

Lecture 5: Flow control

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

Statements in R

- separated by semicolons or newlines
- grouped by curly braces '{' and '}' into blocks

semicolons indicate the end of a statement

newlines not necessarily

Whenever R encounters a syntactically correct statement it executes it and a value is returned

The value of a block is the value of the last statement

if statements

Allow conditional execution of statements

```
if( condition ) {  
    statement_block1 # executed if condition is true  
} else { # else is optional  
    statement_block2  
}
```

The value of condition is coerced to logical

- If integer or numeric, 0 is FALSE , rest are true
- Using other modes isn't really recommended

If the value has length more than one, only the first is used

Since `else` is optional, don't put it on its own line!

Can dispense with braces for one-line statements:

```
if(condition) statement1    else statement2
```

if/else statements can be nested:

```
if( condition1 ) {  
    statements1  
} else if( condition2 ) {  
    statements2  
} else {  
    statements3  
}
```

```
In [ ]: p <- rexp(1)
        if( p > 0 ) {
          p_logp <- p * log(p)
        } else {
          p_logp <- 0 # Assuming p >= 0
        }
```

```
In [ ]: if( p > 0 ) p_logp <- p * log(p) else p_logp <- 0
```

if is a function that returns values, so we can also write

```
In [ ]: p_logp <- if( p > 0 ) p * log(p) else 0
```

```
In [ ]: # Less clear:
        p_logp <- 'if'( p > 0, {p * log(p)}, 0)
```

Logical operators

`!` : logical negation

`&` and `&&`: logical 'and' | `|` and `||` : logical 'or' `&` and `|` perform elementwise comparisons on vectors

`&&` and `||` :

- evaluate from left to right
- look at first element of each vector
- evaluation proceeds only until the result is determined
- used inside if conditions

Also useful are `xor()`, `any()`, `all()`

```
In [ ]: c(TRUE, TRUE) & c(TRUE, FALSE)
```

```
In [ ]: c(TRUE, TRUE) && c(TRUE, FALSE)
```

```
In [ ]: c(TRUE, FALSE) | NA
```

```
In [ ]: TRUE && (pi > 1) && {print("Hello"); TRUE}
```

```
In [ ]: TRUE && (pi == 3.14) && {print("Hello"); TRUE}
```

```
In [ ]: c(TRUE, TRUE) & c(TRUE, FALSE) & {print("Hello!"); TRUE}
```

```
In [ ]: c(TRUE, TRUE) & c(FALSE, FALSE) & {print("Hello!"); TRUE}
```

We will look at *lazy* evaluation later

Explicit looping: `for()`, `while()` and `repeat()`

```
for(elem in vect) { # Can be vector or list over  
  Do_stuff_with_elem # successive elements of vect  
}
```

```
In [ ]: x <- 0 # Horrible  
        for(ii in 1:50000) x <- x + log(ii)
```

```
In [ ]: x <- sum(log(1:50000)) # Much more simple and efficient!
```

```
In [ ]: system.time({x<-0; for(i in 1:50000) x <- x + log(i)})
```

```
In [ ]: system.time( x <- sum(log(1:50000)) )
```


An aside on increasing vector lengths

```
In [ ]: system.time({x<-0; for(i in 1:10000) x[i] <- i})  
mean(x)
```

```
In [ ]: system.time({x<-rep(0,10000); for(i in 1:10000) x[i] <- i })  
mean(x)
```

Vectorization

Vectorization allows concise and fast loop-free code

Example: Entropy $H(p) = - \sum_{i=1}^{|p|} p_i \log p_i$ of a prob. distrib.

```
In [ ]: H <- -sum( p * log(p) ) # Vectorized but wrong (p[i] == 0?)
```

```
In [ ]: H <- 0
# Correct but slow
for(i in 1:length(p))
  if(p[i] > 0) H <- H - p[i] * log(p[i])
pos <- p > 0
```

```
In [ ]: H <- - sum( p[pos] * log(p[pos]) )
```

Vectorization isn't always possible though

- when contents of the loop are complicated
- when future iterations depend on the past
- sometimes the cost in human-time of complicated vectorization isn't worth the saved CPU cycles

See the third and fourth Circles in *The R Inferno*, Patrick Burns

Vectorization via `ifelse()`

`ifelse()` has syntax:

```
ifelse(bool_vec, true_vec, false_vec)
```

Returns a vector of length equal to `bool_vec` whose

- i^{th} element is `true_vec[i]` if `bool_vec[i]` is TRUE
- i^{th} element is `false_vec[i]` if `bool_vec[i]` is FALSE
- `true_vec` and `false_vec` are recycled if necessary

Entropy revisited:

```
In [ ]: H <- -sum(ifelse( p > 0, p * log(p), 0 ))
```

`ifelse()` has syntax:

```
ifelse(bool_vec, true_vec, false_vec)
```

`ifelse` is not lazy, usually evaluates all `true_vec` and `false_vec` (unless `bool_vec` is all TRUE or FALSE)

```
In [ ]: x <- c(6:-4)
        sqrt(x) #- gives warning
        sqrt(ifelse(x >= 0, x, NA)) # no warning
```

```
In [ ]: ## Note: the following also gives the warning !
        ifelse(x >= 0, sqrt(x), NA)
```

I prefer to subset vectors

While loops

```
while( condition ) {  
    stuff # Repeat while condition evaluates to TRUE  
}
```

If stuff doesn't affect condition , we loop forever.

Then, we need a break statement. Useful if many conditions

```
while(TRUE) { # Or use 'repeat { ... }'  
    stuff1  
    if( condition1 ) break  
    stuff2  
    if( condition2 ) break  
}
```

```
In [ ]: i <- 4
        while( i > 0 ) {
          print(i)
          i <- i - 1
        }
```

```
In [ ]: i <- 5
        while( i <- i - 1 ) { # while condition has a 'side effect'
          print(i)           # Not recommended
        }
```

```
In [ ]: i <- 4
        while( { print(i); i <- i - 1 } ) {}
        # Correct but ridiculous
```

Might be useful if the block is a function

`break()`, `next()` and `switch()`

`break ()` transfers control to first statement outside loop

`next ()` halts current iteration and advances looping index

Both these commands apply to the innermost loop

Useful to avoid writing up complicated conditions

`switch ()` is another potentially useful alternative to `if`

See documentation (I don't use it much)

The *apply family

Useful functions for repeated operations on vectors, lists etc.

Note (Circle 4 of *The R inferno*): • These are not vectorized operations but are loop-hiding • Cleaner code, but comparable speeds to explicit for loops

```
# Calc. mean of each element of my_list  
rslt_list <- lapply(my_list, FUN = mean)
```

Stackexchange has a nice summary (<http://stackoverflow.com/questions/3505701/r-grouping-functions-sapply-vs-lapply-vs-apply-vs-tapply-vs-by-vs-aggrega>)

The plyr package (discussed later) is nicer