1 Flow Control: If and Switch Statements

It's useful to calculate quantities conditional on some condition being true. The if statement in R lets us do this in a fairly natural way. As an example, consider:

```r
> x <- -2
> if (x>0) 1000 else 5000
[1] 5000
```

This first tests whether x>0. If true, it returns 1000, otherwise it returns 5000. The general form for an if expression is

```r
if (condition) [expression1] else [expression2]
```

This evaluates the condition. If it is true, it returns the value of [expression1], if not, it returns the value of [expression2]. We can use if statements to define functions:

```r
> absolutevalue <- function(x) { if (x<0) -x else x }
> absolutevalue(-3)
[1] 3
> absolutevalue(5)
[1] 5
```

In an if statement, expression1 or expression2 could contains multiple lines of commands; if so, that expression should be enclosed in curly brackets. For example:

```r
> absplus1 <- function(x){ if (x+1 < 0) {y <- -x + y+1} else x+1 }
```

This function takes an input x, and checks if x+1 is negative. If so, it first calculates y=-x, then calculates y+1 (which is then returned). If x+1 is not negative, the function just returns x+1. Note that the variable y is defined inside the function, so it is only accessible inside the function.
2 Flow Control: Looping

The `for` function can be used to repeat a set of calculations many times. The basic syntax is:

```r
for (var in seq) expr
```

where `var` is a variable name, `seq` is a set of values we want to assign to `var`, and `expr` is an expression involving `var` that we want to evaluate. It's easiest to see this in action. Consider the following command:

```r
> for (i in (1:5)) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Here, `var=i`, and `seq=(1:5)`, a vector containing the numbers 1 through 5. So R first sets `i=1`, then evaluates `print(i)` (print the number 1). Then R sets `i=2`, and prints the number 2. And so on.

The expression to be evaluated can involve many steps. In this case, they should be enclosed in curly brackets `{,}` to ensure that R treats them as a single expression:

```r
> for (i in (1:10)) {j <- i-1
+ k <- j*2
+ print(k)
+ }
[1] 0
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
[1] 12
[1] 14
[1] 16
[1] 18
```

Often, we want to record the results of the calculations in a vector. We first create a vector to store the results:

```r
> results <- vector(mode="numeric", length=10)
```
Then in the expression part of the for loop, we make sure to store the result at each step:

```r
> for (i in (1:10)){ j<- i-1
+ k <- j*2
+ results[i] <- k}
```

The vector results now contains the calculations:

```r
> results
[1] 0 2 4 6 8 10 12 14 16 18
```

### 3 Sapply

Loops in R are relatively slow. For the simple examples given above, they are still nearly instantaneous, but if the number of iterations is very large, this can take a long time. (Compiled languages like C or Fortran are usually much faster for iterative calculations.)

An alternative construction can be used in many cases, which can improve both speed and interpretability. Suppose we want to repeatedly apply a certain function to a vector of inputs. Then we can use `sapply(X,FUN)`, which applies the function `FUN` to each element of `X`. The same calculation we did in the for loop can be implemented as follows:

```r
> i <- (1:10)
> i
[1] 1 2 3 4 5 6 7 8 9 10
> newfun <- function(x) 2*(x-1)
> sapply(X=i, FUN=newfun)
[1] 0 2 4 6 8 10 12 14 16 18
```

Here, we first define the vector `i` to contain the numbers 1 through 10. Then we define a function `newfun` which subtracts one from its input, then doubles it. Then we use `sapply` to apply `newfun` to each element of `i`.

Notice that `sapply` takes a *function* as an input. This is an aspect of R that is very powerful: it is a "functional" programming language in that functions are first-class objects. They can be passed to other functions, and a function can even return another function as its output.