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 31, 2023

Booleans

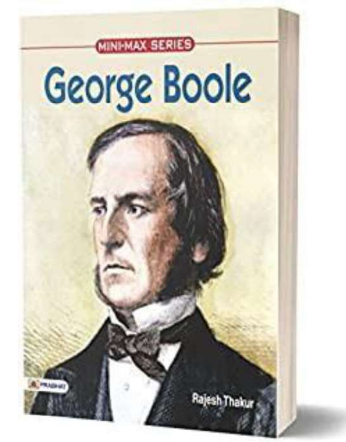
So far we've been considering *straight line code*, meaning executing one statement after another.

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

```
>>> import math
>>> b = ( 30.0 < math.sqrt( 1024 ))
>>> print( b)
True
>>> x = 1            # statement
>>> x < 0            # boolean expression
False
>>> x >= -2          # boolean expression
True
>>> b = ( x == 0 )   # statement containing
                     # boolean expression
>>> print (b)
False
```

Booleans are implemented in the `bool` class.

Booleans

Internally, Python uses 0 to represent False and anything not 0 to represent True. You can convert from Boolean to int using the `int` function and from `int` to `Boolean` using the `bool` function.

```
>>> b1 = (-3 < 3)
>>> print(b1)
True
>>> bool(1)
True
>>> bool(0)
False
>>> bool(4)
True
>>> 
```

Boolean Context

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.

```
>>> 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:

Operator	Meaning	Example
<	Less than	x < 0
<=	Less than or equal	x <= 0
>	Greater than	x > 0
>=	Greater than or equal	x >= 0
==	Equal to	x == 0
!=	Not equal to	x != 0

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

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

What happened on that last line?

Caution

Be very careful using "==" when comparing *floats*, because float arithmetic is approximate.

```
>>> (1.1 * 3 == 3.3)
False                        # What happened?
>>> 1.1 * 3
3.3000000000000003
```

The problem: converting decimal 1.1 to binary yields a *repeating* binary expansion: 1.000110011 . . . = 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. $1/3 = 0.33333\ldots_{10} = 0.1_3$

One Way If Statements

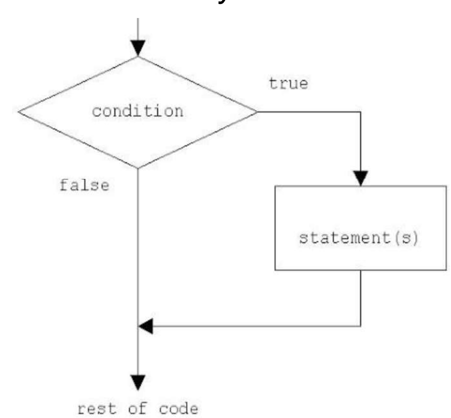
It's often useful to be able to perform an action *only if* some conditions is true.

General form:

```
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.




```
if y != 0:
    z = ( x / y )
```

If Statement Example

In file if_example.py:

```
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):"?

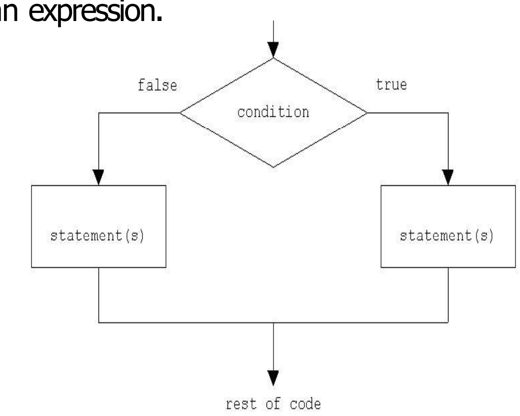
```
>>> runfile('C:/Users/scottm/PycharmProjects/As
Input an integer or 0 to do nothing: >? 10
The number you entered was 10 . Thank you!
>>> runfile('C:/Users/scottm/PycharmProjec
Input an integer or 0 to do nothing: >? 0
```


Two-way If-else Statements

A two-way **If-else** statement executes one of two actions, depending on the value of a Boolean expression.

General form:

```
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:

```
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()
```

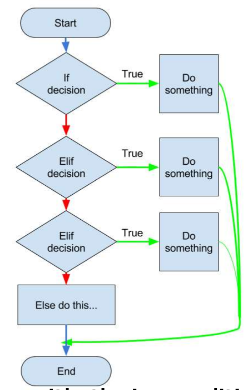
```
Enter the radius of a circle: 4.3
A circle with a radius of 4.3 has an area of 58.088048

Enter the radius of a circle: -3.75
Negative radius entered: -3.75
```


Multiway if-elif-else Statements

If you have multiple options, you can use if-elif-else statements.

