Control Flow

James M. Flegal

Agenda

- Control flow (or alternatively, flow of control)
- if(), for(), and while()
- Avoiding iteration
- Introduction to strings and string operations

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

Conditionals

- Have the computer decide what to do next
  - Mathematically

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}, \quad \psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

  - Computationally

```
if the country code is not "US",
multiply prices by current exchange rate
```

if()

- Simplest conditional

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

- Condition in `if` needs to give *one* `TRUE` or `FALSE` value
- `else` clause is optional
- one-line actions don't need braces

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

if()

- `if` can *nested* arbitrarily deep

```
if (x^2 < 1) {
  x^2
} else {
  if (x >= 0) {
    2*x-1
  } else {
     -2*x-1
  }
}
```

- Can get ugly!

Combining Booleans

- `&` work `|` like `+` or `*` in that they combine terms element-wise
- Flow control wants *one* Boolean value, and to skip calculating what's not needed
- `&&` and `||` give *one* Boolean, lazily

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

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

- This *never* evaluates the complex expression on the right
- Use `&&` and `||` for control, `&` and `|` for subsetting

Iteration

- Repeat similar actions multiple times

```
table.of.logarithms <- vector(length=7,mode="numeric")
table.of.logarithms
```

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

```
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] <- log(i)
}
table.of.logarithms
```

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

for()

```
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] <- log(i)
}
```

- `for` increments a **counter** (here `i`) along a vector (here `1:length(table.of.logarithms)`) and **loops through** the **body** until it runs through the vector
- That is, it **iterates over** the vector
- Note, there is a better way to do this job!
- Can contain just about anything, including
  - if() clauses
  - other for() loops (nested iteration)

Nested iteration example

```
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]
      }
    }
  }
} else {
  stop("matrices a and b non-conformable")
}
```

while()

```
while (max(x) - 1 > 1e-06) {
  x <- sqrt(x)
}
```

- Condition in the argument to `while` must be a single Boolean value (like `if`)
- Body is looped over until the condition is `FALSE` (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
- Every `for()` could be replaced with a `while()`
  - Show this on your own!

Avoiding iteration

- `R` has many ways of *avoiding* iteration, by acting on whole objects
    - Conceptually clearer
    - Leads to simpler code
    - Faster (sometimes a little, sometimes drastically)

Vectorized arithmetic

- Many languages add 2 vectors using

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

- `R` adds 2 vectors using

```
a+b
```

- Triple `for()` loop for matrix multiplication vs. `a %*% b`

Advantages of vectorizing

- Clarity: syntax is about *what* we're doing
- Concision: write less
- Abstraction: 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
```

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

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.
```

- See also `apply()`

Vectorized conditions with `ifelse()`

```
ifelse(x^2 > 1, 2*abs(x)-1, x^2)
```

- 1st argument is a Boolean vector, then pick from the 2nd or 3rd vector arguments as `TRUE` or `FALSE`

What Is Truth?

- 0 counts as FALSE; other numeric values count as TRUE; the strings “TRUE” and “FALSE” count as you’d hope; most everything else gives an error
- Don’t play games here; try to make sure control expressions are getting Boolean values
- Conversely, in arithmetic, FALSE is 0 and TRUE is 1

```
library(datasets)
states <- data.frame(state.x77, abb=state.abb, region=state.region, division=state.division)
mean(states$Murder > 7)
```

```
## [1] 0.48
```

`switch()`

- Simplify nested `if` with `switch()`; give a variable to select on, then a value for each option

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

- Exercise on your own: Set `type.of.summary` to, successively, "mean", "median", "histogram", and "mode", and explain what happens

Unconditional iteration

```
repeat {
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
}
```

"Manual" control over iteration

```
repeat {
  if (watched) { next() }
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
  if (rescued) { break() }
}
```

- `break()` exits the loop; `next()` skips the rest of the body and goes back into the loop
- Both work with `for()` and `while()` as well
- Exercise: how would you replace `while()` with `repeat()`?

Strings and string operations

- Most data we deal with is in character form!
  - web pages can be scraped
  - email can be analyzed for network properties
  - survey responses must be processed and compared
- Even if you only care about numbers, it helps to be able to extract them from text and manipulate them easily.

Characters vs. Strings

- ***Character*** is a symbol in a written language, specifically what you can enter at a keyboard; letters, numerals, punctuation, space, newlines, etc.

