Week 4: Conditionals & Testing

EMSE 6574, Section 11

John Helveston
September 16, 2019

Announcements

1) You can change the way RStudio looks

2) Quiz 1 (getting back at end of class)

3) HW1 update

"Flow Control"

Flow control is code that alters the otherwise linear flow of operations in a program.

This week:

- `if` statements
- `else` statements

Next week:

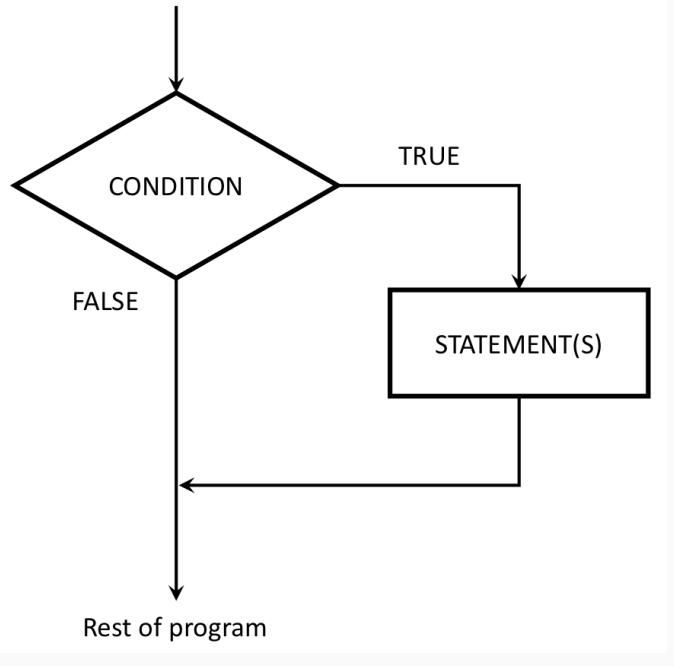
- `for` loops
- `while` loops
- `break` statements
- `next` statements

The `if` statement

Basic format:

```
if ( CONDITION ) {
    STATEMENT1
    STATEMENT2
    ETC
}
```

Here's the general idea:



Practice: What will this return?

60 seconds - no typing!

```
f <- function(x) {
    cat("A")
    if (x == 0) {
        cat("B")
        cat("C")
    }
    cat("D")
}
f(1)
f(0)
```

```
f(1)
```

```
## AD
```

```
f(0)
```

```
## ABCD
```

Example: Absolute value

Write the function `absValue()` that returns the absolute value of a number.

```
absValue <- function(x) {
    if (x < 0) {x = -1*x}
    return(x)
}
```

```
absValue(7)  # Returns 7
```

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

```
absValue(-7) # Also returns 7
```

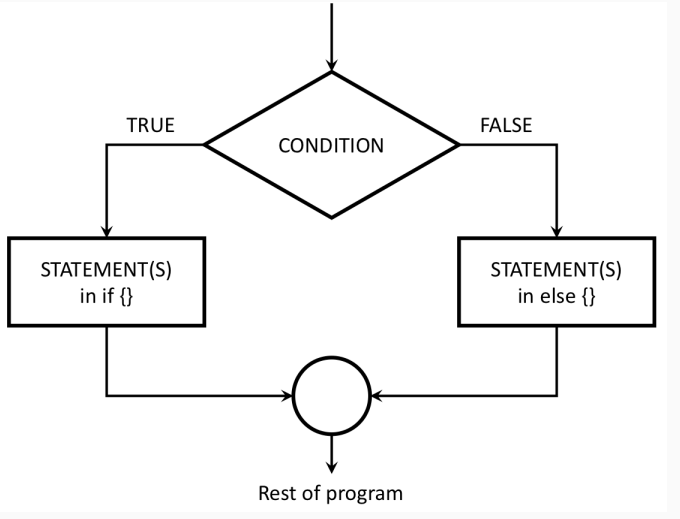
```
## [1] 7
```

Adding an `else` to an `if`

Basic format:

```
if ( CONDITION ) {
  STATEMENT1
  STATEMENT2
  ETC
} else {
  STATEMENT3
  STATEMENT4
  ETC
}
```

Here's the general idea:



Practice: What will this return?

