Week 4: *Conditionals & Testing*

🏛 EMSE 4571 / 6571: Intro to Programming for Analytics

👤 John Paul Helveston

📅 February 08, 2024

Quiz 3

Write your name on the quiz!

Rules:

- Work alone; no outside help of any kind is allowed.
- No calculators, no notes, no books, no computers, no phones.
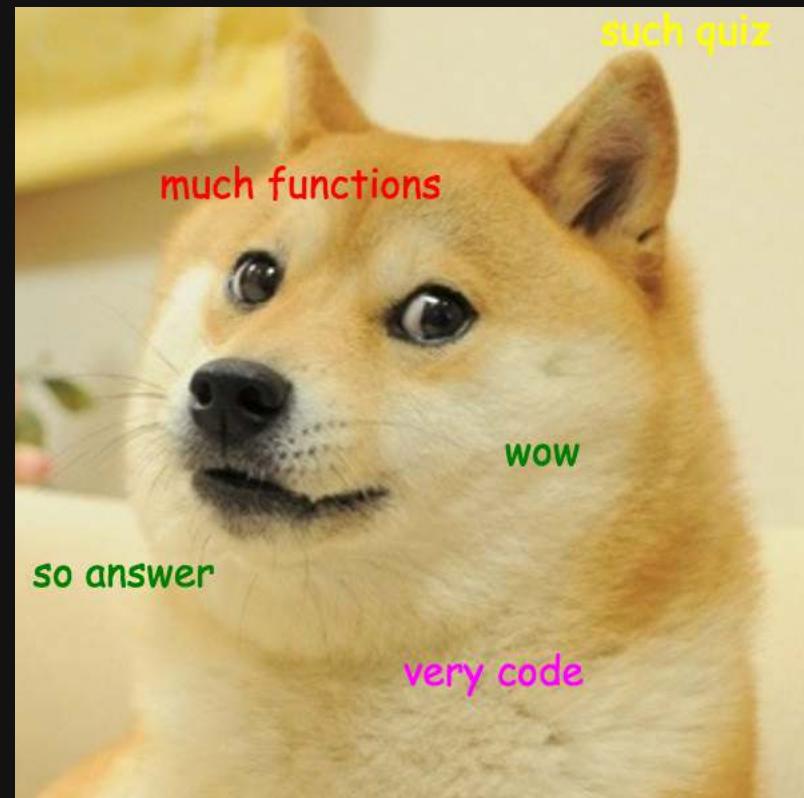
Week 4: *Conditionals & Testing*

1. Conditionals

2. Testing

BREAK

3. Tips

Week 4: *Conditionals & Testing*

1. Conditionals

2. Testing

BREAK

3. Tips

"Flow Control"

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 ) {
    # Do stuff here
}
```

Flow chart:
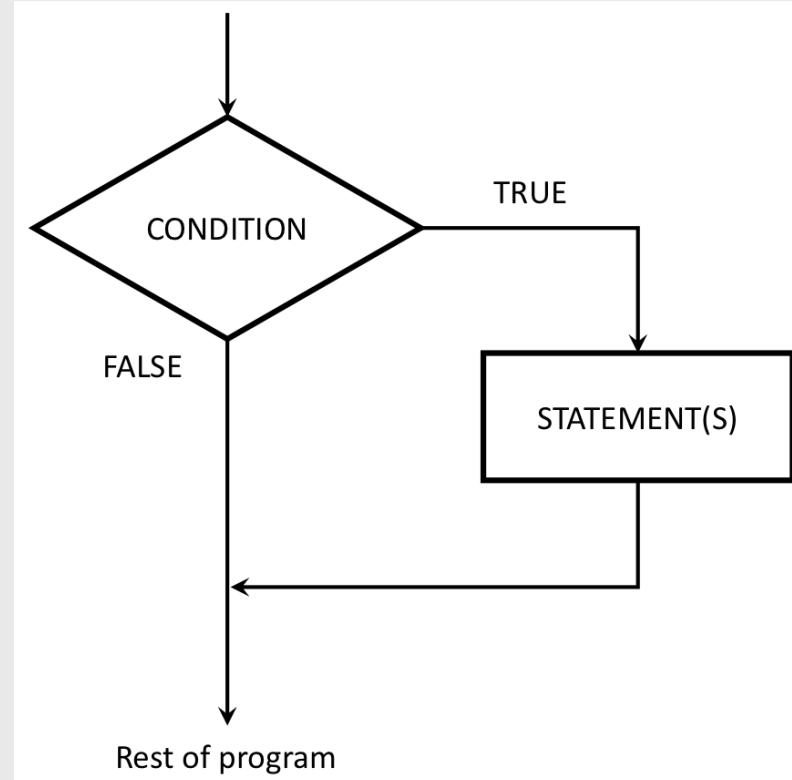
Quick code tracing

Consider this function:

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

What will this print?

