Control Flow

The Median

Calculate the median for a vector of numbers

```
1 0 0 5 3
```

To find the median, we sort the values:

```
0 0 1 3 5
```

Select the middle one:

```
0 0 1 3 5
```

Code it up:

Calculate the median for a vector of numbers

```
x = c(1, 0, 0, 5, 3)
```

To find the median, we sort the values:

```
sort(x)
```

Select the middle one:

```
sort(x)[3]
```

Encapsulate into a function:

```
x = c(1, 0, 0, 5, 3)

myMedian = function(x) {
  sort(x)[3]
}
```

Does this work?

Encapsulate into a function:

We need to make it work for vectors of other lengths

```
myMedian = function(x) {
  sort(x)[length(x)/2]
}
```

Test:

```
> x = c(1, 0, 0, 5, 3)

> myMedian(x)
[1] 0
```

What went wrong?

Revise our function

```
myMedian = function(x) {
  sort(x)[(length(x) + 1)/2]
}
```

OR

```
myMedian = function(x) {
  n = length(x)
  sort(x)[(n + 1)/2]
}
```

A bit easier to read if we create a variable for length(x)

Test:

```
> x = c(1, 0, 0, 5, 3)
> myMedian(x)
[1] 1
```

Another test

```
> y = c(1, 0, 1, 5, 3, 20)
> myMedian(y)
[1] 1
```

Is This Correct?

Revise our function

When `n` is even we compute the median differently: we average the 2 middle values

ODD: We want to choose the `(n+1)/2` largest element when `n` is odd
EVEN: We want to average the `n/2` and `n/2 + 1` largest elements when even

Conditional Evaluation of Code

ODD: We want to choose the `(n+1)/2` largest element when `n` is odd

```
sort(x)[(n + 1)/2]
```

EVEN: We want to average the `n/2` and `n/2 + 1` largest elements when even

```
mean(sort(x)[c(n/2, n/2 + 1)])
```

Control Flow

We need a logical expression that evaluates TRUE when `n` is odd

```
if (n is odd) {
  sort(x)[(n + 1)/2]
} else {
  mean(sort(x)[c(n/2, n/2 + 1)])
}
```

`n %% 2` returns 1 if `n` is odd and 0 if even

Revise our function

```
myMedian = function(x) {
  n = length(x)
  odd = as.logical(n %% 2)

  if (odd) {
    sort(x)[(n + 1)/2]
  } else {
    mean(sort(x)[c(n/2, n/2 + 1)])
  }
}
```

Another Version

```
myMedian = function(x) {
  n = length(x)
  odd = as.logical(n %% 2)

  if (odd) {
    m = sort(x)[(n + 1)/2]
  } else {
    m = mean(sort(x)[c(n/2, n/2 + 1)])
  }
  return(m)
}
```

Alternative

```
myMedian = function(x) {
  n = length(x)
  odd = as.logical(n %% 2)

  if (odd) {
    return(sort(x)[(n + 1)/2])
  } else {
   return(mean(sort(x)[c(n/2, n/2 + 1)]))
  }
}
```

Follow the Flow

x = c(1, 0, 0, 5, 3)
myMedian(x)

myMedian = function(x) {									
n = length(x)									
odd = as.logical(n %% 2)									
if (odd) {									
m = sort(x)[(n+1)/2]									
} else {									
m = mean(sort(x)[c(n/2, n/2 + 1)])									
}									
return(m)									
}									

x = c(1, 0, 0, 5, 3)
myMedian(x)

myMedian = function(x) {									
n = length(x)	x								
odd = as.logical(n %% 2)									
if (odd) {									
m = sort(x)[(n+1)/2]									
} else {									
m = mean(sort(x)[c(n/2, n/2 + 1)])									
}									
return(m)									
}									

x = c(1, 0, 0, 5, 3)
myMedian(x)

myMedian = function(x) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {			x						
m = sort(x)[(n+1)/2]				x					
} else {									
m = mean(sort(x)[c(n/2, n/2 + 1)])									
}									
return(m)									
}									

x = c(1, 0, 0, 5, 3)
myMedian(x)

myMedian = function(x) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {			x						
m = sort(x)[(n+1)/2]				x					
} else {									
m = mean(sort(x)[c(n/2, n/2 + 1)])									
}									
return(m)					x				
}									

y = c(1, 0, 1, 5, 3, 20)
myMedian(y)

