Writing functions

Part II

Control Flow
Tests with if and switch

- The general form of the if construction has the form

```
if(test)
{     ...true statements...   }
else if
{     …other statements…   }
…
else
{     ...false statements...    }
```

where test is a logical expression like x < 0, x < 0 & x > -8.

Control Flow
Tests with if and switch

- R evaluates the logical expression if it results in TRUE then it executes the true statements.
- If the logical expression results in FALSE then it executes the false statements.
- Note that it is not necessary to have the else block.

Ex 1

- Absolute value of a number
- Abs1 <- function(x) {if (x<0) return(-x) else return(x)}
- Abs1(10)
- Abs1(-10)

Ex 2

- Return the sign of a number, -1 for negative, 0 for zero, and +1 for positive
- sign1 <- function (x){
- if (x<0) -1
- else if (x>0) 1
- else 0
- }
- sign1(-3)

Ex3

- Adding two vectors in R of different length will cause R to recycle the shorter vector. The following function adds the two vectors by chopping of the longer vector so that it has the same length as the shorter.

```
myplus <- function(x, y){
n1 <- length(x)
n2 <- length(y)
if(n1 > n2){ z <- x[1:n2] + y}
else{z <- x + y[1:n1]}
z
}
myplus(1:10, 1:3)
[1] 2 4 6
```

Ex 4

- Here is a simple example of a program for calculating the real roots of a quadratic equation.
- # find the zeros of a2*x^2 + a1*x + a0 = 0
- # clear the workspace
- rm(list=ls())
- # input
- a2 <- 1
- a1 <- -4
- a0 <- 4
- # calculate the discriminant
- discrim <- a1^2 - 4*a2*a0

Ex 4

```
# calculate the roots depending on the value of the discriminant
if (discrim > 0) {
  roots <- c( (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2),
              (-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2) )
} else {
  if (discrim == 0) {
    roots <- -a1/(2*a2)
  } else {
    roots <- c()
  }
}
# output
show(roots)
```

switch

- The switch function has the following general form
- switch(object,
- "value1" = {expr1},
- "value2" = {expr2},
- "value3" = {expr3},
- {other expressions}
-)

switch

- If object has value value1 then expr1 is executed, if it has value2 then expr2 is executed and so on.
- If object has no match then other expressions is executed.
- Note that the block {other expressions} does not have to be present, the switch will return NULL in case object does not match any value.
- An expression expr1 in the above construction can consist of multiple statements. Each statement should be separated with a ; or on a separate line and surrounded by curly brackets.

Ex 1 convert unites to meters

```
convert2meters <- function (x,units=c('inches','feet','yards','miles')){
  units <- match.arg(units)
  switch (units,
          inches=x*0.0254,
          feet = x*0.3048,
          yards = x*0.9144,
          miles = x*1609.344)
}

convert2meters(5.8,'feet')
convert2meters(100,'miles')
```

Ex 2 Center of a vector

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
  }
x <- rcauchy(10)
centre(x, "mean")
[1] 0.8760325
> centre(x, "median")
[1] 0.5360891
> centre(x, "trimmed")
[1] 0.6086504
```

Control Flow
Tests with if and switch

- The for command has the following form, where x is a simple variable and vector is a vector.

```
for (x in vector) {
expression_1
...
}
```

for

- When executed, the for command executes the group of expressions within the braces { }, once for each element of vector.
- The grouped expressions can use x, which takes on each of the values of the elements of vector as the loop is repeated.

Ex 1 Calculating factorials for an integer x

- fac=function(x){
- f=1
- if (x<2) {f=1}
- else {for (i in 2:x) f=f*i
- }
- return (f)
- }

- fac(4)

Ex 2 *summing a vector*

- The following example uses a loop to sum the elements of a vector.
- Note that we use the function cat (for concatenate) to display the values of certain variables.
- The advantage of cat over show is that it allows us to combine text and variables together. The combination of characters \n (backslash-n) is used to 'print' a new line.

Ex 2 *summing a vector*

- (x_list <- seq(1, 9, by = 2))

- sum_x <- 0
- for (x in x_list) {
- sum_x <- sum_x + x
- cat("The current loop element is", x, "\n")
- cat("The cumulative total is", sum_x, "\n")
- }
- sum(x_list)

Pension Fund

- Here is an example for calculating the value of a pension fund under compounding interest.
- It uses the function floor(x), whose value is the largest integer smaller than x.

Pension Fund

- # clear the workspace
- rm(list=ls())
- # Inputs
- r <- 0.11 # Annual interest rate
- term <- 10 # Forecast duration (in years)
- period <- 1/12 # Time between payments (in years)
- payments <- 100 # Amount deposited each period

Pension Fund

- # Calculations
- n <- floor(term/period) # Number of payments
- pension <- 0
- for (i in 1:n) {
- pension[i+1] <- pension[i]*(1 + r*period) + payments
- }
- time <- (0:n)*period
- # Output
- plot(time, pension)

Looping with while

- Often we do not know beforehand how many times we need to go around a loop.
- That is, each time we go around the loop, we check some condition to see if we are done yet. In this situation we use a while loop, which has the form
- while (logical_expression) {
- expression_1
- ...
- }

Looping with while

- When a while command is executed, logical_expression is evaluated first.
- If it is TRUE then the group of expressions in braces { } is executed. Control is then passed back to the start of the command: if logical_expression is still TRUE then the grouped expressions are executed again, and so on.
- Clearly, for the loop to stop eventually, logical_expression must eventually be FALSE.
- To achieve this logical_expression usually depends on a variable that is altered within the grouped expressions.

Ex 1 simple example

```
tmp <- 0
 n <- 0
 while(tmp < 100){
   tmp <- tmp + rbinom(1,10,0.5)
   n <- n +1
 }
 cat("It took ",n," iterations to finish \n")
```

Ex 2 *Fibonacci numbers*

```
# calculate the first Fibonacci number greater than 100
# initialise variables
F <- c(1, 1) # list of Fibonacci numbers
n <- 2 # length of F
# iteratively calculate new Fibonacci numbers
while (F[n] <= 100) {
  # cat("n =", n, " F[n] =", F[n], "\n")
  n <- n + 1
  F[n] <- F[n-1] + F[n-2]
}
# output
cat("The first Fibonacci number > 100 is F(", n, ") =", F[n], "\n")
```