Review `if/else`

The basic syntax for an if/else statement is

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

Multiple expressions can be grouped together in the curly braces. A group of expressions is called a *block*.

Here, the word *statement* refers to either a single expression or a block.

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

First, **`condition`** is evaluated. If the result is `TRUE` then **`statement1(s)`** is evaluated. If the result is `FALSE` then **`statement2(s)`** is evaluated.

- If the result has multiple elements, only the first element is checked
- If the result is numeric, `0` is treated as `FALSE` and any other number as `TRUE`.
- All other types give an error
- If the result is `NA`, you will get an error.

The else clause is optional.

```
if( any(x <= 0) ) {
  x = 1+x
}
y = log(x)
```

The result of an if/else statement can be assigned. For example,

```
if ( any(x <= 0) ) {
  y = log(1+x)
} else {
  y = log(x)
}
```

is the same as

```
y = if ( any(x <= 0) ) {
  log(1+x)
} else {
  log(x)
}
```

If/else statements can be nested.

```
if (condition1 ) {
  statement1
} else {
  if (condition2) {
    statement2
  } else {
    if (condition3) {
      statement3
    } else {
      statement4
    }
  }
}
```

Simplified version of nested If/else statements

```
if (condition1 ) {
  statement1
  } else if (condition2) {
    statement2
    } else if (condition3) {
      statement3
      } else {
        statement4
        }
```

When If/else statements are nested.

The conditions are evaluated, in order, until one evaluates to TRUE. Then the associated statement/block is evaluated. The statement in the final else clause is evaluated if none of the conditions evaluates to TRUE.

Common uses of if/else

1. With logical arguments to tell a function what to do

```
myMedian = function(x, hi = NULL){
  if (is.null(hi)) {
    median(x)
  } else {
    median(x[order][-1])
}
```

2. To verify that the arguments of a function are as expected

```
if ( !is.matrix(m) ) {
  stop("m must be a matrix")
}
```

3. Handle errors

```
ratio =
  if (x != 0) {
    y/x
  } else {
    NA
  }
```

OR

```
ratio = if (x !=0) y/x else NA
```

Be careful when you don't use curly braces.

Debugging Strategies

- Write code in a plain text file, e.g. in a script in Rstudio
- source() code into R (do not copy and paste)
- Syntax error will be caught and line number given
- Line numbers may not locate the error exactly
- Sometimes the error occurred earlier, but it gives you a starting place

Here is a function that locates the end points of the whiskers in a box plot:

```
whisker.endpoints = function(x) {
  if (is.na(x) warning("Careful x has NAs")
  qlu = quantile(x, probs = c(0.25, 0.75),
                        na.rm = FALSE)
  iqr = IQR(x, na.rm = FALSE)
  return(x - 1.5 * iqr, x + 1.5 * iqr)
}
```

Let’s Debug this Function

Types of Errors

- Syntax Errors –
  - Parsing error
- Abnormal Termination
  - Function call results in an error
- Warning Message
  - Function call results in a warning
- Silent Mistakes
  - Only Good tests reveal these errors

Some debugging strategies

The `traceback` function prints the sequence of calls that led to the last error. This can show you where in your function something is going wrong.

It may not even be in the function itself, but in another function that is being called within the original function.

```
cv = function(x) sd(x/mean(x))
> cv(0)
Error in var(x, na.rm = na.rm) : missing observations in
cov/cor
> traceback()
3: var(x, na.rm = na.rm)
2: sd(x/mean(x))
1: cv(0)
```

If you have some idea where the error is occurring, place a call to the browser() function in your code near (but before) where the error is occurring.

In the browser, you can run any valid R expression

- You can use check that key variables are what you think they are.
- You can run code to fix something

You can step through the following lines of code with

'n' (or just return) - Advance to the next step.

'c' - continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.

'Q' - exit the browser and the current evaluation and return to the top-level prompt.

You can automatically be placed in browser mode, if you set an option: `options(error = recover)`
To turn off this option: `options(error = NULL)`

If you have some idea where the error is occurring, you can use `print` to check that key variables are what you think they are.

