Booleans

So far we've 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 Boolean **constants** are the values: True, False. A Boolean **expression** evaluates to a Boolean value.

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
>>> 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.

```python
>>> 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.

```python
>>> bool("xyz")
True
>>> bool(0.0)
False
>>> bool("")
False
```  

This is very useful in some programming situations.

**Caution**

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

```python
>>> (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.

Comparison Operators

The following comparison operators are useful for comparing numeric values:

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

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

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

One Way If Statements

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)
```

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

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

In file IfExample.py:

```python
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):"?

```bash
> 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
>
```

Break

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

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:

```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 ComputeCircleArea.py:

```python
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()
```

```bash
> 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.
```
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:

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

In file LeapYear.py:

```python
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's true here?
            else:
                IsLeapYear = False  # What's true here?
            else:
                IsLeapYear = True
        else:
            IsLeapYear = False  # What's true here?
        else:
            IsLeapYear = True
    else:
        IsLeapYear = False
    if IsLeapYear:
        print("Year ", year, "is a leap year.")
    else:
        print("Year ", year, "is not a leap year.")
```

 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:

```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.
If-elif-else Example

In file LeapYear3.py:

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

Notice that we could always replace elif with nested if-else statements. But this is much more readable. *Be careful with your indentation!*

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 )

> 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.

Truth Tables

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A 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>

**Not**: \(\neg A\) is True whenever \(A\) is False.

<table>
<thead>
<tr>
<th>A</th>
<th>\neg A</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A 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.”
Short Circuit Evaluation

Notice that \((A \text{ and } B)\) 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, \((A \text{ or } B)\) 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!

```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. 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?

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

Leap Years Revisited

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

In file LeapYear2.py:

```python
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()
```

> 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.

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"
```

Use of conditional expressions can simplify your code.

In file `TestSorted.py`:

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

```
> python TestSorted.py
Enter three numbers: 3, 5, 9
Ascending
> python TestSorted.py
Enter three numbers: 9, 3, 5
Not ascending
```
Arithmetic expressions in Python attempt to match standard syntax. Thus,

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

is interpreted as representing:

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

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 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**

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</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>

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

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 # assign z first
```
Use parentheses to override precedence or to make the evaluation clearer.

```python
>>> 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
```

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