```r
f(1)
f(0)
```

Quick practice

Write the function `absValue(n)` that returns the absolute value of a number (and no cheating - you can't use the built-in `abs()` function!)

Tests:

- `absValue(7) == 7`
- `absValue(-7) == 7`
- `absValue(0) == 0`

Adding an else

Basic format:

```
if ( CONDITION ) {
  # Do stuff here
} else {
  # Do other stuff here
}
```

Flow chart:
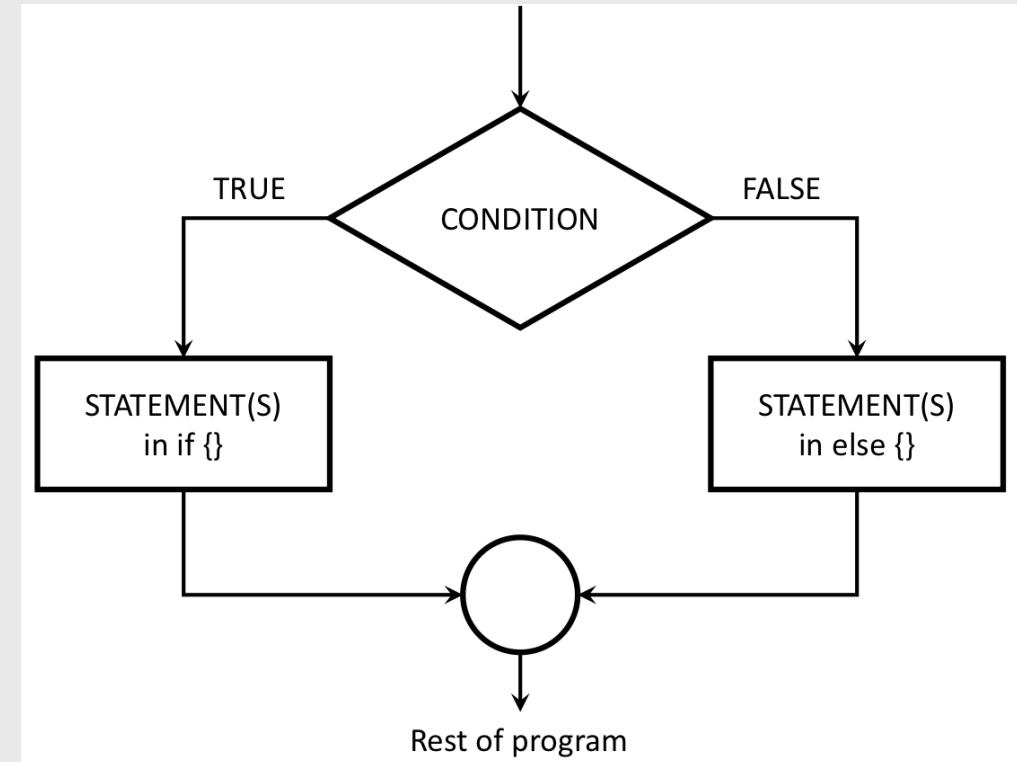
Quick code tracing

Consider this code:

```r
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")
}
```

What will this print?

