Midterm Review

EMSE 4574: Intro to Programming for Analytics

John Paul Helveston

October 15, 2020

Things to review

- Lecture slides, especially practice puzzles covered in class)
- Previous quizzes
- Memorize syntax for:
  - if / else statements
  - loops
  - functions
  - test functions

Operators: Relational (=, <, >, <=, >=) and Logical (&, |, !)

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

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

b) Fill in **relational** operators to make this statement return TRUE:

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

c) Fill in **logical** operators to make this statement return FALSE:

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

Numeric Data

Doubles:

```
typeof(3.14)
```

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

"Integers":

```
typeof(3)
```

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

Actual Integers

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

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

Check if a number is an "integer":

```r
n <- 3
is.integer(n) # Doesn't work!
```

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

```r
n == as.integer(n) # Compare n to a converted version of itself
```

```
## [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
```

Tricky data type stuff

Logicals become numbers when doing math

```r
TRUE + 1 # TRUE becomes 1
```

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

```r
FALSE + 1 # FALSE becomes 0
```

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

Be careful of accidental strings

```r
typeof("3.14")
```

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

```r
typeof("TRUE")
```

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

Integer division: %/%

Integer division drops the remainder

Example:

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

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

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

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

Integer division: %/%

Integer division drops the remainder

What will this return?

```
4 %/% 4
```

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

What will this return?

```
4 %/% 5
```

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

Modulus operator: `%%`

Modulus returns the remainder *after* doing integer division

Example:

```r
5 %% 3
```

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

```r
3.1415 %% 3
```

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

Modulus operator: %%

Modulus returns the remainder *after* doing integer division

What will this return?

```r
4 %% 4
```

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

What will this return?

```r
4 %% 5
```

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

Number "chopping" with 10s (only works with `n > 0`)

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
```

Integer division (`%/%`) "chops" a number and returns everything to the *left*

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

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

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

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

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

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

Basic function syntax

```
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Basic function syntax

In English:

> "`functionName` is a `function` of `arguments` that does..."

```
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Basic function syntax

Example:

> "`squareRoot` is a `function` of `n` that...returns the square root of `n`"

```r
squareRoot <- function(n) {
    return(n^0.5)
}
```

```r
squareRoot(64)
```

```
## [1] 8
```

Test function "syntax"

Function:

```r
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Test function:

```r
test_functionName <- function() {
    cat("Testing functionName()...")
    # Put test cases here
    cat("Passed!\n")
}
```

Writing test cases with `stopifnot()`

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

Function:

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

- `isEven(1)` should be `FALSE`
- `isEven(2)` should be `TRUE`
- `isEven(-7)` should be `FALSE`

Test function:

```r
test_isEven <- function() {
    cat("Testing isEven()...")
    stopifnot(isEven(1) == FALSE)
    stopifnot(isEven(2) == TRUE)
    stopifnot(isEven(-7) == FALSE)
    cat("Passed!\n")
}
```

When testing *numbers*, use `almostEqual()`

Rounding errors can cause headaches:

```
x <- 0.1 + 0.2
x
```

```
## [1] 0.3
```

```
x == 0.3
```

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

```
print(x, digits = 20)
```

```
## [1] 0.30000000000000004441
```

Define a function that checks if two values are *almost* the same:

```
almostEqual <- function(n1, n2,
threshold = 0.00001) {
    return(abs(n1 - n2) <= threshold)
}
```

```
x <- 0.1 + 0.2
almostEqual(x, 0.3)
```

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

Use `if` statements to filter function inputs

Example: 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`.**

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

```r
isEvenNumber(2)
```

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

```r
isEvenNumber("not_a_number")
```

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

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

```r
isEvenNumber(2)
```

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

```r
isEvenNumber("not_a_number")
```

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

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

1. Build the sequence
2. Iterate over it

```r
for (i in 1:5) { # Define the sequence
    cat(i, '\n')
}
```

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

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

1. Define stopping condition
2. Manually increase condition

```r
i <- 1
while (i <= 5) { # Define stopping condition
    cat(i, '\n')
    i <- i + 1 # Increase condition
}
```

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

Search for something in a sequence

Example: count the **even** numbers in sequence: `1, (2), 3, (4), 5`

`for` loop

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

```r
count
```

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

`while` loop

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

```r
count
```

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

The universal vector generator: c()

Numeric vectors

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

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

Character vectors

```
y <- c('one', 'two',
'three')
y
```

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

Logical vectors

```
z <- c(TRUE, FALSE,
TRUE)
z
```

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

Elements in vectors must be the same type

Type hierarchy:

- `character` > `numeric` > `logical`
- `double` > `integer`

Coverts to characters:

```r
c(1, "foo", TRUE)
```

```
## [1] "1"    "foo"
"TRUE"
```

Coverts to numbers:

```r
c(7, TRUE, FALSE)
```

```
## [1] 7 1 0
```

Coverts to double:

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

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

Most functions operate on vector *elements*

```r
x <- c(3.14, 7, 10, 15)
```

```r
round(x)
```

```
## [1]  3  7 10 15
```

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

```r
isEven(x)
```

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

"Summary" functions **return one value**

```r
x <- c(3.14, 7, 10, 15)
```

```r
length(x)
```

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

```r
sum(x)
```

```
## [1] 35.14
```

```r
prod(x)
```

```
## [1] 3297
```

```r
min(x)
```

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

```r
max(x)
```

```
## [1] 15
```

```r
mean(x)
```

```
## [1] 8.785
```

Use brackets [ ] to get elements from a vector

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

Indices start at 1:

