CSC110 Supplemental Reading: Week03

Contents

1 Boolean Expressions 2

1.1.1 Relational Operators 2

1.2 Testing Numbers 3

1.3 Testing floating point numbers 3

1.4 Testing Strings 3

1.5 Precedence of Relational Operators 5

1.6 Compound Logical Expressions 5

1.7 Common Errors Writing Compound Logical Expressions 7

2 The if statement 8

2.1 The Simple if Statement 8

2.2 The if...else Statement 9

2.3 Nested if Structures 10

2.4 Testing Programs with Selection Structures 13

1 Boolean Expressions

Our goal now is to create programs that make **decisions**. Here is a very simple example:

```
num = float( input ("Please enter a number") )
if num > 0:
  print ( "The number is greater than zero" )
```

This code tests the value of **num**... if it is greater than zero, it will display a message. If **num** is less than or equal to zero, no message is displayed.

The if statement contains a **boolean expression**. A boolean expression is a combination of constants, variables, operators, and method or function calls that simplifies to a single **boolean value**. There are only 2 possible boolean values: **True** and **False**.

Here are some more examples of boolean expressions:

Boolean Expressions
`x > 17`
`userName == "David"`
`size <= 1.3`
`2 * weight != 14.7`

1.1.1 Relational Operators

Boolean expressions contain **relational operators**. Here are the relational operators in Python:

Relational Operators	Meaning
==	**equality**
<	**less than**
>	**greater than**
<=	**less than or equal to**
>=	**greater than or equal to**
!=	**not equal**

1.2 Testing Numbers

When relational operators are used to compare numbers, the results are exactly what you would expect:
Given:

```
x = 2
y = 1
```

Expression	Value
x > y	True
x < y	False
x < 2	False
x <= 2	True
x == 2	True
y == x	False
x != y	True
x != 2	False

1.3 Testing floating point numbers

Because of the way that floating point numbers, numbers with decimals, are stored in a computer, there is the chance for some slight representation error. For example, if you try `4.2 * 0.1` in the Python shell, you'll see that the value is not exactly 0.42. Also try `0.1 + 0.2` and see what you get. (To learn more about floating point number representation inside a computer, check this out)

Consequently, testing for equality may not work out as you'd expect. Instead, check if the value is near the expected value, +/- some small change. We can use the abs() function to help with this.

```
calculatedResult = 0.1 + 0.2

expectedResult = 0.3

abs(calculatedResult - expectedResult) < 0.00005
```

This last expression will be True if the difference between the calculatedResult and expectedResult are with 0.00005 of each other.

1.4 Testing Strings

With strings, it works a bit differently. Strings are compared character by character, beginning with the first character in the string. Equality is simple -- the strings must be *exactly* equal. Upper- and lower-case letters are different; spaces and any other characters need to be considered as well.

- `"Go" == "Go"` is True
- `"Go" == "Go "` is False (because of the extra space in the second string)
- `"Go" == "go"` is False (because of casing)

Testing string inequalities (like < or >) uses the ASCII code (discussed in chapter 1). There is no need to memorize any character codes, although you can read about the ASCII table here. However, you do need to remember some features of the ASCII code set:

- The space character has a very low numeric code (32) below all letters and digits
- The upper case letters are sequential. For example, the code for "A" is 65, the code for "B" is 66, the code for "C" is 67, etc
- The lower case letters are sequential and this set comes AFTER the upper case letters. For example, the code for "a" is 97, the code for "b" is 98, etc. Notice how these codes are larger than the codes for the uppercase characters.
- The digits are sequential. The code for the character "0" is 48, the code for the character "1" is 49,etc. **Be careful!** We are talking characters here, "0" and "1", not the numbers 0 and 1.

The value of the numeric codes that represent the characters are used when testing strings. Applying this knowledge, we can evaluate some string inequalities:

`"X" > "A"` is True (because the code for "X" is bigger than the code for "A")
`"3" > "0"` is True (because the code for "3" is bigger than the code for "0")
`"b" > "W"` is True (because all the lower-case letters have higher codes than any upper-case letter)

Now let's look at longer strings. If the characters in a given position match, then the comparison moves to the next position. So...

`"AB" > "AA"` is True. The first characters, "A", are the same, so the test must go to the second characters. "B" is larger than "A" so the entire expression is true.

`"ABGH" > "ABFZK"` is True. The first 2 characters are the same so the decision happens at the 3rd letter. The "G" is larger than the "F", so the first string is lexicographically larger (regardless of what letters come afterwards). Notice that this has nothing to do with the length. Though the second string is longer, it is still 'less than' the first string.

also, `"CAT " > "CAT"` is True. The first string has a space after the T, the second string does not. Since the second string has no character after the 'T', the absence of a character is always 'less than' any character.

