Exam 1 Review

EMSE 6574, Section 11

John Helveston

October 11, 2019

Things to review

- Lesson pages (esp. tips sections)
- All lecture slides (esp. functions covered in class)
- Practice code tracing (look at previous quizzes, lecture slides, etc.)
- Memorize function (and test function) syntax!

Logical comparisons

Consider the following objects:

```r
w <- TRUE
x <- FALSE
y <- FALSE
z <- TRUE
```

Write code to answer the following questions:

a Write a logical statement that compares the objects `x`, `y`, and `z` and returns `TRUE`

b) Write a logical statement that compares the objects `x`, `y`, and `z` and returns `FALSE`

c) Fill in the correct relational operators to make this statement return `TRUE`:

`! (x __ y) & ! (z __ y)`

d) Fill in the correct relational operators to make this statement return `FALSE`:

`! (w __ y) | (z __ y)`

Order of operations

What would this return?

```r
sqrt(1 + abs(-8))
```

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

What would this return?

```r
sqrt(2*64^0.5)
```

```
## [1] 4
```

Numeric Data

Floats / Doubles:

```r
class(3.14)
```

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

```r
typeof(3.14)
```

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

"Integers":

```r
class(3)
```

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

```r
typeof(3)
```

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

Actual Integers

```r
class(3L)
```

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

```r
typeof(3L)
```

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

Check if a number is an "integer" in value:

```r
is.integer(3)
```

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

```r
3 == as.integer(3)
```

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

```r
3 == 3L
```

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

Logical Data

`TRUE` or `FALSE`

```r
x <- 1
y <- 2
```

```r
x > y # Is x greater than y?
```

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

```r
x == y
```

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

Character Data

Things in quotes (which we call, "strings")

```
s <- '3.14'
class(s)
```

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

What happens if we do this?

```
s + 7
```

```
## Error in s + 7: non-numeric argument to binary operator
```

```
a_number <- 3.14
as.character(a_number)
```

```
## [1] "3.14"
```

Integer division

Integer division drops the remainder

```r
4 / 3 # Regular division
```

```
## [1] 1.333333
```

```r
4 %/% 3 # Integer division
```

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

What will this return?

```r
4 %/% 5
```

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

What will this return?

```r
4 %/% 4
```

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

Modulus operator

Modulus returns the remainder after doing integer division

```r
5 %% 3
```

```
## [1] 2
```

```r
3.1415 %% 3
```

```
## [1] 0.1415
```

What will this return?

```r
4 %% 5
```

```
## [1] 4
```

What will this return?

```r
4 %% 4
```

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

Number "chopping"

"Chopping" with orders of 10:

The mod operator (`%%`) "chops" a number and returns everything to the *right*

```r
123456 %% 1
```

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

```r
123456 %% 10
```

```
## [1] 6
```

```r
123456 %% 100
```

```
## [1] 56
```

The integer divide operator (`%/%`) "chops" a number and returns everything to the *left*

```r
123456 %/% 1
```

```
## [1] 123456
```

```r
123456 %/% 10
```

```
## [1] 12345
```

```r
123456 %/% 100
```

```
## [1] 1234
```

Functions

```
FNAME <- function(ARG1, ARG2, ETC) {
    STATEMENT1
    STATEMENT2
    return(VALUE)
}
```

How I remember this:

"function name"	**is a function**		**of ()**	**that does...**
`FNAME`	`<-`	`function`	`(ARG1, ARG2, ETC)`	`{}`

Functions

```
FNAME <- function(ARG1, ARG2, ETC) {
    STATEMENT1
    STATEMENT2
    return(VALUE)
}
```

How I remember this:

"function name"	**is a function**		**of ()**	**that does...**
`FNAME`	`<-`	`function`	`(ARG1, ARG2, ETC)`	`{}`

Example:

"function name"	**is a function**		**of ()**	**that does...**
`mySqrt`	`<-`	`function`	`(n)`	`{ return(n^0.5) }`

Test Functions

```r
testFNAME <- function() {
    cat('Testing FNAME()...')
    # Insert tests here
    stopifnot(FNAME(ARG1, ARG2, ETC) == TRUE)
    cat('Passed!')
}
```

