
i=1
j=1
m=matrix(,2,3)
repeat
{
repeat
{
m[i,j]=x[n]
n=n+1
j=j+1
if(j>3)break
}
j=1
i=i+1
if(i>2)break
}
m
```

Another method:

```
x=c(1,4,7,8,2,9)
n=1
i=1
j=1
m=matrix(,2,3)
test=T
while(test)
{
while(test)
{
m[i,j]=x[n]
n=n+1
j=j+1
if(j==4)
test=F
}
test=T
j=1
i=i+1
if(i==3)
test=F
}
```

Example:

to print out the first few Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21,34 where each number is the sum of the previous two numbers.

```
x <- c(0,1)
while (length(x) < 10) {
    position <- length(x)
    new <- x[position] + x[position-1]
    x <- c(x,new)
}
print(x)
```

To understand how this manages to append the new value to the end of the vector x, try this at the command prompt:

```
> x <- c(1,2,3,4)
> c(x,5)
[1] 1 2 3 4 5
```

The looping variable i values can be of any mode

a) A numeric looping variable :

```
for ( i in c(3, 2, 9, 6))
print ( i^2)
```

or

```
x  <– c(3, 2, 9, 6);  for ( i in 1:4)  print((x[i]^2)
```

b) A character looping variable:

```
 transport.media=c("car","bus","train")
for (i in transport.media)
print(i)
```

II) Conditional Execution (The if statement)

- if (condition) { expression 1 }
- if (cond 1) { expr 1 }

  ```
  else if ( cond 2 ) { expr 2 }
      else { last expr }
  ```
- ifelse (condition, expression for true, expression for false)

Examples:

```
if (mode(x)!="character") log(x)  # try when x="d",3,NA
# test 2 conditions
if (mode(x)!="character" && x>0) log(x)
```

Note that:

|| && not | &

```
x=c(4,1,-9,0)
logx=rep(0,length(x))    # same as logx=0  (any value)
for (i in 1:length(x))
{  if (x[i]>0)  logx[i]=log(x[i])
   else     logx[i]=NA}
#same as
 ifelse(x>0,log(x),NA)  # evaluate a condition for the whole vector or array
ifelse(x>0,sqrt(x),NA)
```

III) Writing Function

Functions do things with data
“Input”: function arguments (0,1,2,…)
“Output”: function result

Syntax:

```
        Function_name <- function ( input arguments )
        {
           function.body ( R expressions )
            return ( list ( output argument ))
         }
```

then you can call the function using the calling routine
function_name (argument)

Example:

```
add = function(a,b)
{ result = a+b
  return(result) }
add(7,8)
```

Note that:

1. All variables declared inside the body of a function are local and vanish after the function is executed.
2. Better to use return function if we need more than one value to return from function.

Examples:

```
Cubic<-function(xx){return(xx^3)}
Cubic(3);xx

Cubic<-function(xx){xx^3}            # same as above
Cubic(3)

Cubic2<-function(xx)
{y=2^xx;return(xx^3,y)}
Cubic2(3)

Cubic2<-function(xx)
{y=2^xx;y2=xx^3}

Cubic2(3)      # Guess what is the output??????????????????/
```

Example:

Writing Functions

This following script uses the function() command to create a function (based on the code above) which is then stored as an object with the name Fibonacci:

```
Fibonacci <- function(n) {
   x <- c(0,1)
   while (length(x) < n) {
      position <- length(x)
      new <- x[position] + x[position-1]
      x <- c(x,new)
   }
   return(x)
}
```

Once you run this code, there will be a new function available which we can now test:

```
> Fibonacci(10)
 [1]  0  1  1  2  3  5  8 13 21 34
> Fibonacci(3)
[1] 0 1 1
> Fibonacci(2)
[1] 0 1
> Fibonacci(1)
[1] 0 1
```

That seems to work nicely - except in the case n == 1 where the function is returning the first *two* Fibonacci numbers! This gives us an excuse to introduce the if statement.

The If statement

In order to fix our function we can do this:

```
Fibonacci <- function(n) {
   if (n==1) return(0)
   x <- c(0,1)
   while (length(x) < n) {
      position <- length(x)
      new <- x[position] + x[position-1]
      x <- c(x,new)
   }
   return(x)
}
```

In the above example we are using the simplest possible if statement:

if (condition) expression

The if statement can also be used like this:

> *if (condition) expression else expression*

And, much like the while and for loops the *expression* can be multiline with curly brackets:

```
Fibonacci <- function(n) {
   if (n==1) {
      x <- 0
   } else {
      x <- c(0,1)
      while (length(x) < n) {
         position <- length(x)
         new <- x[position] + x[position-1]
         x <- c(x,new)
      }
   }
   return(x)
}
```

Example

Create your own function

```
   X<-seq(2,10,2);y<-2:6
   F<-(3*X^4)/(X+y);F
   F1<-function(X,y){(3*X^4)/(X+y)}
   W<-F1(X,y);W

   > X<-seq(2,10,2);y<-2:6
   > F<-(3*X^4)/(X+y);F
   [1]   12.0000  109.7143  388.8000  945.2308 1875.0000
   > F1<-function(X,y){(3*X^4)/(X+y)}
   > W<-F1(X,y);W
   [1]   12.0000  109.7143  388.8000  945.2308 1875.0000
```

```
#function that compute mean and standard error
std.error<-function(x)
{ std.error=sqrt(sum(x-mean(x))^2)/(length(x)*(length(x)-1))
return(list(mean(x),std.error))}
x=c(1,5,7,8,4,6,9)
std.error(x)
```

Construct a function that assign an even number to 1, and an odd number to 0 only at a line (use ifelse)