CSE231 - Lab 02

Comparison Operators, Boolean Logic, Conditionals, Loops

Comparison Operators

We've talked about mathematical operators, your classic addition, subtraction, multiplication, etc., but in all programming languages, there are also comparison operators that return a boolean.

Operator	Name
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To
==	Equal To
!=	Not Equal To

Conditional Statements

You usually use boolean operations in tandem with a conditional statement. These conditional statements become “linked” when using them together:

```
if some_condition:  # runs if true, ‘if’ begins the linking of conditions

    ...

elif some_other_condition:  # runs if true and if all previous conditions were false

    ...

else:  # runs no matter what, only if all previous conditions were false

    ...
```

not

In addition to the comparison operators, there are 3 comparison keywords. I'll only be showing you one of them for now.

The '`not`' keyword results in `True` if the statement evaluates to `not True` (i.e. `False`)

```
- not True == False  # True
- not False == True  # True
- 5 > 4  # True
- not 5 > 4  # False
```

You can think of it as just taking the inverse of whatever boolean statement you're evaluating.

While-loop

A while-loop continuously checks a condition until it is `False`. Like if-statements, it requires a colon and an indented block of code.

Important to note, is that the condition is checked when first evaluated from proceeding lines, and then gets reevaluated after running through its respective block of code for each iteration.

There are some programming languages that have variations on this concept.

For-loop / range()

A for-loop iterates for a specified amount of times, or through a collection, as opposed to checking a condition like the while-loop. Commonly, a for-loop used to iterate for a certain range of numbers is written as '`for i in range(n):`', where '`i`' is an iterative integer that starts from 0, and ends at '`n-1`'.

```
range([start], stop, [step])
```

`range()` is an odd function. When 1 parameter is given, that becomes the '`stop`' parameter. When 2 are given, the first becomes the '`start`', the second becomes the '`stop`'. You can then add a third, '`step`' when you have the first 2 given.

The three parameters do about what you would expect.

`break`

Alternatively, if you don’t want to iterate through your loop again, you can preemptively interrupt it using the aptly named ‘`break`’ statement.

Typically, you would be checking a certain, outside condition where you wouldn’t want your loop to continue.

‘`break`’ is a keyword that stands alone in-line.

`continue`

The opposite of the break-statement would likely be this, the continue-statement. Instead of preemptively *breaking* out of the loop, ‘`continue`’ preemptively *repeats* the loop.

Like ‘`break`’, you would typically use ‘`continue`’ in special cases. It again sits alone in-line.

What's so special about `break`/`continue`?

`break` and `continue`, importantly, skip all lines below themselves that are within the same loop suite. They can *only* be used inside loop structures.

This, combined with `if`/`elif`/`else` statements, make it so that *the order in which you put your statements matter*. You can take advantage of this, but it can also be a hindrance at times.

This is a super important concept to be aware of for future assignments.

L2-4