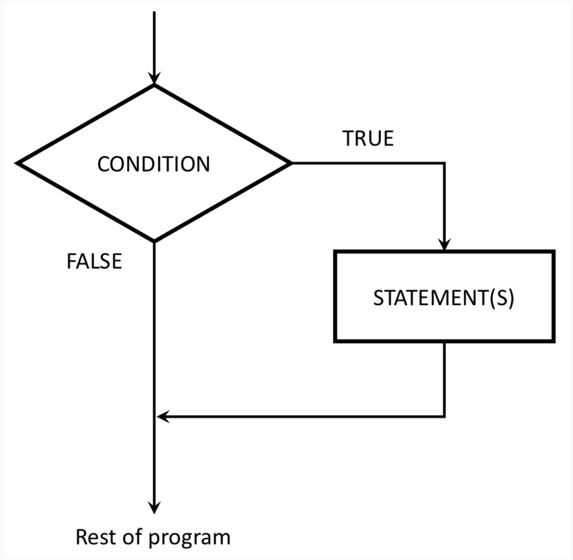
`stopifnot()` stops the function if its argument is not `TRUE`.

The `if` statement

Basic format:

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

Here's the general idea:



Use `if` statements to filter function inputs

Write the function `isEvenNumber(n)` that returns `TRUE` if `n` is an even number and `FALSE` otherwise. If `n` is not a number, the function should return `FALSE` instead of causing an error.

Test cases:

- `isEvenNumber(42) == TRUE`
- `isEvenNumber(43) == FALSE`
- `isEvenNumber('not a number') == FALSE`

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

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

The `almostEqual()` function

```
almostEqual <- function(d1, d2) {
    epsilon = 0.00001
    return(abs(d1-d2) <= epsilon)
}
```

```
0.1 + 0.2 == 0.3
```

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

```
almostEqual(0.1 + 0.2, 0.3)
```

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

Iteration review: loop over a sequence

```r
seq(5)
```

```
## [1] 1 2 3 4 5
```

`for` loop

```r
for (i in seq(5)) {
    cat(i, '\n')
}
```

```
## 1
## 2
## 3
## 4
## 5
```

`while` loop

```r
i <- 1
while (i <= 5) {
    cat(i, '\n')
    i <- i + 1
}
```

```
## 1
## 2
## 3
## 4
## 5
```

Search for something in a sequence

```r
seq(5)
```

```
## [1] 1 2 3 4 5
```

Count of **even** numbers in sequence:

`for` loop

```r
count <- 0
for (i in seq(5)) {
    if (i %% 2 == 0) {
        count <- count + 1
    }
}
count
```

```
## [1] 2
```

`while` loop

```r
count <- 0
i <- 1
while (i <= 5) {
    if (i %% 2 == 0) {
        count <- count + 1
    }
    i <- i + 1
}
count
```

```
## [1] 2
```

Making a sequence

1. Use the `seq()` function

2. Use the `:` operator

```r
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

```r
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

`break` and `next`

`break`

Note: `break` doesn't require `()`

Forces a loop to stop and "break" out of the loop.

```
for (val in 1:5) {
    if (val == 3) {
        break
    }
    cat(val, '\n')
}
```

```
1
2
```

In a nested loop:

```
for (i in 1:3) {
  cat('*')
    for (j in 1:3) {
        if (j == 3) {
            break
        }
        cat(j, '\n')
    }
}
```

```
*1
2
*1
2
*1
2
```

`break` and `next`

`next`

Skips to the *next* iteration of a loop

```
for (val in 1:5) {
    if (val == 3) {
        next
    }
    cat(val, '\n')
}
```

```
1
2
4
5
```

In a nested loop:

```
for (i in 1:3) {
  cat('*')
    for (j in 1:3) {
        if (j == 2) {
            next
        }
        cat(j, '\n')
    }
}
```

```
*1
3
*1
3
*1
3
```

`for` vs. `while`

Use `for` loops when there is a *known* number of iterations.

Use `while` loops when there is an *unknown* number of iterations.

Vectors

The universal vector generator: `c()`

```r
# Numeric vectors
x <- c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

```r
# Character vectors
y <- c('one', 'two', 'three')
y
```

```
## [1] "one"   "two"   "three"
```

```r
# Logical vectors
y <- c(TRUE, FALSE, TRUE)
y
```

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

Elements in vectors must be the same type

If a vector has a *single* character element, R makes everything a **character**:

```r
c(1, 2, "3")
```

```
## [1] "1" "2" "3"
```

```r
c(TRUE, FALSE, "TRUE")
```

```
## [1] "TRUE"  "FALSE" "TRUE"
```

If a vector has numeric and logical elements, R makes everything a **number**:

