Introduction to Programming

Spring 2022

Functions

- For Loop: A Quick Review
- Indefinite Loops
- Common Loop Patterns
  - Interactive Loops
  - Senteniel Loops
  - File Loops
  - Nested Loops
- **Computing with Boolean**
- Other Common Structures
- Event Loop

Computing with Booleans

- `if` and `while` both use Boolean expressions.
- Boolean expressions evaluate to `True` or `False`.
- So far we've used Boolean expressions to compare two values, e.g.

```
(while x >= 0)
```

Boolean Operators

- Sometimes our simple expressions do not seem expressive enough.
- Suppose you need to determine whether two points are in the same position – their x coordinates are equal and their y coordinates are equal.

Boolean Operators

```
if p1.getX() == p2.getX():
    if p1.getY() == p2.getY():
        # points are the same
    else:
        # points are different
else:
    # points are different
```

- Clearly, this is an awkward way to evaluate multiple Boolean expressions!
- Let's check out the three Boolean operators `and`, `or`, and `not`.

Boolean Operators

- The Boolean operators and and or are used to combine two Boolean expressions and produce a Boolean result.
  - `<expr> and <expr>`
  - `<expr> or <expr>`

Boolean Operators

- The `and` of two expressions is true exactly when both of the expressions are true.
- We can represent this in a *truth table*.

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

Boolean Operators

- In the truth table, *A* and *B* represent smaller Boolean expressions.
- Since each expression has two possible values, there are four possible combinations of values.
- The last column gives the value of `A and B` for each combination.

Boolean Operators

- The `or` of two expressions is true when either expressions is true.
- We can represent this in a *truth table*.

A	B	A or B
T	T	T
T	F	T
F	T	T
F	F	F

Boolean Operators

- The only time `or` is false is when both expressions are false.
- Also, note that `or` is true when both expressions are true. This isn't how we normally use "or" in language.

Boolean Operators

- The `not` operator computes the opposite of a Boolean expression.
- `not` is a *unary* operator, meaning it operates on a single expression.

A	not A
T	F
F	T

Boolean Operators

- We can put these operators together to make arbitrarily complex Boolean expressions.
- The interpretation of the expressions relies on the precedence rules for the operators.

Boolean Operators

- Consider `a or not b and c`
- How should this be evaluated?
- The order of precedence, from high to low, is `not`, `and`, `or`.
- This statement is equivalent to
- `(a or ((not b) and c))`
- Since most people don't memorize the Boolean precedence rules, use parentheses to prevent confusion.

Boolean Operators

- To test for the co-location of two points, we could use an and.

```
if p1.getX() == p2.getX() and p2.getY() == p1.getY():
    # points are the same
else:
    # points are different
```

- The entire condition will be true only when both of the simpler conditions are true.

Boolean Operators

- Say you're writing a racquetball simulation. The game is over as soon as either player has scored 15 points.
- How can you represent that in a Boolean expression?

```
scoreA == 15 or scoreB == 15
```

- When either of the conditions becomes true, the entire expression is true. If neither condition is true, the expression is false.

Boolean Operators

- We want to construct a loop that continues as long as the game is not over.
- You can do this by taking the negation of the game-over condition as your loop condition!

```
while not(scoreA == 15 or scoreB == 15):
    #continue playing
```

Boolean Operators

- Some racquetball players also use a shutout condition to end the game, where if one player has scored 7 points and the other person hasn't scored yet, the game is over.

```
while not((scoreA == 15 or scoreB == 15) or \
          (scoreA == 7 and scoreB == 0) or \
          (scoreB == 7 and scoreA == 0)):
    #continue playing
```

Boolean Operators

- Let's look at volleyball scoring. To win, a volleyball team needs to win by at least two points.
- In volleyball, a team wins at 15 points
- If the score is 15 – 14, play continues, just as it does for 21 – 20.

```
(a >= 15 and a - b >= 2) or (b >= 15 and b - a >= 2)
(a >= 15 or b >= 15) and abs(a - b) >= 2
```

Boolean Algebra

- The ability to formulate, manipulate, and reason with Boolean expressions is an important skill.
- Boolean expressions obey certain algebraic laws called Boolean logic or Boolean algebra.

Boolean Algebra

- `and` has properties similar to multiplication
- `or` has properties similar to addition
- `0` and `1` correspond to false and true, respectively.

Algebra	Boolean Algebra
a * 0 = 0	a and false = false
a * 1 = a	a and true = a
a + 0 = a	a or false = a

Boolean Algebra

- Anything ored with true is true:
  - `a or true == true`
- Both and and or distribute:
  - `a or (b and c) == (a or b) and (a or c)`
  - `a and (b or c) == (a and b) or (a and c)`
- Double negatives cancel out:
  - `not(not a) == a`
- DeMorgan's laws:
  - `not(a or b) == (not a) and (not b)`
  - `not(a and b) == (not a) or (not b)`

Boolean Algebra

- We can use these rules to simplify our Boolean expressions.

```
while not(scoreA == 15 or scoreB == 15):
    #continue playing
```

- This is saying something like "While it is not the case that player A has 15 or player B has 15, continue playing."
- Applying DeMorgan's law:

```
while (not scoreA == 15) and (not scoreB == 15):
    #continue playing
```

Boolean Algebra

- This becomes:

```
while scoreA != 15 and scoreB != 15
    # continue playing
```

- Isn't this easier to understand? "While player A has not reached 15 and player B has not reached 15, continue playing."

