CS303E: Elements of Computers and Programming

Conditionals and Boolean Logic

Mike Scott
Department of Computer Science
University of Texas at Austin

Adapted from
Professor Bill Young's Slides

Last updated: May 25, 2021
Booleans

So far we’ve been considering *straight line code*, meaning to do one statement after another.

a.k.a. *sequential flow of control*

But often in programming, you need to ask a question, and *do different things* based on the answer.

**Boolean** values are a useful way to refer to the answer to a yes/no question.

The Boolean **literal values** are the values: True, False.

A Boolean **expression** evaluates to a Boolean value.
Using Booleans

```python
>>> import math
>>> b = ( 30.0 < math.sqrt( 1024 ) )  
>>> print( b )
True
>>> x = 1  
>>> x < 0  
False
>>> x >= -2  
True
>>> b = ( x == 0 )  
>>> print ( b )
False
```

Booleans are implemented in the `bool` class.
Internally, Python uses 0 to represent False and 1 to represent True. You can convert from Boolean to int using the `int` function and from int to Boolean using the `bool` function.

```python
>>> b1 = (-3 < 3)
>>> print(b1)
True
>>> bool(1)
True
>>> bool(0)
False
>>> bool(4)
True
```
In a **Boolean context**—one that expects a Boolean value—False, 0, "" (the empty string), and `None` all is considered False and any other value is considered True.

```python
>>> bool("xyz")
True
>>> bool(0.0)
False
>>> bool("")
False
>>> if 4: print("xyz")  # boolean context
xyz
>>> if 4.2: print("xyz")
xyz
>>> if "ab": print("xyz")
xyz
```

This may be confusion but can be very useful in some programming situations.
Comparison Operators

The following comparison (or relational) operators are useful for comparing numeric values:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>x &lt; 0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>x &lt;= 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>x &gt; 0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>x &gt;= 0</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>x == 0</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>x != 0</td>
</tr>
</tbody>
</table>

Each of these returns a Boolean value, True or False.

```python
>>> x = 10
>>> (x == math.sqrt(100))
True
>>> (x = math.sqrt(100))
SyntaxError: invalid syntax
```

What happened on that last line?
Be very careful using “==” when comparing *floats*, because float arithmetic is approximate.

```python
>>> (1.1 * 3 == 3.3)
False
>>> 1.1 * 3
3.3000000000000003
```

The problem: converting decimal 1.1 to binary yields a *repeating* binary expansion: $1.000110011\ldots = 1.00011$. That means it *can’t be represented exactly* in a fixed size binary representation.

Thought for the day. Some rational numbers are repeating decimals in one base, but not in others. $1/3 = 0.33333\ldots_{10} = 0.1_3$
It’s often useful to be able to perform an action *only if* some conditions is true.

General form:

```python
if boolean-expression:
    statement(s)
```

Note the colon after the boolean-expression. **All of the statements controlled by the if must be indented the same amount.**

```python
if ( y != 0 ):
    z = ( x / y )
```
If Statement Example

In file if_example.py:

```python
def main():
    # A very uninteresting function to
    # illustrate if statements.
    x = int(input("Input an integer or 0 to do nothing: "))
    if (x != 0):
        print('The number you entered was', x, '. Thank you!')
```

Would "if x:" have worked instead of "if ( x != 0 ):"?

```bash
>>> runfile('C:/Users/scottm/PycharmProjects/As:
Input an integer or 0 to do nothing: >>? 10
The number you entered was 10 . Thank you!
```
A two-way **If-else** statement executes one of two actions, depending on the value of a Boolean expression.

**General form:**

```python
if boolean-expression:
    true-case-statement(s)
else:
    false-case-statement(s)
```

Note the colons after the boolean-expression and after the `else`. All of the statements in both *if* and else branches should be indented the same amount.
If-else Statement: Example

In file `compute_circle_area.py`:

```python
import math

def main():
    # Estimate area of circle based on radius from user
    radius = float(input("Enter the radius of a circle: "))
    if (radius >= 0):
        area = math.pi * radius ** 2
        print('A circle with a radius of', radius,
             'has an area of', area)
    else:
        print('Negative radius entered: ', radius)

main()
```

Enter the radius of a circle: 4.3
A circle with a radius of 4.3 has an area of 58.088048

Enter the radius of a circle: -3.75
Negative radius entered: -3.75
If you have multiple options, you can use if-elif-else statements.