```r
f(0)
f(1)
f(2)
```

else if chains

Example - "bracketing" problems:

```r
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)
}
```

Check function output:

```r
getLetterGrade(99)
```
```
#> [1] "A"
```

```r
getLetterGrade(88)
```
```
#> [1] "B"
```

```r
getLetterGrade(70)
```
```
#> [1] "C"
```

```r
getLetterGrade(61)
```
```
#> [1] "D"
```

```r
getLetterGrade(22)
```
```
#> [1] "F"
```

Your turn

Write the function `getType(x)` that returns the type of the data (either `integer`, `double`, `character`, or `logical`).

Basically, it should do the same thing as the `typeof()` function (but you can't use `typeof()` in your solution).

Tests:

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

Week 4: *Conditionals & Testing*

1. Conditionals

2. Testing

BREAK

3. Tips

Why write test functions?

1. They help you understand the problem

2. They verify that a function is working as expected

Test functions help you understand the problem

Rubber Duck Debugging
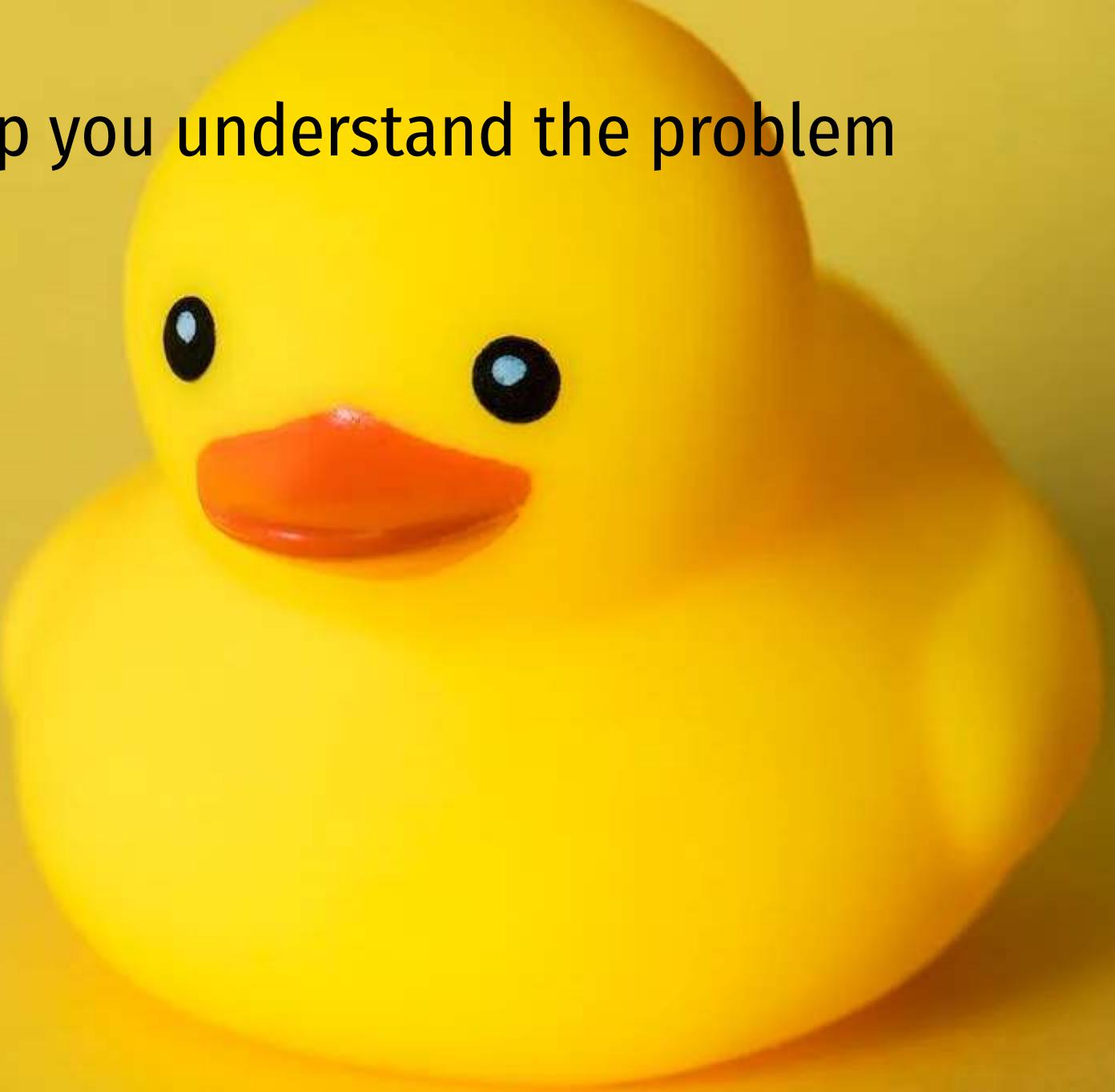
Test function "syntax"

Function:

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

Test function:

```
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")
}
```

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")
}
```

```r
test_isEven()
```

```
#> Testing isEven()...Passed!
```

Write the test function *first*!