2 minutes - no typing!

```
f <- function(x) {
    cat("A")
    if (x == 0) {
        cat("B")
        cat("C")
    } else {
        cat("D")
        if (x == 1) {
            cat("E")
        } else {
            cat("F")
        }
    }
    cat("G")
}
f(0)
f(1)
f(2)
```

```
f(0)
```

```
## ABCG
```

```
f(1)
```

```
## ADEG
```

```
f(2)
```

```
## ADFG
```

`else if` chains

Often times you'll need to check for more than one condition.

"Bracketing" problems (like setting grades) are a good example.

```
getLetterGrade <- function(score) {
    if (score >= 90) {
        grade = "A"
    } else if (score >= 80) {
        grade = "B"
    } else if (score >= 70) {
        grade = "C"
    } else if (score >= 60) {
        grade = "D"
    } else {
        grade = "F"
    }
    return(grade)
}
```

```
cat(" 99 -->", getLetterGrade(99))
```

```
##  99 --> A
```

```
cat(" 88 -->", getLetterGrade(88))
```

```
##  88 --> B
```

```
cat(" 70 -->", getLetterGrade(70))
```

```
##  70 --> C
```

```
cat(" 61 -->", getLetterGrade(61))
```

```
##  61 --> D
```

```
cat(" 22 -->", getLetterGrade(22))
```

```
##  22 --> F
```

Practice - Write the output by hand

5 minutes - no typing!

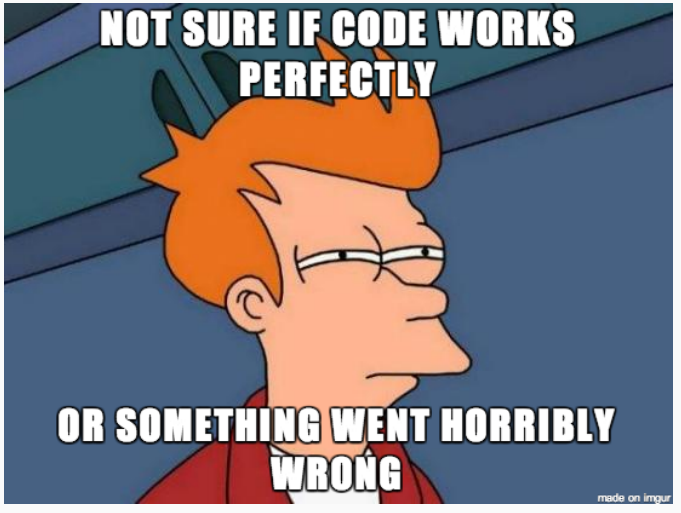
```
f1 <- function(x) {
    x = x + 1
    if ((x %% 2) == 0) {
        x = x - 1
    }
    y = 2*x
    cat(y, '\n')
}
f2 <- function(x) {
    if ((x %% 3) == 0) {
        cat('woo!\n')
        cat(x %/% 3)
    }
    cat(x %% 2, '\n')
}
```

```
f3 <- function(x) {
    if (x > 0) {
        cat('cat')
        x = 2*x
    } else if (x <= 0) {
        x = abs(x)
        cat('tac')
    }
    cat(x , '\n')
}
```

Write the output of this code by hand:

```
cat(f1(7))
cat(f1(12))
cat(f2(9))
cat(f2(11))
cat(f3(-9))
cat(f3(15))
```

Why write test functions?



- They help you understand the problem
- They verify that a function is working as expected

Test function "syntax"

Basic format:

```
functionName <- function(ARG1, ARG2,...) {
    STATEMENTS
    return(VALUE)
}

testFunctionName <- function() {
    cat("Testing functionName()...")
    <insert test cases>
    cat("Passed!\n")
}
```

Test case types

- **Normal Cases**: Typical inputs.
- **Large Cases**: Typical input, larger than usual.
- **Edge Cases**: Pairs of inputs that bound important points, e.g., if checking whether `n < 2`, two edge cases are when `n = 1.99`, `n = 2`.
- **Special Cases**: Negative numbers, `0` and `1` for integers, the empty string (`""`), and different type inputs, e.g. `"2"` instead of `2`.
- **Varying Results**: Cover multiple possible results, e.g. both `TRUE` and `FALSE` outcomes.