`'L', 'i', 'n', 'c', 'o', 'l'`

- ***String*** is a sequence of characters bound together

`Lincoln`

- R does not have a separate type for characters and strings

```r
mode("L")
```

```
## [1] "character"
```

```r
mode("Lincoln")
```

```
## [1] "character"
```

```r
class("Lincoln")
```

```
## [1] "character"
```

Making Strings

- Use single or double quotes to construct a string; use `nchar()` to get the length of a single string. Why do we prefer double quotes?

```
"Lincoln"
```

```
## [1] "Lincoln"
```

```
"Abraham Lincoln"
```

```
## [1] "Abraham Lincoln"
```

```
"Abraham Lincoln's Hat"
```

```
## [1] "Abraham Lincoln's Hat"
```

```
"As Lincoln never said, \"Four score and seven beers ago\""
```

```
## [1] "As Lincoln never said, \"Four score and seven beers ago\""
```

Whitespace

- The space, `" "` is a character; so are multiple spaces `"     "` and the empty string, `""`.
- Some characters are special, so we have "escape characters" to specify them in strings.
  - quotes within strings: `\"`
  - tab: `\t`
  - new line: `\n` (use this when possible)
  - carriage return `\r`

Character data type

- One of the atomic data types, like `numeric` or `logical`
- Can go into scalars, vectors, arrays, lists, or be the type of a column in a data frame.

```
length("Abraham Lincoln's beard")
```

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

```
length(c("Abraham", "Lincoln's", "beard"))
```

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

```
nchar("Abraham Lincoln's beard")
```

```
## [1] 23
```

```
nchar(c("Abraham", "Lincoln's", "beard"))
```

```
## [1] 7 9 5
```

Character-valued variables

- They work just like others, e.g., with vectors

```
president <- "Lincoln"
nchar(president)  # NOT 9
```

```
## [1] 7
```

```
presidents <- c("Fillmore","Pierce","Buchanan","Davis","Johnson")
presidents[3]
```

```
## [1] "Buchanan"
```

```
presidents[-(1:3)]
```

```
## [1] "Davis"   "Johnson"
```

Displaying characters

- Know `print()`, of course; `cat()` writes the string directly to the console. If you're debugging, `message()` is preferred syntax in R.

```
print("Abraham Lincoln")
```

```
## [1] "Abraham Lincoln"
```

```
cat("Abraham Lincoln")
```

```
## Abraham Lincoln
```

```
cat(presidents)
```

```
## Fillmore Pierce Buchanan Davis Johnson
```

```
message(presidents)
```

```
## FillmorePierceBuchananDavisJohnson
```

Substring operations

- ***Substring*** is a smaller string from the big string, but still a string in its own right.
- A string is not a vector or a list, so we ***cannot*** use subscripts like `[[ ]]` or `[ ]` to extract substrings; we use `substr()` instead.

```
phrase <- "Christmas Bonus"
substr (phrase, start=8, stop=12)
```

```
## [1] "as Bo"
```

- Can also use `substr` to replace elements

```
substr(phrase, 13, 13) <- "g"
phrase
```

```
## [1] "Christmas Bogus"
```

substr() for string vectors

- `substr()` vectorizes over all its arguments

```
presents
```

```
## [1] "Fillmore" "Pierce"   "Buchanan" "Davis"    "Johnson"
```

```
substr(presidents,1,2)   # First two characters
```

```
## [1] "Fi" "Pi" "Bu" "Da" "Jo"
```

```
substr(presidents,nchar(presidents)-1,nchar(presidents))   # Last two
```

```
## [1] "re" "ce" "an" "is" "on"
```

```
substr(presidents,20,21)    # No such substrings so return the null string
```

```
## [1] "" "" "" "" ""
```

```
substr(presidents,7,7)      # Explain!
```

```
## [1] "r" ""  "a" ""  "n"
```

Dividing strings into vectors

- `strsplit()` divides a string according to key characters, by splitting each element of the character vector `x` at appearances of the pattern `split`.

```
scarborough.fair <- "parsley, sage, rosemary, thyme"
strsplit (scarborough.fair, ",")
```

```
## [[1]]
## [1] "parsley"   " sage"     " rosemary" " thyme"
```

```
strsplit (scarborough.fair, ", ")
```

```
## [[1]]
## [1] "parsley"  "sage"     "rosemary" "thyme"
```