General Form:

```python
if boolean-expression1:
    statement(s)
elif boolean-expression2:
    statement(s)
elif boolean-expression3:
    ...
else:  # optional
    statement(s)
```

You can have any number of `elif` branches with their conditions. The else branch is optional.
Sample Program: Calculate US Federal Income Tax

### Simplified US Federal Income Tax Table

<table>
<thead>
<tr>
<th>Single filers</th>
<th>Tax rate</th>
<th>Taxable income bracket</th>
<th>Tax owed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
<td>$0 to $9,875</td>
<td>10% of taxable income</td>
</tr>
<tr>
<td></td>
<td>12%</td>
<td>$9,876 to $40,125</td>
<td>$987.50 plus 12% of the amount over $9,875</td>
</tr>
<tr>
<td></td>
<td>22%</td>
<td>$40,126 to $85,525</td>
<td>$4,617.50 plus 22% of the amount over $40,125</td>
</tr>
<tr>
<td></td>
<td>24%</td>
<td>$85,526 to $163,300</td>
<td>$14,605.50 plus 24% of the amount over $85,525</td>
</tr>
<tr>
<td></td>
<td>32%</td>
<td>$163,301 to $207,350</td>
<td>$33,271.50 plus 32% of the amount over $163,300</td>
</tr>
</tbody>
</table>

Source: [https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets](https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets)
# Ask user for income and calculate US Federal income tax for 2021.
# Tax rates and income bracket data from
# https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets

def main():
    income = int(input('Enter 2021 income: '))
    print()
    if income <= 9_875:
        tax = income * 0.1
        bracket = "10%"
    elif income <= 40_125:
        tax = 987.5 + (income - 9_875) * 0.12
        bracket = "12%"
    elif income <= 85_525:
        tax = 4617.50 + (income - 40_125) * 0.22
        bracket = "22%"
    elif income <= 163_300:
        tax = 14605.50 + (income - 85_525) * 0.24
        bracket = "24%"
    else:
        tax = 33271.50 + (income - 163_300) * 0.32
        bracket = "32%"

    print('An income of', income, 'places you in the', bracket, 'income bracket.')
    print('The US Federal tax on an income of', income, 'is', tax)
Let’s take a break here and resume in the next video.
Python has **logical operators** (and, or, not) that can be used to make compound Boolean expressions.

- **not** : logical negation
- **and** : logical conjunction
- **or** : logical disjunction

Operators **and** and **or** are always evaluated using *short circuit evaluation*.

\[
( x \mod 100 == 0 ) \text{ and not } ( x \mod 400 == 0 )
\]
Truth Tables

**And:** \((A \text{ and } B)\) is True whenever both \(A\) is True and \(B\) is True.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(A \text{ and } B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

\[\text{Not: not } A\]

is True whenever \(A\) is False.

<table>
<thead>
<tr>
<th>(A)</th>
<th>(\text{not } A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

**Or:** \((A \text{ or } B)\) is True whenever either \(A\) is True or \(B\) is True.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(A \text{ or } B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Remember that “is True” really means “is not False, the empty string, 0, or None.”
Notice that \((A \text{ and } B)\) is False, if \(A\) is False; it doesn’t matter what \(B\) is. 
\textit{So there’s no need to evaluate }\(B\), \textit{if }\(A\) \textit{is False!}

Also, \((A \text{ or } B)\) is True, if \(A\) is True; it doesn’t matter what \(B\) is. 
\textit{So there’s no need to evaluate }\(B\), \textit{if }\(A\) \textit{is True!}

```python
>>> x = 13
>>> y = 0
>>> legal = (y == 0 or x / y > 0)
>>> print(legal)
True
```

Python doesn’t evaluate \(B\) if evaluating \(A\) is sufficient to determine the value of the expression. \textit{That’s important sometimes.} This is called \textit{short circuiting} the evaluation. Stopping early when answer it know.
In a Boolean context, Python doesn’t always return True or False, just something equivalent. What’s going on in the following?

```python
>>> "" and 14
'\n>>> bool("" and 14)
False
>>> 0 and "abc"
0
>>> bool(0 and "abc")
False
>>> not (0.0)
True
>>> not (1000)
False
>>> 14 and ""
'\n>>> 0 or "abc"
'abc'
>>> bool(0 or 'abc')
True
```