Consider the following examples:

`"DOG" > "DOB"` is True because "G" is greater than "B"
`"Dog" == "dog"` is False because "D" is not equal to "d"
`"Barb" > "apple"` is False because capital B has a lower numeric code than lowercase a. Again, this doesn't depend on the length of the string at all.

1.5 Precedence of Relational Operators

Here you see an arithmetic expression and a boolean expression. Given:
height = 3.0
length = 36.0

	Arithmetic Expression	Boolean Expression
Expression	`4 * height - math.sqrt( length ) / 3`	`12 * height > length + 2`
Value of the Expression	`10.0`	`False`

We are familiar with arithmetic expressions and the order of operators. Now let's look at the boolean expression. There are actually 3 operators there: *, > and +. Which one goes first? As a rule, all arithmetic operations happen first, then the relational operators. Here are some more logical expressions and their values, given these assignments:

n = 4
word = "hello"

Expression	Value
`n + 20 > 17`	True
`n * 4 != n ** 2`	False
`len(word) >= 5`	True
`"help" > "hear"`	True
`"bottle" <= "bOy"`	False
`"candy" == "Candy"`	False

1.6 Compound Logical Expressions

You can also combine multiple boolean tests into one expression, known as a compound expression. For this you need to use **logical operators**. Here are some examples:

Compound Expressions

`x == 17 or x > 23`
`userName == "David" and userAge < 21`
`not (size == 1.3)`

There are three logical operators as seen in the examples above: `and` , `or` , and `not`. They function as they do in English.

- For example, the first expression in the table above will be true if x == 17 OR it would also be true if X > 23.
- For the second expression, both simple expressions must be true: userName must have the value "David" AND userAge must be less than 21 for this to be true.
- In the third example, the ! operator turns a *true* into a *false*, and a *false* into a *true*.

Here are 3 truth tables to help explain how these logical operators work. The true and false in the tables below can be any boolean expression:

and	
Expression	**Value**
True and True	True
True and False	False
False and True	False
False and False	False

or	
Expression	**Value**
True or True	True
True or False	True
False or True	True
False or False	False

not	
Expression	**Value**
not True	False
not False	True

Given the following variables, here are some expressions and their values:

```
x = 3
y = 17.5
word = "tree"
```

Expression	Value
`x > y and word == "tree"`	False
`x != 17 or y > 18`	True
`"tree" > "task" and x == 3`	True
`not ( x < y)`	False
`x < y or y <10`	True

Here is the precedence for all operators, arithmetic, relational, and logical:

arithmetic operators
relational operators
logical operators
assignment operators

1.7 Common Errors Writing Compound Logical Expressions

A compound logical expression is one that makes use of logical operators to join 'sub conditions'. We use constructions like this all the time in our daily lives, and the logic works the same way. However, students new to programming sometimes make mistakes translating spoken conditions into code. Consider the following statement: "If the number is between 7 and 11, tell the user that they win!" How would you write the logical expression to meet these requirements? Here is a common error:

```
number >= 7 and <= 11
```

The correct form of the expression is

```
number >= 7 and number <= 11
```

Notice in the correct expression that each relational operator has its own set of values to compare, and the logical operator is used to join these sub-expressions.

Here's another one to try: "If either of the two occupants of the hotel room is at least 55 years old, use the 'senior citizen rate'." How would you write the test expression?

Here is appropriate code:

```
age1 >= 55 or age2 >= 55
```

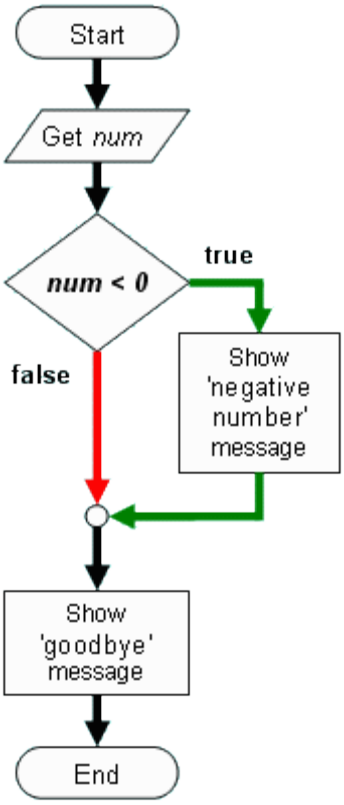
One more very common error in logical expressions is to mistakenly use the assignment operator (=) in place of the equality operator (==). This is not a syntax error, but will not produce the desired results, so check your code carefully!