Testing with `stopifnot()`

`stopifnot()` stops the function if whatever is inside the `()` is not `TRUE`.

```
isEvenNumber <- function(n) {
    return((n %% 2) == 0)
}
```

Test cases:

- `isEvenNumber(42)` should be `TRUE`
- `isEvenNumber(43)` should be `FALSE`

```
testIsEvenNumber <- function() {
    cat("Testing isEvenNumber()... ")
    stopifnot(isEvenNumber(42) == TRUE)
    stopifnot(isEvenNumber(43) == FALSE)
    cat("Passed!\n")
}
```

```
testIsEvenNumber()
```

```
## Testing isEvenNumber()... Passed!
```

Testing function inputs

What if we gave `isEvenNumber()` the wrong input type?

```
isEvenNumber('42')
```

```
## Error in n%%2: non-numeric argument to binary operator
```

An improved function that checks inputs:

```
isEvenNumber <- function(n) {
    if (! is.numeric(n)) { return(FALSE) }
    return((n %% 2) == 0)
}
```

Now add more test cases:

```
testIsEvenNumber <- function() {
    cat("Testing isEvenNumber()...")
    stopifnot(isEvenNumber(42) == TRUE)
    stopifnot(isEvenNumber(43) == FALSE)
    stopifnot(isEvenNumber('not a number') == FALSE)
    cat("Passed!\n")
}
```

Debugging your code

Use `traceback()` to find the steps that led to an error.

```
f <- function(x) {
    return(x + 1)
}
g <- function(x) {
    return(f(x) - 1)
}
```

```
g('a')
```

```
## Error in x + 1: non-numeric argument to binary operator
```

```
traceback()
```

```
2: f(x) at #2
1: g("a")
```

Group Practice

20 minutes - In groups of 4, write the following functions:

For each of the following functions, start by writing a test function, e.g. `testIfFactor()`, that tests the function for a variety of values of inputs. Consider cases that you might not expect.

1. Write the function `isFactor(f, n)` that takes two int values `f` and `n`, and returns `TRUE` if `f` is a factor of `n`, and `FALSE` otherwise. Note that every integer is a factor of `0`. Assume `f` and `n` will only be numeric values.
2. Write the function `isMultiple(m, n)` that takes two integer values `m` and `n` and returns `TRUE` if `m` is a multiple of `n` and `FALSE` otherwise. Note that `0` is a multiple of every integer other than itself. Also, you should make constructive use of the `isFactor(f, n)` function you just wrote above. Assume `m` and `n` will only be numeric values.
3. Write the function `isPositiveMultipleOf4Or7(n)` that returns `TRUE` if `n` is a positive multiple of 4 or 7 and `FALSE` otherwise. Allow for cases where `n` is any data type.

Group Practice

20 minutes - In groups of 4

Write the function `getInRange(x, bound1, bound2)` which takes 3 numeric values: `x`, `bound1`, and `bound2`, where `bound1` is not necessarily less than `bound2`. If `x` is between the two bounds, just return it unmodified. Otherwise, if `x` is less than the lower bound, return the lower bound, or if `x` is greater than the upper bound, return the upper bound. For example:

- `getInRange(1, 3, 5)` returns `3` (the lower bound, since 1 is below the range [3,5])
- `getInRange(4, 3, 5)` returns `4` (the original value, since 4 is below the range [3,5])
- `getInRange(6, 3, 5)` returns `5` (the upper bound, since 6 is above the range [3,5])
- `getInRange(6, 5, 3)` returns `5` (the upper bound, since 6 is above the range [3,5])

Start by writing the test function `testGetInRange()` that tests for a variety of values of `x`, `bound1`, `bound2`.

Bonus: Re-write `getInRange(x, bound1, bound2)` without using conditionals

HW 2

- Start now!

- Don't modify the test functions!