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

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

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

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

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

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

Slicing with a vector of indices:

```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
```

Use negative integers to *remove* elements

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

```r
x[-1] # Drops the first element
```

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

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

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

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

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

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

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

Slicing with logical indices

```
x <- seq(1, 20, 3)
x
```

```
## [1]  1  4  7 10 13 16 19
```

```
x > 10 # Create a logical vector based on some condition
```

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

Slice x with logical vector - only TRUE elements will be returned:

```
x[x > 10]
```

```
## [1] 13 16 19
```

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
```

Comparing vectors with `all()` and `any()`

`all()`: Check if *all* elements are the same

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

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

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

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

`any()`: Check if *any* elements are the same

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

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

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

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

Begin list of all problems solved in class

General function writing

`eggCartons(eggs)`: Write a function that reads in a non-negative number of eggs and prints the number of egg cartons required to hold that many eggs. Each egg carton holds one dozen eggs, and you cannot buy fractional egg cartons.

- eggCartons(0) == 0
- eggCartons(1) == 1
- eggCartons(12) == 1
- eggCartons(25) == 3

`militaryTimeToStandardTime(n)`: Write a function that takes an integer between 0 and 23 (representing the hour in military time), and returns the same hour in standard time.

- militaryTimeToStandardTime(0) == 12
- militaryTimeToStandardTime(3) == 3
- militaryTimeToStandardTime(12) == 12
- militaryTimeToStandardTime(13) == 1
- militaryTimeToStandardTime(23) == 11

Number chopping

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

Tests:

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

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

Tests:

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

Top-down design

Create a function, `isRightTriangle(a, b, c)` that returns `TRUE` if the triangle formed by the lines of length `a`, `b`, and `c` is a right triangle and `FALSE` otherwise. Use the `hypotenuse(a, b)` function in your solution. **Hint**: you may not know which value (`a`, `b`, or `c`) is the hypotenuse.

```
hypotenuse <- function(a, b) {
    return(sqrt(sumOfSquares(a, b)))
}
```

```
sumOfSquares <- function(a, b) {
    return(a^2 + b^2)
}
```

Conditionals (if / else)

`getType(x)`: Write the function `getType(x)` that returns the type of the data (either `integer`, `double`, `character`, or `logical`). Basically, it does the same thing as the `typeof()` function (but you can't use `typeof()` in your solution).

- `getType(3) == "double"`
- `getType(3L) == "integer"`
- `getType("foo") == "character"`
- `getType(TRUE) == "logical"`

Conditionals (if / else)

For each of the following functions, start by writing a test function that tests the function for a variety of values of inputs. Consider cases that you might not expect!

`isFactor(f, n)`: Write the function `isFactor(f, n)` that takes two integer values 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, e.g. `2` is a factor of `6`.

`isMultiple(m, n)`: Write the function `isMultiple(m, n)` that takes two integer values 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. Hint: You may want to use the `isFactor(f, n)` function you just wrote above. Assume `m` and `n` will only be numeric values.

Conditionals (if / else)

Write the function `getInRange(x, bound1, bound2)` which takes 3 numeric values: `x`, `bound1`, and `bound2` (`bound1` is not necessarily less than `bound2`). If `x` is between the two bounds, just return `x`, but 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 [3,5])
- `getInRange(4, 3, 5)` returns `4` (the original value, since 4 is between [3,5])
- `getInRange(6, 3, 5)` returns `5` (the upper bound, since 6 is above [3,5])
- `getInRange(6, 5, 3)` returns `5` (the upper bound, since 6 is above [3,5])

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

`for` loops

`sumFromMToN(m, n)`: Write a function that sums the total of the integers between `m` and `n`.
Challenge: Try solving this without a loop!

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

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

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

`sumOfOddsFromMToN(m, n)`: Write a function that sums every *odd* integer between `m` and `n`.

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

`for` loop with `break` & `next`

`sumOfOddsFromMToNMax(m, n, max)`: Write a function that sums every *odd* integer from `m` to `n` until the sum is less than the value `max`. Your solution should use both `break` and `next` statements.

- `sumOfOddsFromMToNMax(1, 5, 4) == (1 + 3)`
- `sumOfOddsFromMToNMax(1, 5, 3) == (1)`
- `sumOfOddsFromMToNMax(1, 5, 10) == (1 + 3 + 5)`

`while` loops

`isMultipleOf4Or7(n)`: Write a function that returns `TRUE` if `n` is a multiple of 4 or 7 and `FALSE` otherwise.

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

`nthMultipleOf4Or7(n)`: Write a function that returns the nth positive integer that is a multiple of either 4 or 7.

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

Loops / Vectors

`isPrime(n)`: Write a function that takes a non-negative integer, `n`, and returns `TRUE` if it is a prime number and `FALSE` otherwise. Use a loop or vector:

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

`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 a while loop!

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

Vectors

`reverse(x)`: Write a function that returns the vector in reverse order. You cannot use the `rev()` function.

- `all(reverseVector(c(5, 1, 3)) == c(3, 1, 5))`
- `all(reverseVector(c('a', 'b', 'c')) == c('c', 'b', 'a'))`
- `all(reverseVector(c(FALSE, TRUE, TRUE)) == c(TRUE, TRUE, FALSE))`

`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.

- `alternatingSum(c(5,3,8,4)) == (5 - 3 + 8 - 4)`
- `alternatingSum(c(1,2,3)) == (1 - 2 + 3)`
- `alternatingSum(c(0,0,0)) == 0`
- `alternatingSum(c(-7,5,3)) == (-7 - 5 + 3)`