A. Correct
B. Wrong

myMedian = function(x) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {									
m = sort(x)[(n+1)/2]									
} else {									
m = mean(sort(x)[c(n/2, n/2 + 1)])			x						
}									
return(m)				x					
}									

More on the median

The median need not be uniquely determined when n is even

```
0  1  1  3  5  20
```

Formal Definition: Any value such that at least half the numbers are at or below and at least have the numbers are at or above. The values 1 and 3, and any value in between can be considered a median

Revise our Function

Add a parameter to our function; call it `hi`

```
0  1  1  3  5  20
```

if `hi` is TRUE choose 3
if `hi` is FALSE choose 1
if `hi` is NULL choose 2 (the conventional).

Final Version

```
myMedian = function(x, hi = NULL) {
  n = length(x)
  odd = as.logical(n %% 2)

  if (odd) {
    return(sort(x)[(n + 1)/2])
  } else if(is.null(hi)) {
    return(mean(sort(x)[c(n/2, n/2 + 1)]))
    }  else if(hi) {
       return(sort(x)[n/2 + 1])
       } else return(sort(x)[n/2])
}
```

y = c(1, 0, 1, 5, 3, 20)
myMedian(y, TRUE)

myMedian = function(x, hi = NULL) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {			x						
return(sort(x)[(n+1)/2])									
} else if (is.null(hi)) {				x					
return(mean(sort(x)[c(n/2, n/2 + 1)]))									
} else if (hi) {					x				
return(sort(x)[n/2 + 1]						x			
} else {									
return(sort(x)[n/2]									
}									
}									

y = c(1, 0, 1, 5, 3, 20)
myMedian(y, FALSE)

A.Correct
B.Wrong

myMedian = function(x, hi = NULL) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {			x						
return(sort(x)[(n+1)/2])									
} else if (is.null(hi)) {				x					
return(mean(sort(x)[c(n/2, n/2 + 1)]))									
} else if (hi) {									
return(sort(x)[n/2 + 1]									
} else {									
return(sort(x)[n/2] }					x				
}									
}									

y = c(1, 0, 1, 5, 3, 20)
myMedian(y)

A.Correct
B.Wrong

myMedian = function(x, hi = NULL) {									
n = length(x)	x								
odd = as.logical(n %% 2)		x							
if (odd) {			x						
return(sort(x)[(n+1)/2])									
} else if (is.null(hi)) {				x					
return(mean(sort(x)[c(n/2, n/2 + 1)]))					x				
} else if (hi) {						x			
return(sort(x)[n/2 + 1]									
} else {									
return(sort(x)[n/2]									
}									
}									

Control Flow Recap

Control Flow structures allow us to control which statements are evaluated and in what order.

In R the primary ones consist of

- `if/else` statements and
- `ifelse()` function
- `for` and `while` loops

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 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)
}
```

When we discussed Boolean algebra before, we met the operators **`&`** (AND) and **`|`** (OR).

Recall that these are *vectorized* operators.

If/else statements, on the other hand, are based on a single, "global" condition. So we often see constructions using `any` or `all` to express something related to the whole vector, like

```
if ( any(x < -1 | x > 1) ) {
  warning("Value(s) in x outside the
interval [-1,1]")
}
```

(We'll discuss error handling more later.)

Also, the else clause is optional. Another way to do the above is

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

Note that this version this changes `x` as well.

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.

Some common uses of if/else clauses

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])
}
```

Some common uses of if/else clauses

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

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

3. To handle common numerical errors

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

This can be more compactly written because each block consists of only one statement

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

Be careful when you don't use curly braces.

A note about formatting if/else statements:

When the if statement is not in a block, the else (if present) must appear on the same line as `statement1` or immediately following the closing brace. For example,

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

will be an error if not part of a larger block and/or function. I strongly suggest using the format

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

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 – maximum 80 characters
- Indentation
  - Use 2 spaces for each sub-block (or tab)
  - Do not mix tabs and spaces
- Spacing
  - Put space after comma,
  - Before and after infix ops
  - Before left (except in a function call

Guidelines

- Curly braces
  - Opening { not on own line
  - Closing } on own line
  - May omit { } when code is 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