2 The if statement

2.1 The Simple if Statement

Our goal is to create program code that makes **decisions**. Here is a very simple example:

```
num = float( input("Please enter a number: ") )
if num < 0 :
    print ( "The number is negative" )
print  ( "Goodbye!" )
```



The **if statement** makes a decision: if the value of the boolean expression num < 0 is **True**, the **conditional statement (or 'action')** is executed and the statement is displayed . If the value of the logical expression is **False**, it does NOT execute the conditional statement. To the right is the flowchart for this if statement. Notice how it will always follow one of two possible paths: it will either display the message (the green path), or it won't display the message (the red path). Also notice how both paths lead to the same exit point; any statements after this program code are not part of the if statement and will be executed.

Python uses indenting, just like with functions, to identify code that is part of the if statement (the conditional part) or if it is after the if statement. The last print statement will always execute. It is after the if statement; notice that it is not indented.

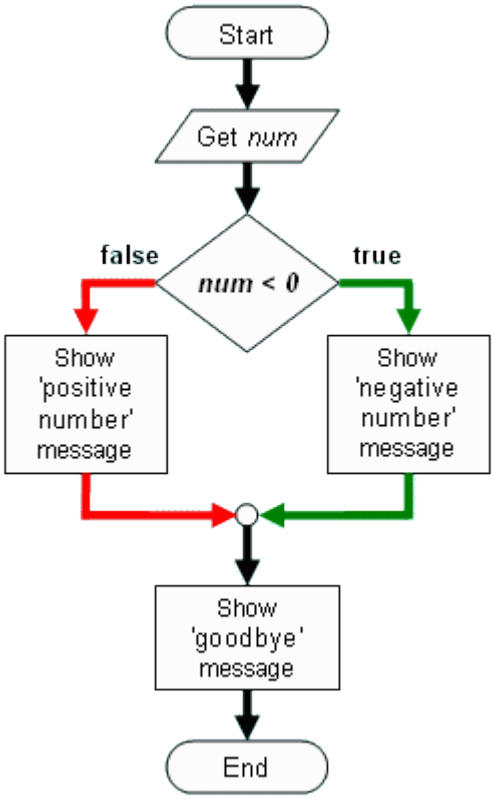
An if statement is an example of a control structure. It controls the path of execution. We'll see other control structures next week when we learn about loops.

2.2 The if...else Statement

A program can choose between two different paths in the program. Look at the flowchart and program code below. This introduces a new keyword, **else**. It is required in this example...

```
num = float( input( "Enter a number: "  ) )
if num < 0 :
    print ( "The number is negative." )
else:
    print( "The number is positive." )

print( "Good-bye!" )
```

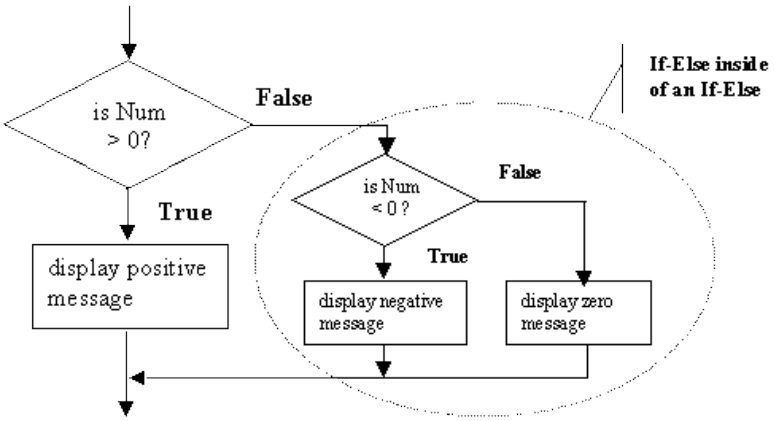


If the test expression is true, then the conditional block below the if will execute; the conditional block under the else will be skipped over. If the test expression is false, the first block is skipped over, and the conditional block under the else will execute. Regardless of which path is taken, whatever code follows this if...else statement will always execute.

2.3 Nested if Structures

You can add more paths to the decision structure by **nesting** one *if...else* structure inside of another. One path will be followed if the value entered is positive (> 0), one will be followed if the value is negative (< 0), and the third path will be followed if neither of the first two paths is followed (num equals 0). The code for this flowchart is shown on the left-hand side.

