Operators and Condition Evaluation
Relational and Logical Operators

- A *condition* is an expression
  - Involving relational operators (such as < and >=)
  - Logical operators (such as and, or, and not)
  - Evaluates to either True or False
- Conditions used to make decisions
  - Control loops
  - Choose between options

Relational Operators

- Relational operator *less than* (<) can be applied to
  - Numbers
  - Strings
  - Other objects
- For strings, the ASCII table determines order of characters

Relational Operators

Python Notation	Numeric Meaning	String Meaning
==	equal to	identical to
!=	not equal to	different from
<	less than	precedes lexicographically
>	greater than	follows lexicographically
<=	less than or equal to	precedes lexicographically or is identical to
>=	greater than or equal to	follows lexicographically or is identical to
in		substring of
not in		not a substring of

Relational operators

ASCII Values

- ASCII values determine order used to compare strings with relational operators.
- Associated with keyboard letters, characters, numerals
  - ASCII values are numbers ranging from 32 to 126.
- A few ASCII values.

32 (space)	48 0	66 B	122 z
33 !	49 1	90 Z	123 {
34 “	57 9	97 a	125 }
35 #	65 A	98 b	126 ~

ASCII Values

- The ASCII standard also assigns characters to some numbers above 126.
- A few of the higher ASCII values.

162 ¢	177 ±	181 µ	190 ¾
169 ©	178 ²	188 ¼	247 ÷
176 °	179 ³	189 ½	248 ø

- Functions **`chr(n)`** and **`ord(str)`** access ASCII values

Relational Operators

- Example: Determine whether following conditions evaluate to *True* or *False*

(a) $1 <= 1$
(b) $1 < 1$
(c) "car" < "cat"
(d) "Dog" < "dog"
(e) "fun" in "refunded"

Relational Operators

- Example: Determine whether following conditions evaluate to *True* or *False*

  (a=4, b=3, c="hello", d="bye")

(a) (a + b) < (2 * a)

(b) (len(c) − b) == (a/2)

(c) c < ("good" + d)

- *len() is a function that returns the length of a string*

Relational Operators

- Some rules
  - An int *can* be compared to a float.
  - Otherwise, values of different types *cannot* be compared
  - **Relational operators can be applied to lists or tuples

```
[3, 5] < [3, 7]
[3, 5] < [3, 5, 6]
[3, 5, 7] < [3, 7, 2]
[7, "three", 5] < [7, "two", 2]
```

Logical Operators

- Based on Boolean algebra (input values are boolean rather than numbers)
- In mathematics and mathematical logic, Boolean algebra is the branch of algebra in which the values of the variables are the truth values true and false, usually denoted 1 and 0 respectively
- Elementary algebra works with the values of the variables that are numbers, and the prime operations are addition and multiplication
- Boolean algebra works with 0s and 1s as input values, the main operations of Boolean algebra are the conjunction (and) denoted as ∧, the disjunction (or) denoted as ∨, and the negation (not) denoted as ¬.
- Boolean algebra is a formalism for describing logical operations in the same way that elementary algebra describes numerical operations.

Logical Operators

- Logical operators enable combining conditions which are likely to use relational operators
- Logical operators are the reserved words **`and`**, **`or`**, **`not`**
- Conditions that use these operators are called compound conditions

Logical Operators

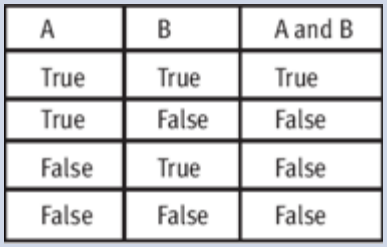
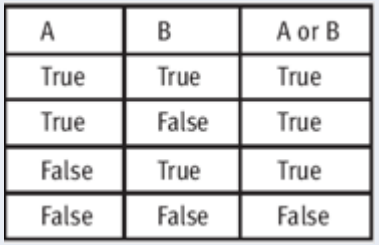
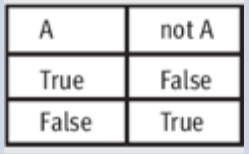
- Given: *cond1* and *cond2* are conditions
  - **`cond1 and cond2`** true only if both conditions are true

    Example: (a == 3 and b == 5)
  - **`cond1 or cond2`** true if either or both conditions are true

    Example: (a == 3 or c < 10)
  - **`not cond1`** is false if the condition is true, true if the condition is false

    Example: not(a < b)

Logical Operators

Operator	Truth Table
AND	
OR	
NOT	

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

Operator Precedence

- Operator Precedence from Highest to Lowest

TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	`**`
Arithmetic negation	`-`
Multiplication, division, remainder	`*`, `/`, `%`
Addition, subtraction	`+`, `-`
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical negation	`not`
Logical conjunction and disjunction	`and`, `or`
Assignment	`=`

Logical Operators

- Example: Given n = 4, answ = “Y” Determine expressions = *True* or *False*

(a) (2 < n) and (n < 6)
(b) (2 < n) or (n == 6)
(c) not (n < 6)
(d) (answ == “Y”) or (answ == “y”)
(e) (answ == “Y”) and (answ == “y”)
(f) not (answ == “y”)
(g) ((2 < n) and (n == 5 + 1)) or (answ == “No”)
(h) ((n == 2) and (n == 7)) or (answ == “Y”)
(i) (n == 2) and ((n == 7) or (answ == “Y”))

The `bool` Data Type

- Objects True and False are said to have Boolean data type
  - Of data type **`bool`**.
- What do these lines display?

```
x = 2
y = 3
var = x < y
print(var)
```

```
x = 5
print((3 + x) < 7)
```

Simplifying Conditions

Use of De Morgan's Laws – allows us to simplify complementing a logical expression

- **How do we express that x is between the range 10 (not including 10) and 20 (including 20)?**

```
x > 10 and x <= 20
```

- **Suppose we want to find the complement of this condition i.e. x is not within this specified range**

```
not(x > 10 and x <= 20)
```

Simplifying Conditions

Using De Morgan's laws:

(not(cond1 and cond2)

is the same as

not(cond1) or not(cond2)

i.e. complement each individual condition and complement the logical operator joining the two conditions

```
not(x > 10 and x <= 20)  --->  x <= 10 or x > 20
```

- **Similarly:**

not(cond1 or cond2)

is the same as

not(cond1) and not(cond2)

Simplifying Conditions

Also note that Python supports this syntax:

```
(x > 10) and (x <= 20)
```

can be replaced with the condition

```
10 < x <= 20
```

```
(x <= 10) or (x > 20)
```

can be replaced with the condition

```
not(10 < x <= 20)
```

Short-Circuit Evaluation

- Consider the condition **`cond1 and cond2`**
  - If Python evaluates **`cond1`** as false, it does not bother to check **`cond2`**
- Similarly with **`cond1 or cond2`**
  - If Python finds **`cond1`** true, it does not bother to check further
- Think why this feature helps for

```
(number != 0) and (m == (n / number))
```