```r
c(1, 2, TRUE, FALSE)
```

```
## [1] 1 2 1 0
```

If a vector has integers and floats, R makes everything a **float**:

```r
c(1L, 2, pi)
```

```
## [1] 1.000000 2.000000 3.141593
```

Using vectors instead of a loop: Summation

```
x <- 1:10
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Using a loop:

```
total <- 0
for (item in x) {
    total <- total + item
}
total
```

```
## [1] 55
```

Using the `sum()` function:

```
sum(x)
```

```
## [1] 55
```

Vector slicing wit brackets `[ ]`

```r
x <- seq(1, 10)
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
x[1] # Returns the first element
```

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

```r
x[3] # Returns the third element
```

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

```r
x[1:3]  # Returns the first three elements
```

```
## [1] 1 2 3
```

```r
x[c(2, 7)] # Returns the 2nd and 7th elements
```

```
## [1] 2 7
```

```r
x[length(x)] # Returns the last element
```

```
## [1] 10
```

Use negative integers to *remove* elements

```r
x <- seq(1, 10)
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
x[-1] # Returns everything except the first element
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

```r
x[-1:-3]  # Returns everything except the first three elements
```

```
## [1]  4  5  6  7  8  9 10
```

```r
x[-c(2, 7)] # Returns everything except the 2nd and 7th elements
```

```
## [1]  1  3  4  5  6  8  9 10
```

```r
x[-length(x)] # Returns everything except the last elements
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

Logical indices

```r
x <- seq(10, 1)
x
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Create a logical vector:

```r
x > 5
```

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

Use logical vector in brackets to filter elements:

```r
x[x > 5]
```

```
## [1] 10  9  8  7  6
```

Comparing vectors

Check if 2 vectors are the same:

```r
x <- c(1, 2, 3)
y <- c(1, 2, 3)
```

```r
x == y
```

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

To check if *all* elements are the same, use `all()`:

```r
all(x == y)
```

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

Comparing vectors

To check if *any* elements are the same, use `any()`:

```r
a <- c(1, 2, 3)
b <- c(-1, 2,-3)
a == b
```

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