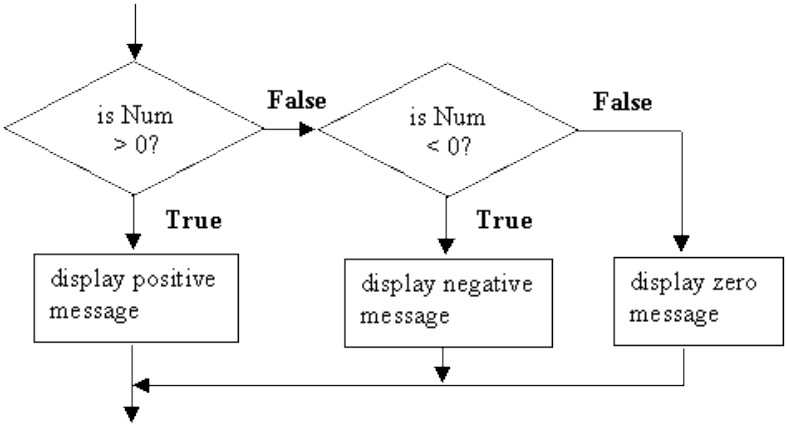
```
num = float( input( " Enter a number: "))
if num > 0 :
    print ( num, "is greater than zero" )
else:
    if num < 0 :
        print( num, " is less than zero" )
    else:
        print( num , " is equal to zero" )
```



Notice that the second *if-else* statement is contained completely within the second **block** of the first *if-else* statement. Indenting is critical to keep your code accurate.

The logic can be restructured slightly, as seen below, and the code can be shortened. Notice that the code is also indented differently, to make it look cleaner.

```
num = float( input ("Enter a number:"))
if num > 0:
    print ( num, " is greater than zero" )
elif num < 0:
    print ( num , " is less than zero")
else:
    print ( num, " is equal to zero" )
```



This concept can be extended indefinitely, using as many elif branches as necessary, so that your program can perform a long sequence of tests -- a multi-way selection -- if necessary.

Here is another example, written 3 different ways. The first attempt is the worst. The middle attempt corrects the problem, and the third attempt introduces an improvement over the middle one.

Poor example. Each individual if statement needs to be tested though only 1 will be true. It would be nice if the subsequent testing could be skipped once one of the test expressions is true.

```
score = float( input("Enter value") )

if score >= 0.0 and score < 0.25:
    print( "Bottom" )
if score >= 0.25 and score < 0.5):
    print ( "3rd quartile" )
if score >= 0.5 and score < 0.75):
    print ( "2nd quartile" )
if score >=0.75 and score <= 1.0:
    print ( "Top quartile" )
```

Here is the solution, using the **elif structure**. Once one of the test conditions is true, the rest of the test conditions are skipped.

```
score = input("Enter value")

if score >= 0.0 and score < 0.25:
    print ( "Bottom" )
elif score >= 0.25 and score < 0.5:
    print ( "3rd quartile" )
elif score >= 0.5 and score < 0.75:
    print ( "2nd quartile" )
else: # notice that no test is needed here
    print ( "Top quartile" )
```

Smarter solution using **what we know at each step**. Look at the test of **(score <0.5)** and notice we don't test the lower bound as we did in the previous example. Why not? Well, if the test above it (the **if** test) is false, then we KNOW that the value is 0.25 or better, so we only have to test < 0.5. This only works when we use elif or else.

```
score = input("Enter value")

if score >= 0.0 and score < 0.25:
    print ( "Bottom" )
elif score < 0.5:
    print ( "3rd quartile" )
elif score < 0.75 :
    print ( "2nd quartile" )
else :
    print ( "Top quartile" )
```

I find myself using nested if statements when I'm testing 2 different things, not testing multiple cases of the same expression. Here is an example:

```
age = float(  input( "Enter age" )  )
married = input( "Enter m for married or s for single: ")
married = married.upper( ) # this way we only have to test upper case strings

lowest = False # flag variable to signal if this user qualifies for the lowest
rate #calculating car insurance, based on age and marital status

# insurance is higher for younger, single people
if age < 25:
    if married == 'M':
        insurance = 300
    else :
        insurance = 350
else:  # this is for people 25 and older
    if married == 'M':
        insurance = 250
        lowest = True
    else:
        insurance = 275

if lowest :
    print ( "You qualified for our lowest rate!" )
print ( 'your rate is', insurance )
```

Notice the variables **lowest** above. It is an example of a boolean variable. A **boolean variable** contains one of 2 values: True or False. The variable **lowest** is also referred to as a **flag variable.** It is used to signal if something happened (such as the user input qualifying for the lowest rate category).

2.4 Testing Programs with Selection Structures

When testing a program that uses selection structures, test cases must be chosen to carefully check the program's operation down the different paths and right at the threshold of each of the conditions. The threshold value is the value where the condition changes. For example, in this insurance example, 25 is the threshold value. To fully test this sample code, you'd want to test 24 and 25 to make sure they fall into the right categories. It is very easy for 'off-by-one' errors to creep into conditions (for example, using '>' when '>=' is needed), and the only way to catch these logic errors is to test the code thoroughly.