Lecture 7 - Boolean operators

Computer Programming

Robert D. Vincent and Samia Hilal

Marianopolis College

March 9, 2022

Boolean algebra

- As previously mentioned, Boolean values can have only one of two values, `True` or `False`.
- Useful for representing program conditions.
- Important in `while` and `if` conditions.
- There are three basic Boolean operators:
  - *Negation* (commonly called `not`)
  - *Conjunction* (commonly called `and`)
  - *Disjunction* (commonly called `or`)
- Conveniently, Python uses the reserved words `not`, `and`, and `or` for these operations.

Truth tables

- A *truth table* is often used to represent Boolean operations.
- The output is represented in the last column.
- The inputs in the first column(s).
- The simplest truth table is that of `not` or negation:

P	not P
True	False
False	True

Truth table for `and` (conjunction)

P	Q	P and Q
True	True	True
False	True	False
True	False	False
False	False	False

Truth table for `or` (disjunction)

P	Q	P or Q
True	True	True
False	True	True
True	False	True
False	False	False

Boolean algebraic rules

- Commutative:

```
P and Q == Q and P
P or Q == Q or P
```

- Distributive:

```
A and (B or C) == (A and B) or (A and C)
A or (B and C) == (A or B) and (A or C)
```

- Associative:

```
(A or B) or C == A or (B or C)
(A and B) and C == A and (B and C)
```

- Double negation:

```
not not P == P
```

Other useful rules to remember

- Negating a comparison:

```
not (A > B) == A <= B
not (A < B) == A >= B
```

- Exclusive or:

P	Q	P xor Q
True	True	False
False	True	True
True	False	True
False	False	False

De Morgan's laws

- It is often useful to remember that:

```
not (P and Q) == (not P) or (not Q)
not (P or Q) == (not P) and (not Q)
```

- For example:

```
not (n != 0 and m <= 100)
```

- is equivalent to:

```
(not n != 0) or (not m <= 100)
```

- which simplifies to:

```
n == 0 or m > 100
```

Boolean expressions in Python

- Python has three Boolean operators, two binary and one unary.
- *expression* `and` *expression*
- *expression* `or` *expression*
- `not` *expression*

```
>>> a = True
>>> b = False
>>> print(a and b)
False
>>> print(a or b)
True
>>> print(not a)
False
```

Boolean expressions

- Can combine these in complex expressions:

```
>>> a, b, c = 1, 2, 3
>>> a < 0 and b < 0
False
>>> b > 1 and c > 1
True
>>> a < 0 and b < 0 or b > 1 and c > 1
True
```

- Use parentheses with complex expressions:

Boolean operator hierarchy

- Highest precedence: `not`
- Middle precedence: `and`
- Lowest precedence: `or`

```
>>> a,b,c = True,True,False
>>> print(a or b and c)
True
>>> print((a or b) and c)
False
```

- Never hurts to use parentheses!

What is `True`?

- In Python, many different values are interpreted as `False`:
  - Any sequence (string, tuple, or list) of length zero.
  - The numbers 0 or 0.0
  - `None`
  - `False`
- This rule is often exploited in conditional expressions in `if` or `while`.
- The builtin function `bool()` will convert any type to `True` or `False`.

What is True?

```
>>> bool('') # empty string
False
>>> bool([]) # empty list
False
>>> bool(0.0)
False
>>> bool(-0.5)
True
>>> items = [5, 9]
>>> while items: # true until list empty
...     print(items.pop())
...
9
5
```

Short-circuit evaluation

- In most programming languages, Boolean operators implement *short-circuit evaluation*.
- Evaluates right side of a Boolean expression only if needed.
- Consider this statement:

```
if A and B:
```

- If `A` is `False`, the whole expression is `False` for any value of `B`.
- So Python doesn't even evaluate `B`!

Short-circuit evaluation, continued

- Compare with `or`:

```
if A or B:
```

- If `A` is `True` there is no need to evaluate `B`.

```
a, b = 1, 2
# Here a > 0 is True, so
# b > 0 is not evaluated!
if a > 0 or b > 0:
    print("Good day, Madam!")
```

- Becomes significant when we have code with *side effects*.

Some style notes

- Don't compare with `False` or `True`

```
doNext = (x < xMax)
if doNext == True: # Not good style
    x += 1

if doNext == False: #
    print("Finished.")
```

- Instead write:

```
if doNext:              # Preferred
    x += 1

if not doNext:
    print("Finished.")
```

More examples

- Instead of:

```
if a == False and b == True:
```

- Write:

```
if not a and b:
```

- Instead of:

```
if not (a == False or b == False):
```

- Write:

```
if a and b:
```

Boolean sequence functions

- `any()` - True if any element is True.
- `all()` - True if all elements are True.

```
>>> L = [3, 4, 9, 0]
>>> print(all(L))
False
>>> print(any(L))
True
>>> T = (0, 0, 0)
>>> print(any(T))
False
>>> X = [True, True, True]
>>> print(all(X))
True
```

Summary

- Boolean expressions can be created with `and`, `or`, and `not`.
- Rules of Boolean algebra can be used to help simplify and understand these expressions.
- Most Python values have a Boolean "interpretation".
  - The number zero is `False`
  - An empty string, tuple, or list is `False`.
  - None is False!
  - Anything else is `True`!