Control Flow and Iteration

Canhong Wen

Agenda

- Control flow (or alternatively, flow of control)
  - `if()`, `else if()`, `ifelse()`, `switch()`
- Iteration
  - `for()`, `while()`

Control flow

Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated

A *control flow statement* is a statement whose execution results in a choice being made as to which of two or more paths should be followed

Summary of the control flow tools in R:

- `if()`, `else if()`, `else`: standard conditionals
- `ifelse()`: conditional function that vectorizes nicely
- `switch()`: handy for deciding between several options

`if()` and `else`

Use `if()` and `else` to decide whether to evaluate one block of code or another, depending on a condition.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

```
x = 0.5
if (x >= 0) {
  x
} else {
  -x
}
```

```
## [1] 0.5
```

- Condition in `if()` needs to give *one* `TRUE` or `FALSE` value
- Note that the `else` statement is optional
- Single line actions don't need braces, i.e., could shorten above to

```
if (x >= 0) x else -x
```

if()

We can use `else if()` arbitrarily many times following an `if()` statement

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

```
x = -2

if (x^2 < 1) {
  x^2
} else if (x >= 1) {
  2*x-1
} else {
 -2*x-1
}
```

```
## [1] 3
```

- Each `elseif()` only gets considered if the conditions above it were not `TRUE`
- The `else` statement gets evaluated if none of the above conditions were `TRUE`
- Note again that the `else` statement is optional

Quick decision making

In the `ifelse()` function we specify a condition, then a value if the condition holds, and a value if the condition fails

```
ifelse(x > 0, x, -x)
```

```
## [1] 2
```

One advantage of `ifelse()` is that it vectorizes nicely.

```
x <- -2:2
ifelse(x > 0, x, -x)
```

```
## [1] 2 1 0 1 2
```

Deciding between many options

Instead of an `if()` statement followed by `elseif()` statements (and perhaps a final `else`), we can use `switch()`. We pass a variable to select on, then a value for each option

```
type.of.summary = "mode"

switch(type.of.summary,
       mean=mean(x.vec),
       median=median(x.vec),
       histogram=hist(x.vec),
       "I don't understand")
```

```
## [1] "I don't understand"
```

- Here we are expecting `type.of.summary` to be a string, either "mean", "median", or "histogram"; we specify what to do for each
- The last passed argument has no name, and it serves as the `else` clause
- Try changing `type.of.summary` above and see what happens

Reminder: Boolean operators

Remember our standard Boolean operators, `&` and `|`. These combine terms elementwise

```
u.vec = runif(10, -1, 1)
u.vec
```

```
##  [1] -0.18284222 -0.90189104 -0.02025565  0.25098637  0.45316209
##  [6]  0.52742369 -0.05074679  0.30431945  0.08022087  0.20981184
```

```
u.vec[-0.5 <= u.vec & u.vec <= 0.5] = 999
u.vec
```

```
##  [1] 999.0000000  -0.9018910 999.0000000 999.0000000 999.0000000
##  [6]   0.5274237 999.0000000 999.0000000 999.0000000 999.0000000
```

Flow control wants *one* Boolean value, and to skip calculating what's not needed.

Combining Booleans

In contrast to the standard Boolean operators, `&&` and `||` give *one* Boolean, lazily: meaning we terminate evaluating the expression ASAP

```
(0 > 0) && (all.equal(42%%6, 169%%13))
```

```
## [1] FALSE
```

```
all.equal(42%%6, 169%%13)
```

```
## [1] TRUE
```

- This *never* evaluates the complex expression on the right.
- In control flow, we typically just want one Boolean
- Use `&&` and `||` for control or conditionals, `&` and `|` for subsetting or indexing

Iteration

Computers: good at applying rigid rules over and over again. Humans: not so good at this. Iteration is at the heart of programming

Summary of the iteration methods in R:

- `for()`, `while()` loops: standard loop constructs
- Vectorization: use it whenever possible! Often faster and simpler
- `apply()` family of functions: alternative to `for()` loop, these are built-in R functions
- `**ply()` family of functions: another alternative, very useful, from the `plyr` package

`for()`

A `for()` loop increments a **counter** variable along a vector. It repeatedly runs a code block, called the **body** of the loop, with the counter set at its current value, until it runs through the vector

```
n = 10
log.vec = vector(length=n, mode="numeric")
for (i in 1:n) {
  log.vec[i] = log(i)
```

```
}
log.vec
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

Here `i` is the counter and the vector we are iterating over is `1:n`. The body is the code in between the braces.

- Note that there is a better way to do this job!

Breaking from the loop

We can **break** out of a `for()` loop early (before the counter has been iterated over the whole vector), using `break`

