Finishing up the primality testing program
Python associates boolean values to everything

- Every object (e.g., “6”, 9.98, “”) has an associated boolean value.

- Use the `bool` function to find out the boolean value of an object.

- **Examples:** Try evaluating
  
  ```python
  bool(“a”)     bool(0)      x = 6
  bool(“”)     bool(1)      bool(x)
  ```
What is True? And what is False?

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>The constant True</td>
<td>False</td>
</tr>
<tr>
<td>1, numbers other than 0</td>
<td>0</td>
</tr>
<tr>
<td>Non-empty strings</td>
<td>Empty strings</td>
</tr>
</tbody>
</table>

Later when we study *Lists, Dictionaries*, etc., we will see that empty instances of these types of objects are also considered False.
A new version of the `intToBinary` program

```
while n:
    suffix = str(n%2) + suffix
    n = n/2
```

The boolean expression after the `while` can just be `n` instead of `n > 0`. 
and and or are “short-circuit” operators

- **A and B:**
  - A is evaluated first.
  - If A is **False** then the expression evaluates to **False**, *without B being evaluated*.
  - If A is **True** then B is evaluated and the expression evaluates to the value of B.
Try evaluating these example expressions

- \( \frac{100}{0} \)
- False and \( \frac{100}{0} \)
- \( \frac{100}{0} \) and False
- True and \( \frac{100}{0} \)
- \( \frac{100}{0} \) and True
and and or are “short-circuit” operators

- **A or B:**
  - A is evaluated first.
  - If A is True then the expression evaluates to True, *without B being evaluated*.
  - If A is False then B is evaluated and the expression evaluates to the value of B.
Final remarks on primality testing

- In the *worst case*, the while-loop in the programs makes $\sqrt{n}$ iterations.

- For an input with, say 100 digits, what might the running time be?

- $n = 10^{100}$. Therefore $\sqrt{n} = 10^{50}$. Even if each iteration of the while-loop took a nanosecond ($10^{-9}$ seconds), the program would take $3.17 \times 10^{33}$ years!
So how are numbers with 300 digits tested?

- Based on facts in *number theory* (an area of mathematics), several fast primality-testing algorithms have been developed.

- **Examples:**
  
  *Miller-Rabin test:*
  
  This is a *randomized* algorithm – a step in the algorithm performed by rolling dice.
  
  The algorithm is not always correct! A composite number may be classified as a prime, with small and tune-able error probability.
More in-depth discussion

- Data types
- Variables
- Key words
- Built-in functions
- Modules
- Control flow statements
Data types

- We have seen four data types thus far:
  - int: -90, 8987
  - float: 9.98, -3.54
  - str: “hello”, “a”
  - bool: True, False
Numeric data types

- Python supports four numeric data types:
  - *plain integers*,
  - *long integers*,
  - *floating point numbers*, and
  - *complex numbers*.

- Plain integers, i.e., objects of type `int`, are those that fit in 32 bits.
A *bit* (short for binary digit) is the smallest unit in a computer.

A *byte* is 8 bits; a *word* is 2 bytes (16 bits).

Any integer that can be represented in binary using 4 bytes (or 2 words or 32 bits) is an *int* type object in Python.

The largest *int* object is

\[ 2^{31} - 1 = 2147483647 \]

And the smallest is -2147483648
Playing with these notions

- Try
  ```python
  import sys
  sys.maxint
  ```

- Also try this
  ```python
  n = -37
  bin(n)
  n.bit_length()
  ```

- Try this also
  ```python
  type(sys.maxint+1)
  ```
Integers of type `long` can be arbitrarily large (or small). In other words, the type `long` provides infinite precision.

A long constant can be explicitly specified by appending an `L` at the end of the integer. Try

```java
x = 875L
```

`type(x)`

Operations can be performed on a mix of `long` and `int` objects; the type of the answer will be the larger type, i.e., `long`. 

```java
x = 875;
```

`type(x)`
The float type

- Numbers with decimal points are easily represented in binary:
  - 0.56 (in decimal) = 5/10 + 6/100
  - 0.1011 (in binary) = \( \frac{1}{2} + 0/4 + 1/8 + 1/16 \)

- However, not all real numbers (even rational numbers) can be represented *exactly* by finite sums of these fractions.

- So always be wary of (small) errors in dealing with floating point numbers. Try 0.1 + 0.2.