A Few versions of the Primality Testing Program

FEB 4TH, 2013

Primality Testing: Version 0

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

factor = 2
isPrime = True

while factor < n:
    if (n % factor == 0):
        isPrime = False

    factor = factor + 1

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

An Easy Improvement

- As soon as we discover that a "candidate factor" is actually a factor of *n*, we know that *n* is a composite.

- We can therefore exit the loop at this point and not consider any more "candidate factors."

- The `break` statement provides a convenient way to exit a `while`-loop even before the boolean expression in the `while`-statement is falsified.

Discussing the code: The break statement

- The break statement forces the program to exit out of the smallest enclosing while-loop (or for-loop).

- **Example:**

```
n = 10
while n < 20:
    if n % 7 == 0:
        break
    n =n + 1
print n
```

Primality Testing: Version 1

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

factor = 2
isPrime = True

while factor < n:
    if (n % factor == 0):
        isPrime = False
        break
    factor = factor + 1

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

Understanding the Improvement

- If the input is a composite, then the `break` statement provides some savings in running time because the program does not have to run through all candidate factors 1 through N-1.

  - **Example:** 987654321 is a composite and
    987654321 = 3 x 329218107.
    So the break statement causes the loop to iterate twice.
    Without the `break` the loop would iterate about a billion times.

- For prime number inputs, there is no speed-up.

Another Improvement

- A number n does not have any factors larger than n/2, except itself. So we could stop generating candidate factors at n/2.

- But, wait we can do much better!

  We know **√n x √n = n**. Hence, if n has a factor larger than √n, then it has a factor smaller than √n also.

- This means that only factors 2, 3,..., floor(√n) need to be considered.

Example

- Say n = 123. Now $\sqrt{123}$ = 11.090536506409418.
- So if 123 has a factor greater than 11.09, then it has factor less than 11.09.
- This means in looking at "candidate" factors, we only need to look at numbers 2, 3, …, 11.

Primality Testing: Version 2

```
import math

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

factor = 2
isPrime = True
factorUpperBound = math.sqrt(n)

while factor <= factorUpperBound:
    if (n % factor == 0):
        isPrime = False
        break

    factor = factor + 1

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

Modules in Python

- A *module* in Python is a file that defines a collection of related functions.

- All the functions in a module can be used after the module has been *imported*, using the `import` statement (usually at the beginning of the program).

- A function `f` in a module `m` is called as

  `m.f`(arguments).

  For example, the `sqrt` function in the `math` module is called as `math.sqrt(n)`.

The math module

- Contains many functions:
  - Power and logarithmic functions
  - Trignometric functions
  - Hyperbolic functions
  - Mathematical constants

- Examples:
  - `math.log10(x)`: returns the logarithm to the base 10 of x.
  - `math.pow(x, y)`: returns x raised to the power of y.

Example Problem

Write a program that reads a positive integer and outputs the number of digits in the integer.

- Version with **while**-loops

```
n = int(raw_input("Enter a positive integer: "))

counter = 0
while n > 0:
    counter = counter + 1
    n = n / 10

print counter
```

Version with math functions

```
import math

n = int(raw_input("Enter a positive integer: "))
print int(math.log10(n)+1)
```

Questions

- How do we know what modules Python supports?
- How do we know what functions Python's `math` modules supports?

Answers:

- For all matters related to Python visit

  `http://docs.python.org/2/`

  This is the authoratative source on Python. I visit this website

  all the time when I program in Python.
- python.org contains a Python tutorial that is a great reference.
- Section 9.2 is on the math module and contains a list of math functions available in the module.
- There is a *module index* that lists all modules that Python 2.7.3 comes with.
- This is a good time for you to look over parts of the Python tutorial (e.g., 3.1.1 Numbers, 3.1.2 Strings, 3.2 First Steps Towards Programming, 4.1 If statements).

Back to primality testing

How much improvement do we get from considering “candidate factors” only up till square root of *n*?

- To answer these types of questions, a visit to “The Prime Pages” at http://primes.utm.edu/ is a good idea.
- Here you will see lots of lists of primes, including a list of the first 50 million primes.
- 982,451,653 is the 50 million-th prime; square root of this is roughly 31,344.
- So the difference is about 1 billion iterations versus about 31 thousand iterations!
- We will return to this issue of how much speed-up we get when we learn to *time* our programs in the next lecture.

Primality Testing: Version 3

```
# 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):
    # test if n is evenly divisible by factor
    if (n % factor == 0):
        isPrime = False
                    break

    factor = factor + 1

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

Discussing the code: Comments in Python

- The program contains "comments," i.e., text that is ignored by Python but serves to help the reader understand the code.

- Writing code first and then adding comments is backwards. We will never do this again. Now that we have talked about comments, we will always write comments and code together.

- Comments are preceded by the "#" symbol.

- Documenting code using comments is a critical part of programming.

- Comments are typically provided:
  - at the beginning of the program,
  - at the start of a block of code that performs a particular task, e.g., the while-loop that generates and tests factors,
  - to document the purpose of variables, etc.

- Later we will discuss a different mechanism for commenting a Python program called *documentation strings*.

Discussing the code: Basic guidelines for commenting

- Comments that contradict the code are worse than no comments at all!
- Comments that state the obvious (e.g., # This is a while-loop) make for unnecessary clutter are also worse than no comments at all.
- For now the comments you write should (i) help the reader understand your algorithm and (ii) help the reader understand tricky snippets of code.
- Comments can also be used to turn off lines of code that were inserted for the purposes of debugging.
- Your intended audience for documentation: your classmates, your graders, yourself a few weeks into the future.