General Form:

```
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

Source: https://www.nerdwallet.com/article/taxes/federal-income-tax-brackets

Single filers

Tax rate	Taxable income bracket	Tax owed
10%	$0 to $9,875	10% of taxable income
12%	$9,876 to $40,125	$987.50 plus 12% of the amount over $9,875
22%	$40,126 to $85,525	$4,617.50 plus 22% of the amount over $40,125
24%	$85,526 to $163,300	$14,605.50 plus 24% of the amount over $85,525
32%	$163,301 to $207,350	$33,271.50 plus 32% of the amount over $163,300

income_tax.py

```python
# 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 = 4_617.50 + (income - 40_125) * 0.22
        bracket = "22%"
    elif income <= 163_300:
        tax = 14_605.50 + (income - 85_525) * 0.24
        bracket = "24%"
    else:
        tax = 33_271.50 + (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)
```


Break

Maybe take a break?

Logical Operators

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 % 100 == 0 ) and not ( x % 400 == 0 )
```

Truth Tables

And: (A and B) is True whenever both A is True and B is True.

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

Or: (A or B) is True whenever either A is True or B is True.

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

Not: not A is True whenever A is False.

A	not A
False	True
True	False

Remember that "is True" really means "is not False, the empty string, 0, or None."

Short Circuit Evaluation

Notice that (A and B) is False, if A is False; it doesn't matter what B is. *So there's no need to evaluate B, if A is False!*

Also, (A or B) is True, if A is True; it doesn't matter what B is. *So there's no need to evaluate B, if A is True!*

```
>>> 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. *That's important sometimes.*
This is called *short circuiting* the evaluation.
Stopping early when answer it know.

Boolean Operators

In a Boolean context, Python doesn't always return True or False, just something equivalent. What's going on in the following?

```
>>> "" and 14
''                      # equivalent to False
>>> bool("" and 14)
False                   # coerced to False
>>> 0 and "abc"
0                       # equivalent to False
>>> bool(0 and "abc")
False                   # coerced to False
>>> not(0.0)            # same as not( False )
True
>>> not(1000)           # same as not( True )
False
>>> 14 and ""
''                      # equivalent to False
>>> 0 or "abc"          # same as False or True
'abc'                   # equivalent to True
>>> bool(0 or 'abc')    # coerced to True
True
```


Leap Years

Here's a concise way to do a Leap Year computation:

```
# 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.
```


Conditional Expressions

A Python **conditional expression** returns one of two values based on a condition.

Consider the following code:

```
# 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:

```
parity = "even" if ( num % 2 == 0 ) else "odd"
```


Conditional Expression

General form:

```
expr-1 if boolean-expr else expr-2
```

It means to return `expr-1` if `boolean-expr` evaluates to True, and to return `expr-2` otherwise.

```
# find maximum of x and y
max = x if (x >= y) else y
```


Conditional Expression

Use of conditional expressions can simplify your code.

In file test_sort.py:

```
# 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
```

Operator Precedence

Arithmetic expressions in Python attempt to match widely used mathematical rules of precedence. Thus,

```
3 + 4 * (5 + 2)
```

is interpreted as representing:

```
(3 + ( 4 * ( 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.

Precedence

The following are the precedence rules for Python, with items higher in the chart having higher precedence.

Operator	Meaning
+, -	Unary plus, minus, like - 3, +12
**	Exponentiation
not	logical negation
*, /, //, %	Multiplication, division, integer division, modulus
+, -	Binary plus, minus
<, <=, >, >=	Comparison
==, !=	Equal, not equal
and	Conjunction
or	Disjunction

Precedence Examples

```
>>> -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.

Precedence

Operators on the same line have equal precedence.

Operator	Meaning
+, -	Binary plus, minus
*, /, //, %	Multiplication, division, integer division, remainder

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     # assign z first
```

Use Parentheses to Override Precedence

Use parenthesis to override precedence or to make the evaluation clearer.

```
>>> 10 - 8 + 5              # an expression
7                           # what precedence will do
>>> (10 - 8) + 5
7                           # override precedence
>>> 10 - (8 + 5)
-3                          # not particularly clear
>>> 5 - 3 * 4 / 2
-1.0                        # better
>>> 5 - ((3 * 4) / 2)
-1.0
```

Work to make your code easy to read!