Algorithms - a simple introduction in Python: Part Six

... software is the nearest thing to magic that we've yet invented. It's pure "thought stuff" – which means that it enables ingenious or gifted people to create wonderful things out of thin air. All you need to change the world is imagination, programming ability and access to a cheap PC.

John Naughton, writing in *The Observer*

The Fibonacci Series

Leonardo Fibonacci was a brilliant Italian mathematician. It was he who popularised the use of Arabic numerals (as opposed to Roman numerals) in the West. His most famous work concerns an interesting series of numbers. The series begins with 0 and 1. Each following term is made up by adding together the preceding two numbers.

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...
```

The numbers in this series turn out to have very interesting properties. For instance, the further along the series you go, the closer the result of dividing one number by the next comes to the "golden ratio". Also, many naturally occurring structures seem to echo the Fibonacci series - for instance the spirals of the seeds in sunflower heads.[1]

Anyway, here we are mainly concerned with generating the numbers themselves. Let's say we want a program that, when we give it a number, n, will return the nth Fibonacci number (zero is the 0th term, 55 the 10th etc.)

```python
# fib_r.py - calculates the Fibonacci series, recursively,
# using command line arguments

import sys

def main():
    if (len(sys.argv) != 2):
        usage()
    else:
        try:
            num = int(sys.argv[1])
            print("Fibonacci of", num, "is", fib(num))
        except ValueError:
            usage()

def usage():
    """Tells user how to run the program."""
    print("Usage: for some positive integer, n:\nfib_r.py n")

def fib(n):
    """Takes an integer n and returns the nth Fibonacci number"""
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)

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

This program is different from the ones we have looked at so far in that it uses command-line arguments.

[1] To read more about this, visit: http://www.turingsunflowers.com

To use it, open a terminal in the directory where the file is saved. Then type (assuming python3 is the command to launch Python):

```
python3 fib_r.py 11
```

This runs the program and passes 11 to it. It should then give you the result. To make the program work this way, we need to import “sys”. Any arguments passed to the program are stored in a tuple: “sys.argv”. The 0th element in the tuple is always the program name, so we are interested in the next item. The program takes whatever is at sys.argv [1] and tries to turn it into an integer. If there are more or less than 2 items in sys.argv, or the program cannot make an integer out of what it finds, the usage() function is called.

Once again I have used a recursive algorithm here. It should work well for small-ish values of n. However it is very inefficient, not least because having *two* recursive calls to fib() means some values are computed many times.

For example:

$$\begin{array}{llcccccc}
\text{fib}(5) & = & & \text{fib}(4) & & & + & \text{fib}(3) \\
 & = & \text{fib}(3) & + & \text{fib}(2) & & + & \text{fib}(2) + \text{fib}(1) \\
 & = & \text{fib}(2) + \text{fib}(1) & + & \text{fib}(1) + \text{fib}(0) & & + & \text{fib}(1) + \text{fib}(0) + 1 \\
 & = & \text{fib}(1) + \text{fib}(0) + 1 & + & 1 + 0 & & + & 1 + 0 + 1 \\
 & = & 1 + 0 + 1 & + & 1 & & + & 2 \\
 & = & 5 & & & & &
\end{array}$$

Task

Write a program that asks how many of the Fibonacci numbers the user would like to see. Use an iterative algorithm to produce the numbers and print them to the screen.

Extension

There’s another nice way to generate Fibonacci numbers.

It turns out that, if we take a to stand for the current Fibonacci value and b to be the previous one, the following transformation will give us the next pair of terms:

$$\begin{gathered} a' = bq + aq + ap \\ b' = bp + aq \end{gathered}$$

The initial values of p and q are 0 and 1. If we use a = 21 and b = 13, we get:

$$\begin{gathered} a' = 13 + 0 + 21 = 34 \\ b' = 0 + 21 = 21 \end{gathered}$$

Now, if we were to perform that transformation *twice*, we would in effect be squaring the process. The new transformation has the same form as the old one, but now there are new values p’ and q’:

$$\begin{gathered} p' = p^2 + q^2 \\ q' = q^2 + 2pq \end{gathered}$$

For example, let us take a = 34, b = 21, p = 0 and q = 1.

```
p' = 0 + 1
q' = 1 + 0
```

If we perform the transformation on a and b using p' and q', we get:

```
a' = 21 + 34 + 34 = 89
  b' = 21 + 34 = 55
```

This is precisely what we wanted - the new transformation has given us fib(11) and fib(10) from an input of fib(9) and fib(8).

Now that we can "square" the transformation, we can write a program that uses the same "successive squaring" concept that we used in our exponentiation program.

```
# fib_alg.py Using the algebraic method.

def main():
    print(fib(int(input("Fibonacci of: "))))

def square(x):
    """Returns x times x."""
    return x * x

def even(x):
    """Returns True only if x is even."""
    if x % 2 == 0:
        True
    else:
        False

def fib(x):
    """This function launches fib_iter() with the correct starting values
      for a, b, p and q. The user-supplied number is used as a counter."""
    return fib_iter(1, 0, 0, 1, x)

def fib_iter(a, b, p, q, count):
    """This function uses successive squaring and the
        algebraic method of finding Fibonacci numbers."""
    if count == 0:
        return b
    elif even(count):
        return fib_iter(a,b,square(p)+square(q),square(q)+2*p*q,count/2)
    else:
        return fib_iter(b*q+a*q+a*p,b*p+a*q,p,q,count-1)

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

As always, type the code in. Make sure you understand how it works and check that it gives you the expected results.

If you are unclear about anything, remember that all the Python documentation can be found at www.python.org.