CS303E: Elements of Computers and Programming

Selections

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: August 27, 2024 at 14:25

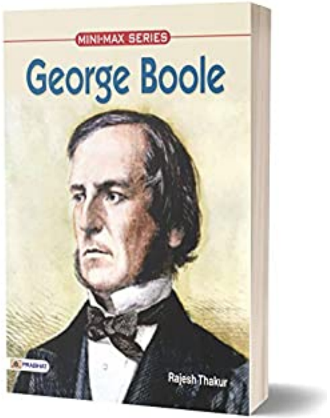
Booleans

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

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

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

The Python Boolean **constants** 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 1 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
>>> int( b1 )
1
>>> bool( 1 )
True
>>> bool( 0 )
False
>>> bool( 4 )          # what happened here?
True
```

Boolean Context

In a **Boolean context**—one that expects a Boolean value—False, 0, `""` (the empty string), and `None` all stand for False and *any other value* stands for True.

```
>>> bool("xyz")
True
>>> bool(0.0)
False
>>> bool("")
False
>>> if 4: print("xyz")      # 4 == True, in this context
xyz
>>> if "ab": print("xyz")   # "ab" == True
xyz
>>> if "": print("xyz")     # "" == False
>>>
```

This is very useful in many programming situations.

Comparison Operators

The following comparison 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.

```
>>> import math
>>> x = 10
>>> ( x == math.sqrt( 100 ))
True
```

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
```

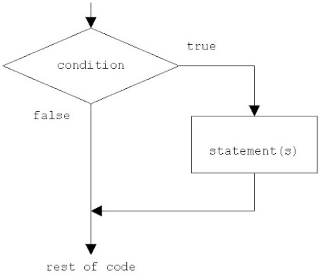
The problem: converting decimal 1.1 to binary yields a *repeating* binary expansion: $1.000110011\ldots = 1.0\overline{0011}$. That means *it can't be represented exactly* in a fixed size binary representation.

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

General form:

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

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




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

If Statement Example

In file `IfExample.py`:

```
def main():
    """ A pretty uninteresting function to illustrate
    if statements. """
    x = int( input("Input an integer, or 0 to stop: "))
    if ( x != 0 ):
        print( "The number you entered was", \
               x, ". Thank you!")

main()
```

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

```
> python IfExample.py
Input an integer, or 0 to stop: 3
The number you entered was 3 . Thank you!
> python IfExample.py
Input an integer, or 0 to stop: 0
>
```

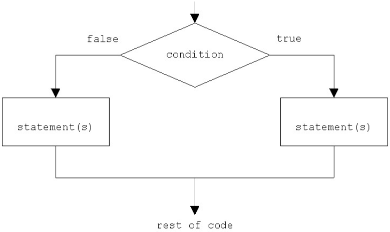
How could you get rid of the space before the period?

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 `ComputeCircleArea.py`:

```
import math

def main():
    """ Compute the area of a circle, given radius. """
    radius = float( input("Input radius: ") )
    if ( radius >= 0 ):
        area =  math.pi * radius ** 2
        print( "A circle with radius", radius, \
               "has area", format(area, "<5.2f") )
    else:
        print( "Negative radius entered.")

main()
```

```
> python ComputeCircleArea.py
Input radius: 4.3
A circle with radius 4.3 has area 58.09
> python ComputeCircleArea.py
Input radius: -3.4
Negative radius entered.
```

Let's take a break here and resume in the next video.

Nested If Statements: Leap Year Example

The statements under an `if` can themselves be `if` statements.

For example: Suppose you want to determine whether a particular year is a leap year. The algorithm is as follows:

1. If `year` is a multiple of 4, then it's a leap year;
2. unless it's a multiple of 100, and then it's not;
3. unless it's also a multiple of 400, and then it is.

Nested If Statements: Is Leap Year?