```r
all(a == b)
```

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

```r
any(a == b)
```

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

Begin list of all problems solved in class

Number chopping

1) `onesDigit(x)`: Write a function that takes an integer and returns its ones digit.

2) `tensDigit(x)`: Write a function that takes an integer and returns its tens digit.

`onesDigit(x)` tests:

- onesDigit(123) == 3
- onesDigit(7890) == 0
- onesDigit(6) == 6
- onesDigit(-54) == 4

`tensDigit(x)` tests:

- tensDigit(456) == 5
- tensDigit(23) == 2
- tensDigit(1) == 0
- tensDigit(-7890) == 9

General function writing

1) `eggCartons(eggs)`: Write a program that reads in a non-negative number of eggs and prints the number of egg cartons required to hold that many eggs (given that each egg carton holds one dozen eggs, and you cannot buy fractional egg cartons). Be sure your program works for multiples of 12, including 0.

- eggCartons(0) == 0
- eggCartons(1) == 1
- eggCartons(12) == 1
- eggCartons(13) == 2
- eggCartons(24) == 2
- eggCartons(25) == 3

2) `militaryTimeToStandardTime(n)`: Write a program that takes an integer between 0 and 23 (representing the hour in military time), and returns the same hour in standard time. For example, 17 in military time is 5 o'clock in standard time.

- militaryTimeToStandardTime(0) == 12
- militaryTimeToStandardTime(1) == 1
- militaryTimeToStandardTime(11) == 11
- militaryTimeToStandardTime(12) == 12
- militaryTimeToStandardTime(13) == 1
- militaryTimeToStandardTime(23) == 11

Conditionals (if / else)

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

Loops / Vectors

1) `sumFromMToN(m, n)`: Write a function that sums the total of the integers between `m` and `n`.
Challenge: Try solving this without a loop (it's possible - Google it!).

- `sumFromMToN(5, 10) == (5+6+7+8+9+10)`
- `sumFromMToN(1, 1) == 1`

2) `sumEveryKthFromMToN(m, n, k)`: Write a function to sum every kth integer from `m` to `n`.

- `sumEveryKthFromMToN(5, 20, 7) == (5 + 12 + 19)`
- `sumEveryKthFromMToN(1, 10, 2) == (1 + 3 + 5 + 7 + 9)`
- `sumEveryKthFromMToN(0, 0, 1) == 0`

3) `sumOfOddsFromMToN(m, n)`: Write a function that sums every *odd* integer between `m` and `n`.
Challenge: Try solving this without a loop (Hint: use a vector operation...we'll cover this next week!).

- `sumOfOddsFromMToN(4, 10) == (5 + 7 + 9)`
- `sumOfOddsFromMToN(5, 9) == (5 + 7 + 9)`

Loops / Vectors

1) `isMultipleOf4Or7(n)`

Write a function that returns `TRUE` if `n` is a multiple of 4 or 7 and `FALSE` otherwise. Here's some test cases:

- `isMultipleOf4Or7(0) == FALSE`
- `isMultipleOf4Or7(1) == FALSE`
- `isMultipleOf4Or7(-7) == FALSE`
- `isMultipleOf4Or7(4) == TRUE`
- `isMultipleOf4Or7(7) == TRUE`
- `isMultipleOf4Or7(28) == TRUE`
- `isMultipleOf4Or7('notANumer') == FALSE`

2) `nthMultipleOf4Or7(n)`

Write a function that returns the nth positive integer that is a multiple of either 4 or 7. Hint: use `isMultipleOf4Or7(n)` as a helper function! Here's some test cases:

- `nthMultipleOf4Or7(1) == 4`
- `nthMultipleOf4Or7(2) == 7`
- `nthMultipleOf4Or7(3) == 8`
- `nthMultipleOf4Or7(4) == 12`
- `nthMultipleOf4Or7(5) == 14`
- `nthMultipleOf4Or7(6) == 16`
- `nthMultipleOf4Or7(10) == 28`

Loops / Vectors

1) `isPrime(n)`

Write a function that takes a non-negative integer, `n`, and returns `TRUE` if it is a prime number and `FALSE` otherwise. Here's some test cases:

- `isPrime(1) == FALSE`
- `isPrime(2) == TRUE`
- `isPrime(7) == TRUE`
- `isPrime(13) == TRUE`
- `isPrime(14) == FALSE`

2) `nthPrime(n)`

Write a function that takes a non-negative integer, `n`, and returns the nth prime number, where `nthPrime(1)` returns the first prime number (2). Hint: use `isPrime(n)` as a helper function! Here's some test cases:

- `nthPrime(1) == 2`
- `nthPrime(2) == 3`
- `nthPrime(3) == 5`
- `nthPrime(4) == 7`
- `nthPrime(7) == 17`

Vectors

1) `reverse(x)`

Write a function that returns the vector in reverse order. You cannot use the `rev()` function. You cannot use loops! Test cases:

- `reverse(c(5, 1, 3)) == c(3, 1, 5)`
- `reverse(c('a', 'b', 'c')) == c('c', 'b', 'a')`
- `reverse(c(FALSE, TRUE, TRUE)) == c(TRUE, TRUE, FALSE)`
- `reverse(seq(10)) == seq(10, 1, -1)`

2) `middleValue(a)`

Write a function that takes a vector of numbers `a` and returns the value of the middle element (or the average of the two middle elements). Test cases:

- `middleValue(c(0,0,0)) == 0`
- `middleValue(c(1,2,3)) == 2`
- `middleValue(c(4,5,6,7,8)) == 6`
- `middleValue(c(5,3,8,4)) == mean(c(3,8))`
- `middleValue(c(4,5,6,7)) == mean(c(5,6))`

Vectors

1) `dotProduct(a, b)`

The "dot product" of two vectors is the sum of the products of the corresponding terms. So the dot product of the vectors `c(1,2,3)` and `c(4,5,6)` is `(1*4) + (2*5) + (3*6)`, or `4 + 10 + 18 = 32`. Write a function that takes two vectors and returns the dot product of those vectors. If the vectors are not equal length, ignore the extra elements in the longer vector. Try not to use loops! Test cases:

- `dotProduct(c(1,2,3), c(4,5,6)) == 32`
- `dotProduct(c(1,2), c(4,5,6)) == 14`
- `dotProduct(c(2,3,4), c(-7,1,9)) == 25`
- `dotProduct(c(0,0,0), c(-7,1,9)) == 0`

2) `alternatingSum(a)`

Write a function that takes a vector of numbers `a` and returns the alternating sum, where the sign alternates from positive to negative or vice versa. Test cases:

- `alternatingSum(c(5,3,8,4)) == (5 - 3 + 8 - 4)`
- `alternatingSum(c(1,2,3)) == (1 - 2 + 3)`
- `alternatingSum(c(0,0,0)) == 0`