CptS 111, Spring 2023
Lect. #9, Feb. 8, 2023
Class Notes

Today's Agenda:

1. Relational/Comparison operators revisited
2. Boolean/Logical operators and expressions
3. Operator chaining
4. Nested conditionals
5. Precedence

Ch. 4 (cont.)
1. Relational/Comparison Operators Revisited

On Monday, we discussed the six comparison/relational operators used in test expressions to compare two operands:

x < y	Is x less than y?
x <= y	Is x less than or equal to y?
x == y	Is x equal to y?
x >= y	Is x greater than or equal to y?
x > y	Is x greater than y?
x != y	Is x not equal to y?

When used in a test expression, these result in one of the Boolean variables `True` or `False`. Note that floats and integers can be equal to each other:

In [1]:
```
# Integers and floats can be equal

1 == 1.0
```

Out[1]:
```
True
```

However, as I mentioned on Monday, it's not a good idea to compare float values to each other because the value of a float may not be what you think it is, which is usually a result of the finite precision of floats. Consider the following two examples:

In [2]:
```
# It's not a good idea to test for equality between two floats
# First, multiply 0.1 and 10 which should result in 1

x = 0.1
y = 10 * x
y == 1.0
```

Out[2]:
```
True
```

In [3]:
```
# Second, add 0.1 ten times; this should also result in 1
# Sum of 10 x's.

z = (x + x + x + x + x + x + x + x + x + x)  # Add 0.1 10 times
z == 1.0
```

Out[3]:
```
False
```

In [4]:
```
# What happened? Adding 0.1 ten times and multiplying 0.1 by 10
# should both give 1

print('0.1 * 10 =', y)
print('0.1 summed 10 times =', z)
```

```
0.1 * 10 = 1.0
0.1 summed 10 times = 0.9999999999999999
```

In the first example, the value of `y` is 1.0 as expected, but in the second example, the value of `z` isn't quite 1.0 because of the way floats are handled.

Comparing integers is straightforward. For example,

In [5]:
```
# Is the same true for integers?  2 + 2 = 4!

int1 = 4
int2 = 2 + 2
int1 == int2
```

Out[5]:
```
True
```

In [6]:
```
# 10 * 4 = 40, but what about if we add 4 ten times?

int3 = 10 * int1
int4 = int2 + int2 + int2 + int2 + int2 + int2 + int2 + int2 + int2 + int2
int3 == int4
```

Out[6]:
```
True
```

Because decimal integers can be written exactly in terms of binary sums, integers don't have rounding errors!

Comparing strings is a little bit more complicated. Recall from Ch. 3 that ASCII characters (how many are there?) can be equated to their decimal and binary values, e.g., `A` is `65` and `a` is `97`. Also, `A < Z`, and `a < z`, but what about `a > Z`?

In [7]:
```
# 'a' is less that 'z', but is 'a' > 'Z'?

if 'a' < 'z':
    print('a is < z')
if 'a' > 'Z':
    print('a is > Z')
```

```
a is < z
a is > Z
```

In [8]:
```
# Of course, this all has to do with the decimal values of ASCII
# characters; any lowercase letter is larger than any uppercase
# letter!

print('a is', ord('a'))  # Arg of ord() is a single character
print('z is', ord('z'))
print('Z is', ord('Z'))
print('Thus, a < z (', ord('a'), ' < ', ord('z'), '),', sep='')
print('but a > Z (', ord('a'), ' > ', ord('Z'), ').', sep='')
```

```
a is 97
z is 122
Z is 90
Thus, a < z (97 < 122),
but a > Z (97 > 90).
```

Uppercase letters are smaller than lowercase letters! Because of this, we can even compare lists of strings.

In [10]:
```
# Compare two different lists

list1 = ['a', 'b', 'c']
list2 = ['a', 'b', 'D']
if list1 > list2:
    print("['a', 'b', 'c'] > ['a', 'b', 'D']")
```

```
['a', 'b', 'c'] > ['a', 'b', 'D']
```

2. Boolean/Logical Operators and Boolean Expressions

There are three Boolean operators: `and`, `or`, and `not`. The first two are used frequently; `not` must be used with care. `not` has higher precedence than `and` which has higher precedence than `or`, i.e.,

```
not > and > or
```

- `and` requires ***both*** Boolean/logical expressions to be `True` to test `True`
- `or` requires ***only one*** Boolean/logical expression to be `True` to test `True`

Note: If the first test expression in a Boolean expression using `and` is false, the second test expression won't be evaluated. Similarly, if the first test expression in a Boolean expression using `or` is true, the second test expression won't be evaluated. ***Remember this because it can be the source of errors***. For example:

In [45]:
```
# We can't divide by 0, but x / 0 isn't evaluated!

x = 5
if x < 10 or x / 0:
    print('x < 10')
```

```
x < 10
```

In [46]:
```
# If we reverse the comparisons, let's see what happens

x = 5
if x / 0 or x < 10:
    print('x < 10')
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [46], in <cell line: 4>()
      1 # If we reverse the comparisons, let's see what happens
      3 x = 5
----> 4 if x / 0 or x < 10:
      5     print('x < 10')

ZeroDivisionError: division by zero
```

In [47]:
```
# Again, if the first test expression is wrong when 'and' is used,
# Python will skip the second one because the two evaluate to false

x = 10
if x != 10 and x / 0:
    print("You can't divide by zero.")
else:
    print('x is 10')
```

```
x is 10
```

Sometimes the Boolean/logical `not` operator is useful, but often we can avoid using it to make our code more readable. For example, consider the following:

In [48]:
```
# The Boolean 'not' can be confusing; here we test to see whether x is
# greater than 3

x = 1
if not x <= 3:
    print('x > 3')
```

In [49]:
```
# 'x > 3' is equivalent to 'not x <= 3'
# This is easier for me to understand

if x > 3:
    print('x > 3')
```

In [50]:
```
# The use of parentheses here means that the expression evaluates to
# True if x is less than or equal to 7 or x is greater than or equal to 0

if not (x > 7 or x < 0):
    print('x <= 7 and x >= 0' )
```

```
x <= 7 and x >= 0
```

In [51]:
```
# We can use a Boolean 'and' expression to get an expression equivalent
# to the 'not(x > 7 or x < 0)'
# This is easier for me to understand

if x <= 7 and x >= 0:         # Use parentheses to make this more readable
    print('x <=7 and x >= 0')
```

```
x <=7 and x >= 0
```

3. Operator Chaining

Python allows us to combine comparison expressions easily using operator chaining. Using operator chaining,

```
if x > 5 and x < 10:
```

can be written simply as:

```
if 5 < x < 10:
```

Operator chaining isn't common in programming languages. It's another example of how Python makes programming easier.

In [52]:
```
# Last example using operator chaining
# This is even easier for me to understand!

if 0 <= x <= 7:
    print('x <= 7 and x >= 0')
```

```
x <= 7 and x >= 0
```

In [53]:
```
# Another example of operator chaining

grade_in_hs = 10
if 9 <= grade_in_hs <= 12:
    print("She's a high school student.")
```

```
She's a high school student.
```

Note that in the example above, we could have also used the following operator chaining:

In [54]:
```
# This is an equivalent expression

if 12 >= grade_in_hs >= 9:
    print("She's a high school student.")
```

```
She's a high school student.
```

4. Nested Conditionals

We can nest conditionals as many times as needed:

```
if <test expression>:
    if <test expression>:
        if <test expression>:
            <code>
        else:
            <code>
    else:
        <code>
else:
    <code>
```

As you read in your zyBook, you must keep track of the correct indentation in your blocks of code!

In [55]:
```
# Nested conditionals

day = input('Enter day of the week: ')
if day == 'Saturday' or day == 'Sunday':
    temp_forecast = int(input('Enter the temperature forecast [integer]: '))
    weather_forecast = input('Enter the weather_forecast [clear, rainy]: ')
    if 60 <= temp_forecast <= 75 and weather_forecast == 'clear':
        print("Let's go for a hike!")
    elif temp_forecast < 60 and weather_forecast == 'rainy':
        print("I'm going read a book.")
    else:
        print("I'm open to suggestions.")
else:
    print('Go to work or to school!')
```

```
Enter day of the week: Sunday
Enter the temperature forecast [integer]: 60
Enter the weather_forecast [clear, rainy]: clear
Let's go for a hike!
```

5. Precedence

Recall that in arithmetic expressions, certain operators have precedence over others, e.g., multiplication has higher precedence than addition. When we combine arithmetic expressions with comparison and logical expressions, we also need to know the rules of precedence: Arithmetic operators have precedence over comparison/relational operators which have precedence over Boolean/logical operators. ***Remember this!***

arithmetic > comparison/relational > Boolean/logical

When in doubt, use parentheses!

In [23]:
```
(8 * 7 > 8 * 6) and (9 * 10 < 8 * 10)
```

Out[23]:
```
False
```

In the example above, the math calculations are performed first, then the comparison operations, and finally the logical (`and`) operation. Thus the progression is:

```
(56 > 48) and (90 < 80)     # First arithmetic operations
(True) and (False)          # Next comparison operations (> and <)
False                       # Finally, logical operations (and)
```

Where we recall that in order for the `and` operator to give a `True` result, both operands must be `True` .