# ICS3U: Logical Operators

## Logical Operators: `and`, `or` and `not`

There are three **logical operators** in Python: `and`, `or` and `not`. Logical operators are used with boolean values, and since conditions evaluate to either `True` or `False`, they are often used with `if` statements to compare conditions. Both `and` and `or` compare two boolean values, while `not` uses only one.

A **truth table** is a chart that shows the results of using a logical operator. Truth tables for `and`, `or` and `not` are shown below.

| $B_1$ | $B_2$ | $B_1$ `and` $B_2$ |
|---|---|---|
| `True` | `True` | `True` |
| `True` | `False` | `False` |
| `False` | `True` | `False` |
| `False` | `False` | `False` |

| $B_1$ | $B_2$ | $B_1$ `or` $B_2$ |
|---|---|---|
| `True` | `True` | `True` |
| `True` | `False` | `True` |
| `False` | `True` | `True` |
| `False` | `False` | `False` |

| B | `not` B |
|---|---|
| `True` | `False` |
| `False` | `True` |

Both boolean values $B_1$ and $B_2$ must be `True` in order for $B_1$ `and` $B_2$ to be `True`. In all other cases, at least one of $B_1$ or $B_2$ is `False`, and so $B_1$ and $B_2$ is `False`.

If either $B_1$ or $B_2$ are `True` (or possibly both), then $B_1$ `or` $B_2$ is `True`. The only case in which $B_1$ `or` $B_2$ is `False` occurs when both $B_1$ and $B_2$ are `False`.

The truth table for `not` is much simpler, as it only deals with one boolean value. If a boolean value B is `True`, then `not` B is False. Similarly, if B is `False`, then `not` B is `True`.

## Using Logical Operators In Python

Consider a program that reads a letter from the user and displays a message if it is either an uppercase or lowercase ‘A’. Such a program might look like this.

```
letter = input("Please enter a letter: ")
if letter == "A":
   print("You entered A.")
elif letter == "a":
   print("You entered A.")
else:
   print("You did not enter A.")
```

Instead of using both `if` and `elif` statements to cover the uppercase and lowercase values, we might rewrite the program using `or` like this.

```
letter = input("Please enter a letter: ")
if letter == "A" or letter == "a":
   print("You entered A.")
else:
   print("You did not enter A.")
```

Assume that the user enters the letter 'a' (lowercase). The interpreter examines the first condition, `letter == "A"`, which is `False`. It is followed by `or`, so the interpreter examines the second condition, `letter == "a"`, which is `True`. Since one of the conditions was `True`, the entire expression evaluates as `True`, and the code inside of the `if` block is executed.

This is a good approach for two reasons. First, we have a logical association with the uppercase and lowercase values of A in the English language, so grouping them makes sense. Second, the resulting action (a `print` statement) is identical for either letter. Grouping them eliminates this redundancy.

When two or more conditions are linked via a logical operator, the Python interpreter employs **short-circuiting** when evaluating. In the case of `or`, this means that if the first condition is `True`, then it does not matter whether the second condition is `True` or `False`, as the two will evaluate as `True` overall. In the case of `and`, then if the first condition is `False`, it will not matter whether the second condition is `True` or `False`, as the two will evaluate as `False`. The interpreter will *not* evaluate the second condition, as it will only waste time.

For example, consider the case where the user enters 'A' (uppercase) in the earlier example. First, the condition `letter == "A"` is evaluated and found to be `True`. Since this is followed by `or`, the value of the second condition, `letter == "a"`, is irrelevant. The interpreter will immediatly execute the code inside of the `if` block.

# Logical Operators vs. `elif` and Nested `if` Statements

In the last example, we used `or` to replace an `elif` statement. Since both `elif` and `or` provide alternate choices, it is often possible to substitute one for the other if the resulting action is the same. Similarly, `and` can often take the place of a nested `if` statement. MORE.

Below is code that asks the user to enter an integer value within a fixed range. If number is between 10 and 20, a message is displayed. Alternate messages are displayed if the number is outside of the range, or if the number is not between 10 and 20.

```
num = int(input("Enter an integer between 1 and 100: "))
if num >= 10:
   if num <= 20:
      print("The number is between 10 and 20.")
elif num < 0:
   print("Number is out of bounds.")
elif num > 100:
   print("Number is out of bounds.")
else:
   print("Number is not between 10 and 20.")
```

The previous code, rewritten to use logical operators instead of nested `if` and multiple `elif` statements, is below. It is shorter, and probably easier to read and understand.

```
num = int(input("Enter an integer between 1 and 100: "))
if num >=10 and num <= 20:
   print("The number is between 10 and 20.")
elif num < 0 or num > 100:
   print("Number is out of bounds.")
else:
   print("Number is not between 10 and 20.")
```

# "Truth" In Python

Beginning Python programmers often stumble upon an interesting logical error when they first start using logical operators in `if` statements. Try running the following code, using a variety of values.

```
letter = input("Please enter a letter: ")
if letter == "x" or "X":
   print("You entered X.")
else:
   print("You did not enter X.")
```

It does not matter what the user types: the program will *always* indicate that the user has entered the letter 'X'. This might seem strange, but it is actually by design. Before we can understand *why* this happens, it helps to know in greater detail how Python tests for truth.

When an `if` statement is encountered, Python checks a condition that follows it. Usually this is stated *explicitly*, like below.

```
if play_again == True:
   print("OK, starting a new game.")
   ...
```

But Python also allows for *implicit* truth testing. An equivalent expression might look like this.

```
if play_again:
   print("OK, starting a new game.")
   ...
```

Note that there is no comparison operator or value to compare after the variable name. When only a variable (or other object) is present, Python interprets the variable (or object) itself as either `True` or `False`. For a boolean data type like `play_again`, this makes sense, but other data types like integers and strings can also be tested for truth implicitly.

Some of the values that are interpreted as `False` include: 0, 0.0, `""` (empty string), () (empty tuple), [] (empty list), and `None`. There are others too, but we will probably not encounter them in this course. On the other hand, values like 7, 3.5, "hello", (4, 3), and [1, -2, 5] are interpreted as `True`. In general, any non-zero, non-empty value is `True` in Python. This is why the program involving the letter 'X' resulted in a logical error: the two conditions being checked were `letter == "x"` and `"X"`. Since 'X' is a non-zero, non-empty value, it is evaluated implicitly as `True`. To prevent the logic error, we need to explicitly state the comparison in the condition.

```
if letter == "x" or letter == "X":
   ...
```