# equivalent to False
# coerced to False
# equivalent to False
# coerced to False
# same as not( False )
# same as not( True )
# equivalent to False
# same as False or True
# equivalent to True
# coerced to True
Here's a concise way to do a Leap Year computation:

```python
# Determine if year entered is a leap year or not.
def main():
    year = int(input('Enter a year: '))
    is_leap_year = (((year % 4 == 0)  
        and (not (year % 100 == 0) or (year % 400 == 0))))

    if is_leap_year:
        print(year, "is a leap year.")
    else:
        print(year, 'is not a leap year."

main()
```

Note the use of outer parenthesis on the assignment to is_leap_year to avoid the use of the continuation character, "\"."
Leap Years Revisited

> python LeapYear2.py
Enter a year: 2000
Year 2000 is a leap year.
> python LeapYear2.py
Enter a year: 1900
Year 1900 is not a leap year.
> python LeapYear2.py
Enter a year: 2004
Year 2004 is a leap year.
> python LeapYear2.py
Enter a year: 2005
Year 2005 is not a leap year.
A Python **conditional expression** returns one of two values based on a condition.

Consider the following code:

```python
# Set parity according to num
if (num % 2 == 0):
    parity = "even"
else:
    parity = "odd"
```

This sets variable `parity` to one of two values, “even” or “odd”.

An alternative is:

```python
parity = "even" if (num % 2 == 0) else "odd"
```
General form:

\[ \text{expr-1 if boolean-expr else expr-2} \]

It means to return \text{expr-1} if \text{boolean-expr} evaluates to True, and to return \text{expr-2} otherwise.

```python
# find maximum of x and y
max = x if (x >= y) else y
```
Use of conditional expressions can simplify your code.

In file test_sort.py:

```python
# Determine if 3 numbers are in sorted ascending order.
def main():
    x = float(input("Enter first number: "))
    y = float(input("Enter second number: "))
    z = float(input("Enter second number: "))
    print('Ascending' if (x <= y) and (y <= z)
           else 'Not Ascending')

main()
```

Enter first number: 12
Enter second number: 57
Enter second number: 109
Ascending

Enter first number: -26.6
Enter second number: 0.72
Enter second number: -12.75
Not Ascending
Arithmetic expressions in Python attempt to match widely used mathematical rules of precedence. Thus,

\[ 3 + 4 \times (5 + 2) \]

is interpreted as representing:

\[ (3 + (4 \times (5 + 2))) \]

That is, we perform the operation within parenthesis first, then the multiplication, and finally the addition.

To make this happen we precedence rules are enforced.
The following are the precedence rules for Python, with items higher in the chart having higher precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>Unary plus, minus, like - 3, +12</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>not</td>
<td>logical negation</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Multiplication, division, integer division, remainder</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary plus, minus</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>Comparison</td>
</tr>
<tr>
<td>==, !=</td>
<td>Equal, not equal</td>
</tr>
<tr>
<td>and</td>
<td>Conjunction</td>
</tr>
<tr>
<td>or</td>
<td>Disjunction</td>
</tr>
</tbody>
</table>
Precedence Examples

```python
>>> -3 * 4
-12
>>> - 3 + - 4
-7
>>> 3 + 2 ** 4
19
>>> 4 + 6 < 11 and 3 - 10 < 0
True
>>> 4 < 5 <= 17
# notice special syntax
True
>>> 4 + 5 < 2 + 7
False
>>> 4 + (5 < 2) + 7
# this surprised me!
11
```

Most of the time, the precedence follows what you would expect.
Operators on the same line have equal precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>Binary plus, minus</td>
</tr>
<tr>
<td>*, /, //, %</td>
<td>Multiplication, division, integer division, remainder</td>
</tr>
</tbody>
</table>

Evaluate them left to right.

All binary operators are *left associative*. Example: \( x + y - z + w \) means \( ((x + y) - z) + w \).

Note that assignment is *right associative*.

\[
x = y = z = 1 \quad \# \text{ assign } z \text{ first}
\]
Use parenthesis to override precedence or to make the evaluation clearer.

```python
>>> 10 - 8 + 5
7
>>> (10 - 8) + 5
7
>>> 10 - (8 + 5)
-3
>>> 5 - 3 * 4 / 2
-1.0
>>> 5 - ((3 * 4) / 2)
-1.0
```

# an expression
# what precedence will do
# override precedence
# not particularly clear
# much better

Always try to make your code as easy to read as possible!