Stats 20 Notes

Zooey Nguyen

Week 2

Chapter 3 Notes

Relational operations are vectorised (function/operator is applied to each element of a vector individually).

```
c(3,8) > 3
```

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

Vectors are compared element by element.

```
c(3,8) < c(1,4)
```

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

Recycling is done on this as well (if a vector is shorter, R will just lengthen it until it matches by repeating the vector), but as always it will give you a warning when you do this.

```
c(1,4) == c(5,3,1)
```

```
## Warning in c(1, 4) == c(5, 3, 1): longer object length is not a multiple of
## shorter object length
```

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

`any()` – takes a logical vector, returns TRUE if any values are TRUE.

```
first_ten_over_seven <- 1:10 > 7
first_ten_over_seven
```

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

```
any(1:10 > 7)
```

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

`all()` – takes a logical vector, returns TRUE if all values are TRUE. You can use this to check if two vectors are equal (all elements match).

```
first_ten_over_seven <- 1:10 > 7
first_ten_over_seven
```

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

```
all(1:10 > 7)
```

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

```
first_five <- c(1,2,3,4,5)
seq_five <- 1:5
all(first_five == seq_five)
```

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

`identical()` takes two objects and returns TRUE if they are exactly identical. This is different from == because this assuredly returns a single logical value, but == may not work as expected in all cases for your intentions (like if your variables end up being different lengths you don't just get FALSE, or if one is NA, the result from == is NA).

```r
# first_five is doubles because c() generates that, seq() is integers
identical(first_five,seq_five)
```

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

Discussion

- debugging strats
  - put `print`/`message` functions everywhere
    - danger: if you put `print` in the wrong place (like within a function), you may change the return type of the function. using `message` will avoid this
    - best to insert within control flows
  - desk check – stepping through your code and tracking your variables every step of the way
    - tracking outside of the program, like you manually do this
    - possible structure: table with the columns as variables and the rows are each step where a variable is changed. maybe make another column to note what happens in each row
  - use the browser! `browser()`
    - `help` will let you select from multiple options to track your program.
    - `where` will show you the order of function calls.
    - to go line-by-line, use `n` to take each step from the top of the program, here you can see the exact points your program is erroring
  - rubber ducky debugging method
  - always read the help files for your functions!!! `help(function_name)`
    - always read the detail/notes sections for potentially unintuitive behaviors of a function
- urgency levels, least to most
  - `message(message)` - can help track at what point in the run the program is
  - `warning(warning message)` - your code will work because you are making a decision about the output, but you anticipate that this may not be what the user means all the time
  - `stop(error message)` - stops the program to handle your own exceptions, defensively code (you should ensure things are validated from the jump, not later on)
- `function(...)` – ellipsis ... works as a sort of variable, indicating the function will take any number of unnamed args. ellipsis is preserved within the function when used.

Lecture 2019-10-09