Consider “commenting out” lines of your code where the error might occur, then adding them back in one by one.

Code that’s Hard to Read

```
disobey = function(x,epsilon){
if(any(x<=0)){x[x<=0]=epsilon}
for(i in x){g(i)}}
```

This code is hard to read which makes it hard to debug

Code that's Easy to Read

```
obey = function(x, epsilon) {
  if(any(x <= 0)) {
    x[x <= 0] = epsilon
  }
  for (i in x) {
    cat(i,"\n")
  }
}
```

This code follows simple guidelines for spacing and indentation that makes it easy to read and easy to debug

Style Rules

Adapted from Google's https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml

Guidelines

Variable names –

- Use meaningful variable names

- Prefer all lower case, except `varName` is OK, although Google prefers `var.name` and others use `var_name`

- Make function names verbs

Guidelines

- Line length – max 60 to 80 characters
- Indentation
    - Use 2 spaces (or tab) for each code block
    - Do not mix tabs and spaces
- Spacing
    - Put space after comma,
    - Before and after infix ops, e.g. +, -, *
    - Before left (except in a function call

Guidelines

- Curly braces
  - Opening { not on own line
  - Closing } on own line
  - May omit { } when code block contains only one line, but be consistent
- Else –
  - always use

```
} else {
```

Guidelines

- Semicolons – Never Use
- Function definition and calls
  - First list arguments without defaults
  - Break lines after ,

- Function documentation
  - Comment your code
  - Top of function: What function does, what are inputs, and output

Catching Errors

Catching errors

1. The function `stop` stops execution of the current expression and prints a specified error message.

```
showstop = function(x){
  if(any(x < 0)) stop("x must be >= 0")
  return("ok")
}
```

```
> showstop(1:5)
[1] "ok"

> showstop(c(-1, 1))
Error in showstop(c(-1, 1)) : x must be >= 0
```

2. A similar function is `stopifnot`. It has the advantage of being able to take multiple conditions.

```
showstopifnot = function(x){
  stopifnot(x >= 0, x %% 2 == 1)
  return("ok")
}
```

```
> showstopifnot(1)
[1] "ok"

> showstopifnot(c(1, -1))
Error: all(x >= 0) is not TRUE

> showstopifnot(c(1,2))
Error: x %%2 == 1 is not all TRUE
```

3. The `warning` function just prints a warning message without stopping the execution of the function.

```
ratio.warn = function(x, y){
  if(any(y == 0))
    warning("Dividing by zero")
  return(x/y)
}
```

```
> ratio.warn(x = 1, y = c(1, 0))
[1]   1 Inf
Warning message:
In ratio.warn(x = 1, y = c(1, 0)) : Dividing by zero

> ratio.warn(x = 1:3, y = 1:2)
[1] 1 1 3
Warning message:
In x/y : longer object length is not a multiple of shorter
object length
```

4. Finally, `try` allows you to try the code. If it doesn't produce an error then you proceed. If it does, you decide what to do

```
ratio.try = function(x, y){
  z = try(x/y, silent = TRUE)
  if(inherits(z, "try-error")) {
     warning("Division problem")
     z = NULL
  }
  return(z)
}

> ratio.try(x = 1, y = c(1, 0))
[1]   1 Inf

> ratio.try(x = 1, y = "r")
NULL
Warning: Division problem
```

Efficient programming

The first rule of efficient programming in R is to make use of vectorized calculations and the apply mechanisms whenever possible.

You can check how much time it takes to evaluate any expression by wrapping it in `system.time()`. Units are in seconds.

```
> system.time(normal.samples = rnorm(1000000))
   user  system elapsed 
  0.196   0.013   0.221 
```

wall clock time

CPU time for R process

CPU time for system on behalf of R

A systematic way to time every part of a function is to use the `Rprof` and `summaryRprof` functions. This can be a handy way to find bottlenecks. The general syntax looks like this:

```
Rprof("profiling")
statements
Rprof(NULL)
summaryRprof("profiling")
```

File to store information

Whatever you want to time