Algorithms - a simple introduction in Python: Part Eight

317 is a prime, not because we think so, or because our minds are shaped in one way rather than another, but because it is, because mathematical reality is built that way.

G. H. Hardy *A Mathematician's Apology*

The Primes

One very well-known method for finding prime numbers is the Sieve of Eratosthenes.

```
First, write out all the integers. (Well, perhaps just up to a certain limit!)
You can cross off 1, because it is not prime.
Then repeat the following steps:
      Find the next highest number (n) that hasn't been crossed off.
      Cross off all of its multiples.
      If you don't find any multiples of n, you are finished.
```

If you wanted to write a Python program that did exactly that, it might look something like this:

```
# sieve.py The Sieve of Eratosthenes

def main():
    print("*** Sieve of Eratosthenes ***")
    nums = [True] * (int(input("Numbers up to: ")) + 1)
    print("\nPrimes:")
    display(sieve(nums))

def sieve(my_list):
    """Takes the list and sieves the indexes.
    Composite index values marked "False"."""
    limit = int(len(my_list) ** 0.5) + 1
    for i in range(2, limit):
        if my_list[i]:
            for j in range(i*i, len(my_list), i):
                my_list[j] = False
    return my_list

def display(some_list):
    """Takes the list and prints the indices of elements marked True.
    Prompts for user to press enter at intervals (for readability)."""
    for (index, is_prime) in enumerate(some_list[2:], 2):
        if is_prime:
            print(index, end = " ")
        if index % 100 == 0:
            print()
        if index % 1000 == 0:
            input("Press enter.")

if __name__ == "__main__":
    main()
```

This is okay, but what if you want to check if a particular number is prime, or find some specific prime without generating a huge list like this?

The Miller-Rabin Algorithm

The program that follows uses the Miller-Rabin algorithm. This is based on a version of "Fermat's Little Theorem".

```
# miller.py The Miller-Rabin Primality test
# Some Carmichael numbers: 561, 1105, 1729, 2465, 2821, 6601.
# Some Mersenne Primes:(2^13)-1, (2^17)-1, (2^19)-1,(2^31)-1,(2^61)-1,(2^89)-1.

import random

TESTS = 10 # default number of times to run the test.

def main():
    print("*** Primes ***")
    print("Enter a number to test for primality. ")
    print("For a list of primes, enter a start point and how many primes you want,
separated by a comma.")
    print("To test a Mersenne Number, type an 'm' followed by a comma and the
exponent.")
    print("Any other input will cause the program to quit.")

    while 1:
        my_string = input("? ")
        try:
            print(is_prime(int(my_string)))
        except ValueError:
            try:
                p, t = my_string.split(',')
                try:
                    p = int(p)
                    t = int(t)
                    if even(p):
                        p += 1
                    print(next_p(p, t, []))
                except ValueError:
                    try:
                        if p == 'm':
                            p = 2 ** int(t) - 1
                        print("Mersenne Number: ", p)
                        print(is_prime(p))
                    except ValueError:
                        break
            except:
                break

def square(x):
    """Squares x."""
    return x*x

def is_prime(x, t = TESTS):
    """Takes a number and the number of times to tun the test."""
    if x < 2:
        return False
    elif x < 4:
        return True
    elif even(x):
        return False
    else:
        return do_tests(x, t)

def even(n):
    """Returns True for even numbers."""
    if n % 2 == 0:
        return True
    else:
        return False

def miller(n):
    """Picks a random number and returns True if it passes the test."""
    return 1 == chk_expmod(random.randrange(1, n - 1), n - 1, n)
```

```
def do_tests(n, t):
    """Returns True only if the number passes t tests."""
    if t == 0:
        return True
    elif miller(n):
        return do_tests(n, t-1)
    else:
        return False

def chk_expmod(b, ind, m):
    """Power modulus function, calls a test for non-trivial square-roots."""
    z = 1
    while ind != 0:
        if even(ind):
            b *= b
            ind //= 2
            b = chk_rt(b,m)
        else:
            z *= b
            ind -= 1
    return z % m

def chk_rt(x, n):
    """Checks for a non-trivial root."""
    if x !=1 and x != n-1 and square(x) % n == 1:
        return 0
    else:
        return x

def next_p(a, b, primes):
    """Runs tests until a list of primes length b can be returned."""
    if b == 0:
        return primes
    elif is_prime(a):
        primes.append(a)
        return next_p(a + 2, b - 1, primes)
    else:
        return next_p(a + 2, b, primes)

# main
if __name__ == "__main__":
    main()
```

Fermat's Little Theorem centers on an exponent modulus function. To find expmod(x, y, m), we raise x to the power y and give the remainder when this is divided by m.

If n is prime and x is a positive whole number less that n, then expmod(x, n, n) will be the same as x modulus n. If n is not prime, then most values for x will fail the test.

This allows us to make a "probabalistic test". There is the possibility that we will have some false positive results, but if we run the test a few times and n passes, there's a very good chance that it is prime.

There is a problem with the Fermat test, however. There are some numbers (called the Carmichael numbers[1] which are not prime, but which will pass the Fermat test.

Miller and Rabin's algorithm uses a modified version which uses expmod(x, n-1, n). If this equals 1, then n is probably prime.

[1] The first five are: 561, 1105, 1729, 2465 and 2821.

There is one more part of the procedure, we have to check to see if we get a non-trivial square root of 1 modulus n during the expmod function. This is why the squaring part of chk_expmod() calls chk_root(). If the value here is not equal to 1, n-1 or some number whose square modulus n is 1, then n is not prime.

I've built this program to either check a number for primality, list the next primes found over a certain value, or check Mersenne numbers.

Mersenne numbers have the form $2^p - 1$. The French mathematician Marin Mersenne showed that if p is prime, then $2^p - 1$ is potentially prime. The largest known primes are Mersenne numbers, with the current record-holder being $2^{43,112,609} - 1$.

I hope these programs have sparked your interest in primality!