Step 1: Write the test function

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

Step 2: Write the function

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

Step 3: Test the function

```r
test_isEven()
```

```
#> Testing isEven()...Passed!
```

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

Example:

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

- isEven(1) == FALSE
- isEven(2) == TRUE
- isEven(-7) == FALSE

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

Example:

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

Need cases that return both TRUE and FALSE

- isEven(52) == TRUE
- isEven(53) == FALSE
- isEven(5) == FALSE
- isEven(-5) == FALSE

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

Example:

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

- isEven(8675309) == FALSE
- isEven(-8675309) == FALSE
- isEven(1) == FALSE
- isEven(-1) == FALSE

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

Example:

```
isPositive <- function(n) {
    return(n > 0)
}
```

- isPositive(0.000001) == TRUE
- isPositive(0) == FALSE
- isPositive(-0.000001) == FALSE

Test cases to consider: NOLES

- **N**ormal cases

- **O**pposites

- **L**arge & small cases

- **E**dge cases

- **S**pecial cases

- Negative numbers
- 0 and 1 for integers
- The empty string, `""`
- Strange input *types*, e.g. `"2"` instead of 2.

Testing function inputs

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

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

```
isEven('42')
```

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

An improved function with input checks:

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

Testing function inputs

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

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

```
isEven('42')
```

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

An improved function that checks inputs:

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

```
isEven('42')
```

```
#> [1] NaN
```

```
isEven(TRUE)
```

```
#> [1] NaN
```

Your turn

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!

1) Write the function `isFactor(f, n)` that takes two integer values and returns TRUE if `f` is a factor of `n`, and FALSE otherwise (e.g. `2` is a factor of `6`). 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 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.