- Pattern is recycled over elements of the input vector

```
strsplit (c(scarborough.fair, "Garfunkel, Oates", "Clement, McKenzie"), ", ")
```

```
## [[1]]
## [1] "parsley"  "sage"     "rosemary" "thyme"
##
## [[2]]
## [1] "Garfunkel" "Oates"
##
## [[3]]
## [1] "Clement"  "McKenzie"
```

Combining vectors into strings

- Converting one variable type to another is called ***casting***

```r
as.character(7.2)           # Obvious
```

```
## [1] "7.2"
```

```r
as.character(7.2e12)        # Obvious
```

```
## [1] "7.2e+12"
```

```r
as.character(c(7.2,7.2e12))  # Obvious
```

```
## [1] "7.2"     "7.2e+12"
```

```r
as.character(7.2e5)         # Not quite so obvious
```

```
## [1] "720000"
```

Building strings from multiple parts

- The `paste()` function is very flexible!
- With one vector argument, works like `as.character()`

```
paste(41:45)
```

```
## [1] "41" "42" "43" "44" "45"
```

Building strings from multiple parts

- With 2 or more vector arguments, combines them with recycling

```
paste(presidents,41:45)
```

```
## [1] "Fillmore 41" "Pierce 42"   "Buchanan 43" "Davis 44"    "Johnson 45"
```

```
paste(presidents,c("R","D"))  # Not historically accurate!
```

```
## [1] "Fillmore R" "Pierce D"   "Buchanan R" "Davis D"    "Johnson R"
```

```
paste(presidents,"(",c("R","D"),41:45,")")
```

```
## [1] "Fillmore ( R 41 )" "Pierce ( D 42 )"   "Buchanan ( R 43 )"
## [4] "Davis ( D 44 )"    "Johnson ( R 45 )"
```

Building strings from multiple parts

- Changing the separator between pasted-together terms

```
paste(presidents, " (", 41:45, ")", sep="_")
```

```
## [1] "Fillmore_ (_41_)" "Pierce_ (_42_)"   "Buchanan_ (_43_)" "Davis_ (_44_)"
## [5] "Johnson_ (_45_)"
```

```
paste(presidents, " (", 41:45, ")", sep="")
```

```
## [1] "Fillmore (41)" "Pierce (42)"   "Buchanan (43)" "Davis (44)"
## [5] "Johnson (45)"
```

- What happens if you give sep a vector?

More complicated example of recycling

- Exercise: Convince yourself of why this works as it does

```
paste(c("HW","Lab"),rep(1:11,times=rep(2,11)))
```

```
##  [1] "HW 1"   "Lab 1"  "HW 2"   "Lab 2"  "HW 3"   "Lab 3"  "HW 4"   "Lab 4"
##  [9] "HW 5"   "Lab 5"  "HW 6"   "Lab 6"  "HW 7"   "Lab 7"  "HW 8"   "Lab 8"
## [17] "HW 9"   "Lab 9"  "HW 10"  "Lab 10" "HW 11"  "Lab 11"
```

Condensing multiple strings

- Producing one big string

```
paste(presidents, " (", 41:45, ")", sep="", collapse="; ")
```

```
## [1] "Fillmore (41); Pierce (42); Buchanan (43); Davis (44); Johnson (45)"
```

- Default value of `collapse` is `NULL` – that is, it won't use it

Function for writing regression formulas

- R has a standard syntax for models: outcome and predictors.

```
my.formula <- function(dep,indeps,df) {
  rhs <- paste(colnames(df)[indeps], collapse="+")
  return(paste(colnames(df)[dep], " ~  ", rhs, collapse=""))
}
my.formula(2,c(3,5,7),df=state.x77)
```

```
## [1] "Income  ~   Illiteracy+Murder+Frost"
```

General search

- Use `grep()` to find which strings have a matching search term
- Reconstituting, make one long string, then split the words
- Counting words with `table()`
- Need to learn how to work with text patterns and not just constants
- Searching for text patterns using regular expressions

Summary

- `if`, nested `if`, `switch`
- Iteration with `for` and `while`
- Avoiding iteration with whole-object ("vectorized") operations
- Text is data, just like everything else