Boolean Algebra

- Sometimes it's easier to figure out when a loop should stop, rather than when the loop should continue.
- In this case, write the loop termination condition and put a not in front of it. After a couple applications of DeMorgan's law you are ready to go with a simpler but equivalent expression.

Boolean Expressions as Decisions

- Boolean expressions can be used as control structures themselves.
- Suppose you're writing a program that keeps going as long as the user enters a response that starts with 'y' (like our interactive loop).
- One way you could do it:

```
while response[0] == "y" or response[0]
== "Y":
```

Boolean Expressions as Decisions

- Be careful! You can't take shortcuts:

```
while response[0] == "y" or "Y":
```

- Why doesn't this work?

By the operational description of `or`, this expression returns either `True` (returned by `==` when `response[0]` is `"y"`) or `"Y"` (when `response[0]` is not `"y"`). Either of these results is interpreted by Python as true.

A more logic-oriented way to think about this is to simply look at the second expression. It is a nonempty string, so Python will always interpret it as true. Since at least one of the two expressions is always true, the `or` of the expressions must always be true as well.

Post-Test Loop

Say we want to write a program that is supposed to get a nonnegative number from the user.

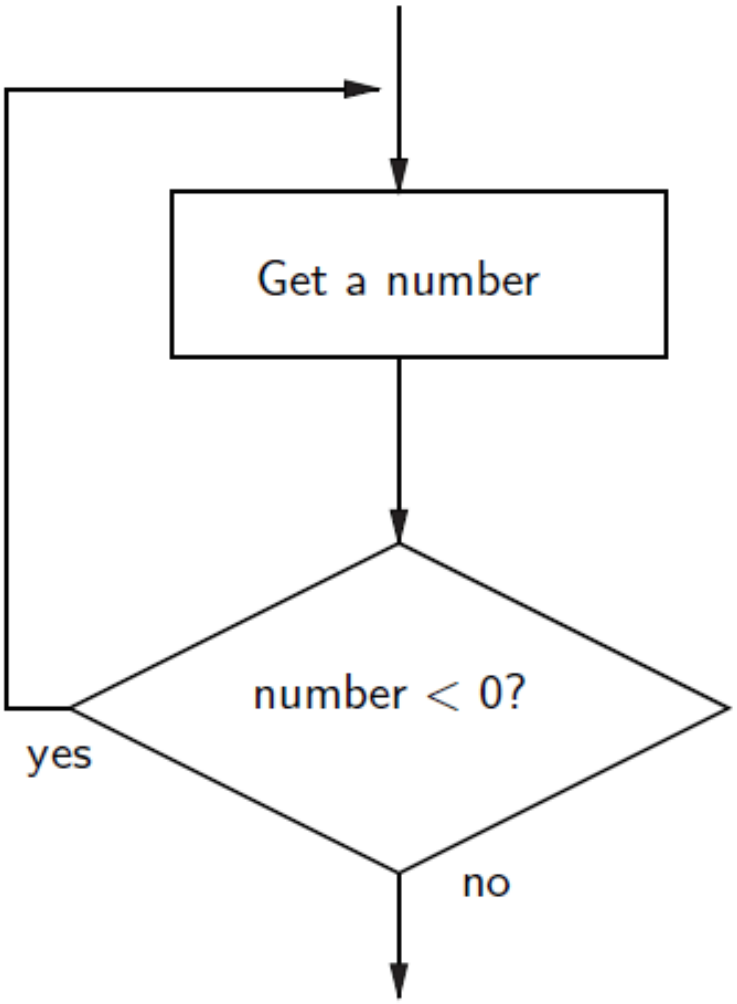
If the user types an incorrect input, the program asks for another value.

This process continues until a valid value has been entered.

This process is *input validation*.

Post-Test Loop

```
repeat
    get a number from the user
until number is >= 0
```



Post-Test Loop

When the condition test comes after the body of the loop it's called a *post-test loop*.

A post-test loop always executes the body of the code at least once.

Python doesn't have a built-in statement to do this, but we can do it with a slightly modified `while` loop.

Post-Test Loop

We seed the loop condition so we're guaranteed to execute the loop once.

```
number = -1          # start with an illegal value
while number < 0: # to get into the loop
    number = float(input("Enter a positive number: "))
```

By setting `number` to –1, we force the loop body to execute at least once.

Post-Test Loop and Break

Some programmers prefer to simulate a post-test loop by using the Python `break` statement.

Executing `break` causes Python to immediately exit the enclosing loop.

`break` is sometimes used to exit what looks like an infinite loop.

Post-Test Loop and break

The same algorithm implemented with a `break`:

```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0: break # Exit loop if number is valid
```

A while loop continues as long as the expression evaluates to true. Since `True` *always* evaluates to true, it looks like an infinite loop!

Post-Test Loop and break

Adding the warning to the `break` version only adds an `else` statement:

```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0:
        break # Exit loop if number is valid
    else:
        print("The number you entered was not positive.")
```

When use Break

To use or not use `break`. That is the question!

The use of break is mostly a matter of style and taste.

Avoid using break often within loops, because the logic of a loop is hard to follow when there are multiple exits.

Exercise

Give a truth table that shows the Boolean vaue of each of the following boolean expressions, for every possible combination of “input” values.

Hint: Including columns for intermediate expressions is helpful.

a> not (P and Q)

b> not(P) and Q