```
n = 10
log.vec = vector(length=n, mode="numeric")
for (i in 1:n) {
  if (log(i) > 2) {
    cat("I'm outta here. I don't like numbers bigger than 2\n")
    break
  }
  log.vec[i] = log(i)
}
```

```
## I'm outta here. I don't like numbers bigger than 2
```

```
log.vec
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 0.0000000 0.0000000 0.0000000
```

Variations on standard `for()` loops

Many different variations on standard `for()` are possible. Two common ones:

- Nonnumeric counters: counter variable always gets iterated over a vector, but it doesn't have to be numeric

```
for (str in c("Prof", "Canhong", "Wen")) {
  cat(paste(str, "declined to comment\n"))
}
```

```
## Prof declined to comment
## Canhong declined to comment
## Wen declined to comment
```

Variations on standard `for()` loops

- Nested loops: body of the `for()` loop can contain another `for()` loop (or several others)

```
for (i in 1:4) {
  for (j in 1:i^2) {
    cat(paste(j,""))
  }
```

```
  cat("\n")
}
```

```
## 1
## 1 2 3 4
## 1 2 3 4 5 6 7 8 9
## 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

A more complex nested iteration example

```
a <- matrix(1:4, nrow = 2)
b <- matrix(1:6, nrow = 2)
c <- matrix(0, nrow=nrow(a), ncol=ncol(b))
if (ncol(a) == nrow(b)) {
  for (i in 1:nrow(c)) {
    for (j in 1:ncol(c)) {
      for (k in 1:ncol(a)) {
        c[i,j] <- c[i,j] + a[i,k]*b[k,j]
      }
    }
  }
  print(c)
} else {
  stop("matrices a and b non-conformable")
}
```

`while()`

A `while()` loop repeatedly runs a code block, again called the **body**, until some condition is no longer true

```
i = 1
log.vec = c()
while (log(i) <= 2) {
  log.vec = c(log.vec, log(i))
  i = i+1
}
log.vec
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

- Condition in the argument to `while` must be a single Boolean value (like `if`)
- Body is looped over until the condition is `FALSE` so can loop forever
- Loop never begins unless the condition starts `TRUE`

`for()` vs. `while()`

- `for()` is better when the number of times to repeat (values to iterate over) is clear in advance
- `while()` is better when you can recognize when to stop once you're there, even if you can't guess it to begin with
- `while()` is more general, in that every `for()` could be replaced with a `while()` (but not vice versa)

`while(TRUE)` or `repeat`

`while(TRUE)` and `repeat`: both do the same thing, just repeat the body indefinitely, until something causes the flow to break. Example (try running in your console):

```
repeat {
  ans = readline("Who is the lecturer of Applied Statistical Software at USTC? ")
  if (ans == "Canhong Wen" || ans == "Teacher Wen" || ans == "Wen Canhong") {
    cat("Yes! You get an 'A'.")
    break
  }
  else {
    cat("Wrong answer!\n")
  }
}
```

Avoiding explicit iteration

- Warning: some people have a tendency to **overuse** `for()` and `while()` loops in R
- R has many ways of *avoiding* iteration, by acting on whole objects
  - It's conceptually clearer
  - It leads to simpler code
  - It's faster (sometimes a little, sometimes drastically)

Vectorized arithmetic

How many languages add 2 vectors:

```
c <- vector(length(a))
for (i in 1:length(a)) {  c[i] <- a[i] + b[i]  }
```

How R adds 2 vectors:

```
a+b
```

or a triple `for()` loop for matrix multiplication vs. `a %*% b`

Advantages of vectorizing

- Clarity: the syntax is about *what* we're doing
- Concision: we write less
- Abstraction: the syntax hides *how the computer does it*
- Generality: same syntax works for numbers, vectors, arrays, ...
- Speed: modifying big vectors over and over is slow in R; work gets done by optimized low-level code

Vectorized calculations

Many functions are set up to vectorize automatically

```
abs(-3:3)
```

```
## [1] 3 2 1 0 1 2 3
```

```r
log(1:7)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

See also `apply()`

Summary

- `if()`, `elseif()`, `else`: standard conditionals
- `ifelse()`: shortcut for using `if()` and `else` in combination
- `switch()`: shortcut for using `if()`, `elseif()`, and `else` in combination
- `for()`, `while()`, `repeat`: standard loop constructs
- Don't overuse explicit `for()` loops, vectorization is your friend!
- `apply()` and `**ply()`: can also be very useful (we've see them before)