Functions and control flow statements

Paul M. Magwene

Functions

In computer programming, functions are the primary mechanism for organizing and abstracting sets of related computations.

Functions: Inputs and output

- Most functions take one or more input values – referred to as **arguments** to the function – and return one or more output values.
- Functions don't need to take inputs; their argument lists can be empty.
- Function don't necessarily return output values – some functions, such as those used to print information to the screen, are called for their "side effects".

When should I write a function?

- To enscapsulate a cohesive set of computations
- You recognize a series of computational steps that you repeat frequently
- As a means for making your code more “modular” and understandable.

Defining a function

The basic syntax for functions in R is:

```
function(arglist) {
  expressions # one or more expressions operating on fxn arguments
  return(value) # explicit return or last expression
}
```

Defining a function: Example

If we wanted to define our own function for calculating the mean of a vector of doubles we could write it as:

```
my.mean <- function(x) {
  sum(x)/length(x)
}

# test our function
height <- c(183, 168, 192, 175, 172, 180)
my.mean(height)
[1] 178.3333
```

In this example:

- `my.mean` is the name of our function
- `x` is the single argument that our function takes
- The return value (output) is the result of evaluating `sum(x)/length(x)`

Functions and scope I

The "scope" of a variable refers to the rules by which R looks up the values of a symbol.

```
area.of.circle <- function(r) {
  pi * r^2
}
```

In the above function, the scope of the variable `r` is the function body itself. `r` gets assigned the value of whatever input is passed to to the function. The scope of `r` is "local" to the function.

To illustrate this:

```
radii <- c(2,4,6,8)
other.radii <- c(1,3,5,7)
area.of.circle(radii)  # here r = radii
[1]  12.56637  50.26548 113.09734 201.06193
area.of.circle(other.radii) # here r = other.radii
[1]   3.141593  28.274334  78.539816 153.938040
```

Functions and scope II

Unlike many programming languages, if R encounters a variable within a function body that isn't in the list of arguments, or isn't defined in base (or other imported packages), it will look in the "global environment" to see if there is a variable with the same name.

```
bad.area.of.circle <- function() {
  pi * r^2
}

r <- 10  # perhaps unrelated to computation of interest
bad.area.of.circle()  # uses 'r' variable from global
[1] 314.1593
```

This local/global lookup can be useful (notice we didn't have to specify `base::pi` in our calculation) but can also be the source of errors in your programs. In general, you should try and write functions such that important variables always have local scope.

For more detailed discussion see the section on lexical scoping in Wickham's book "Advanced R".

Control flow

Code in a function, or in a code block, is evaluated in a top-to-bottom fashion. Control flow statements allow us to change the order of evaluation of code depending on criteria of interest.

Main control flow statements:

- If-else
- For loops
- While loops

For other control flow statements see the Bio 723 workbook.

If-else statements

`if` and `if-else` statements allow specific blocks of code to be evaluated if a logical condition is met.

General form of `if-else`:

```
if (Boolean expression) {
  Code to execute if
  Boolean expression is true
} else {
  Code to execute if
  Boolean expression is false
}
```

If-else example

The following function calculates the area of a rectangle. It takes two arguments, `l` and `w`. If the `w` argument is unspecified it assumes the rectangle of interest is a square (squares are a subset of rectangles) and calculates the area based on just the `l` value.

```
area.rectangle <- function(l, w=NA) {
  if (is.na(w)) {
    l * l
  } else {
    l * w
  }
}

# test the function with and w/out optional arguments
area.rectangle(10)
[1] 100
area.rectangle(10,5)
[1] 50
```

The else in the above example is optional if we put a `return()` call in the if-statement. Could re-write as:

```
area.of.rectangle <- function(l, w=NA) {
  if (is.na(w)) {
    return(l * l)
  }
  l * w
}
```

For loops

A for statement iterates over the elements of a sequence. Commonly used to carry out a calculation on each element of a sequence or to make a calculation that involves all the elements of a sequence.

General form:

```
for (elem in sequence) {
  Do some calculations or
  Evaluate one or more expressions
}
```

Example:

```
my.sum <- function(x) {
  sum <- 0
  for (i in x){    # i is a temporary variable that is equal to the
    sum = sum + i  # corresponding value of x in each iteration
  }
  sum
}

# test my.sum
one.to.ten <- 1:10
my.sum(one.to.ten)
[1] 55
```

While statement

A `while` statement is a looping statement that iterates as long as the condition statement it contains is true.

General form:

```
while(condition) {
  Evaluate one or more expressions
}
```

Example:

```
flips.until.head <- function() {
  # simulate coin flips, counting number of flips
  # before a head is observed
  ct = 1
  while (runif(1) < 0.5) {
    ct = ct + 1
  }
  ct
}

# test our function a couple of times
# output will vary because of `runif()`
flips.until.head()
[1] 1
flips.until.head()
[1] 1
```

Parallels between for-loops and sum and product notation in mathematics

Class participation. What are the mathematical formulae for the following statistical functions?

1. Mean
2. Variance
3. Standard deviation
4. Covariance
5. Correlation

In-class assignment: Write for-loop and "vectorized" implementations of each of the above statistical functions.