Intermission

05:00

Week 4: *Conditionals & Testing*

1. Conditionals

2. Testing

BREAK

3. Tips
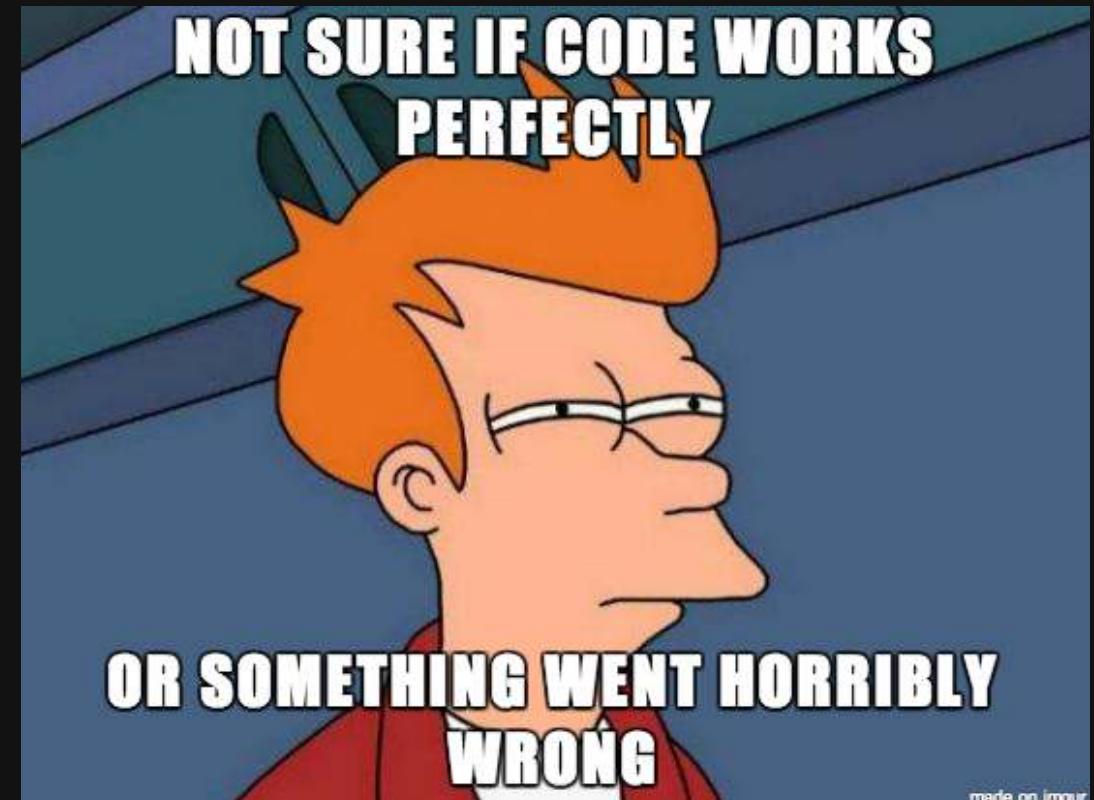
debugging

1.
I got this.
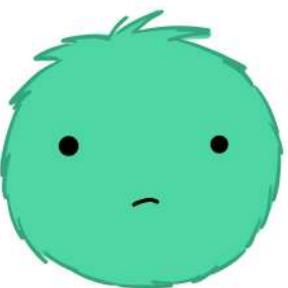
2.
Huh. Really thought that was it.

3.
(...)

4.
Fine. Restarting.

5.
OH WTF.

6.
Zombie meltdown

7.
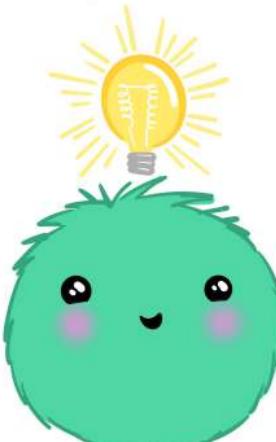
8.
A NEW HOPE!

clickety click click

9.
[insert awesome theme song]

10.
I ♥ CODING!

@allison_horst

Debugging your code

Use `traceback()` to find the steps that led to an error (the "call stack")

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

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

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

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

```r
traceback()
```

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

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

Rounding errors can cause headaches:

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

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

```
x == 0.3
```

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

Rounding errors can cause headaches:

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

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

```
x == 0.3
```

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

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

Rounding errors can cause headaches:

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

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

```r
x == 0.3
```

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

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

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

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

Rounding errors can cause headaches:

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

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

```r
x == 0.3
```

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

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

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

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

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

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

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

Checking for integer values

Since numbers are doubles by default, the `is.integer(x)` function can be confusing:

```
is.integer(7)
```

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

Define a new function that returns TRUE if the *value* is an integer:

```
is.integer.val <- function(x) {
    return(almostEqual(x, round(x)))
}
is.integer.val(7)
```

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

Checking for special data types

Not available: NA
value is "missing"

```
x <- NA
x == NA
```

```
#> [1] NA
```

No value: NULL
no value whatsoever

```
x <- NULL
x == NULL
```

```
#> logical(0)
```

Checking for special data types

Not available: NA
value is "missing"

```
x <- NA
x == NA
```

```
#> [1] NA
```

Have to use special function:

```
is.na(x)
```

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

No value: NULL
no value whatsoever

```
x <- NULL
x == NULL
```

```
#> logical(0)
```

Have to use special function:

```
is.null(x)
```

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

Your turn

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, 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])

You should also write a test function called `test_getInRange()`.

Bonus: Try writing `getInRange(x, bound1, bound2)` without using `if` or `else`

Your turn

isEvenish(x)

Given an arbitrary value x, return TRUE if it is an even number and FALSE otherwise. The function should also work for numbers provided as characters, so isEvenish("2") should return TRUE. Some test cases:

```
isEvenPositiveInt(2) == TRUE
isEvenPositiveInt("2") == TRUE
isEvenPositiveInt("yikes!") == FALSE
isEvenPositiveInt(3.14) == FALSE
isEvenPositiveInt(TRUE) == FALSE
```

Hint: it may be helpful to write an additional function called isEvenNumber(x) that handles the cases when you know x is a numeric value.

HW 4

You'll need to write a *test function* for each function!