In file `LeapYear.py`:

```
def main():
    """ Is entered year a leap year? """
    year = int( input("Enter a year: ") )
    if ( year % 4 == 0 ):
        # Year is a multiple of 4
        if ( year % 100 == 0 ):
            # Year is a multiple of 4 and of 100
            if ( year % 400 == 0 ):
                IsLeapYear = True       # What do you know here?
            else:
                IsLeapYear = False      # What do you know here?
        else:
            IsLeapYear = True
    else:
        IsLeapYear = False              # What do you know here?
    if IsLeapYear:
        print( "Year", year, "is a leap year." )
    else:
        print( "Year", year, "is not a leap year.")

main()
```

Leap Year

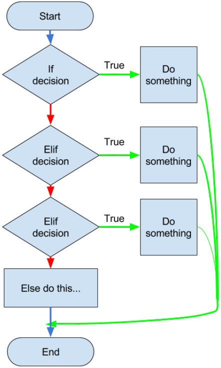
```
> python LeapYear.py
Enter a year: 2000
Year 2000 is a leap year.
> python LeapYear.py
Enter a year: 1900
Year 1900 is not a leap year.
> python LeapYear.py
Enter a year: 2004
Year 2004 is a leap year.
> python LeapYear.py
Enter a year: 2005
Year 2005 is not a leap year.
```

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.

If-elif-else Example

In file LeapYear3.py:

```
def main():
    # Is this a leap year
    year = int( input("Enter a year: ") )
    if ( year % 400 == 0 ):
        IsLeapYear = True
    elif ( year % 100 == 0 ):   # what's true here?
        IsLeapYear = False
    elif ( year % 4 == 0 ):     # what's true here?
        IsLeapYear = True
    else:                       # what's true here?
        IsLeapYear = False
    # Print result.
    if IsLeapYear:
        print( "Year", year, "is a leap year." )
    else:
        print( "Year", year, "is not a leap year.")

main()
```

We could always replace `elif` with nested if-else statements, but this is much more readable. *Be careful with your indentation!*

If-elif-else Example

```
> python LeapYear3.py
Enter a year: 2000
Year 2000 is a leap year.
> python LeapYear3.py
Enter a year: 2004
Year 2004 is a leap year.
> python LeapYear3.py
Enter a year: 1900
Year 1900 is not a leap year.
> python LeapYear3.py
Enter a year: 2005
Year 2005 is not a leap year.
```

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.*

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')        # equivalent to False or True
True
```

Leap Years Revisited

Here's an easier way to do our Leap Year computation:

In file LeapYear2.py:

```
def main():
    """ Input a year and test whether it's a leap year. """
    year = int( input("Enter a year: ") )

    # What's the logic of this assignment?
    IsLeapYear = ( year % 4 == 0 ) and \
        ( not ( year % 100 == 0 ) or ( year % 400 == 0 ) )

    # Print the answer
    if IsLeapYear:
        print( "Year", year, "is a leap year." )
    else:
        print( "Year", year, "is not a leap year.")

main()
```

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.
```

Let's take a break here and resume in the next video.

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:

```
expr1 if boolean-expr else expr2
```

It means to return expr1 if boolean-expr evaluates to True, and to return expr2 otherwise.

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

Why would it be a bad idea to use the variable name max here?

Conditional Expression

Use of conditional expressions can simplify your code.

```
def main():
    """ See if three numbers are input in ascending
        order. """
    xs, ys, zs = input ("Enter three numbers: ").split(",")
    x, y, z = float(xs), float(ys), float(zs)
    print( "Ascending" if ( x <= y and y <= z ) \
             else "Not ascending" )

main()
```

Note: `split()` is not introduced until slideset 8. Without it, you'd have to have three separate `input` statements.

```
> python TestSorted.py
Enter three numbers: 3, 5, 9
Ascending

> python TestSorted.py
Enter three numbers: 9, 3, 5
Not ascending
```

Operator Precedence

Arithmetic expressions in Python attempt to match standard syntax. Thus,

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

is interpreted as representing:

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

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

To make this happen we need *precedence rules*.

Precedence

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

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

Unary plus/minus means a sign, e.g. −3, +4.

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*. Why would it have to be?

```
x = y = z = 1    # assign z first
```

Use Parentheses to Override Precedence

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

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

Remember from the *Zen of Python*: Readability counts!

Next stop: Loops.