One More Version of the Primality Testing Program

FEB 6TH, 2013

Is using break bad programming?

- Some programming “purists” think that the use of the break statement is bad programming practice.
- Comment from on online discussion on programming:

> *Generally, breaking out of loops is considered bad form because it tends to obfuscate your code. It's harder to follow the "flow" of a program with continue/break thrown in everywhere. It's especially worse if you use it in nested loops, etc.*

- I don’t think using the break statement is bad programming practice, but yes it needs to be used with caution.

An alternative to using break

- We want to stay in the loop while

```
        n <= factorUpperBound
```

(there are more factors to consider)

```
                and
```

```
        isPrime == True
```

(we have not yet found a factor)

- We can express this using the *boolean operator* `and` in Python.

Primality testing: Version 4

```
# Programmer: Sriram Pemmaraju
# Date: Jan 30th, 2012
# This program reads a positive integer, greater than 1 and
# determines whether this integer is a prime or not.
# Version 3

import math

n = int(raw_input("Please type a positive integer, greater than 1: "))

factor = 2 # initial value of possible factor
isPrime = True # variable to remember if n is a prime or not
factorUpperBound = math.sqrt(n) # the largest possible factor we need to test is sqrt(n)

# loop to generate and test all possible factors
while (factor <= factorUpperBound) and (isPrime):
    # test if n is evenly divisible by factor
    if (n % factor == 0):
        isPrime = False

    factor = factor + 1

# Output
if isPrime:
    print n, " is a prime."
else:
    print n, " is a composite."
```

Python boolean operators

- and, or, and not are the three Python boolean operators.

- A and B is true only when both A *and* B are true.

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

Examples: play with these

- (x <= 10) and (x > 4)
- (x < 4) and (x > 10)
- (x < 10) and True
- (x >= 0) and False

The or operator

- A or B is True when A is True or B is True or both.
- In other words, A or B is False only when both A and B are False.

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

Examples: play with these

- (x <= 10) or (x > 4)
- (x < 4) or (x > 10)
- (x < 10) or True
- (x >= 0) or False

The not operator

- This is a *unary* operator, i.e., it operates on only one operand.

A	not A
True	False
False	True

- **Examples:**
  - not (x < 10)
  - not (x == 10)
  - not (x>=-10)

How fast is our algorithm?

- In the *worst case*, the while-loop in the programs makes √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 3.17×10^{33} years!

Timing Python programs

- The `time` module contains functions that allow us to determine (within the program), how much time different blocks of code take.

- There are many functions defined in this module. The one we will use most often is called `time` and is called with *no arguments*.

- So once the `time` module has been imported, a call to this function will look like

```
time.time()
```

- It returns the number of seconds (as f loating point number) elapsed since 12 am (midnight), Jan 1st, 1970.

Timing Python programs

```
import time
…
start = time.time()
…
#code you want timed
…
end = time.time()
elapsedTime = end - start
```

This is typically how you would time a piece of Python code.

Example

```
import time
n = 10000000
originalN = n

start = time.time()
while n > 0:
    n = n - 1

end = time.time()
print "It takes", end-start, " seconds for", originalN, "iterations of the while loop."
```

Output:

```
It takes 1.54960203171  seconds for 10000000 iterations of the while loop.
```

Timed version of Primality Testing

- Take a look at the posted program called primalityTestingTimed.py

- Here is the output of this program on a 10-digit prime.

```
Please type a positive integer, greater than 1: 5915587277
5915587277  is a prime.
The while-loop took  0.0328981876373 seconds.
```

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 a prime, with small and tune-able error probability.