Algorithms - a simple introduction in Python: Part Five

Discover new things. Try new ways of doing things. Give yourself a challenge. Remember: *if you can imagine it happening on your computer screen, you can program it!*
Dan Gookin: C All-in-one Desk Reference for Dummies.

Okay, this time we are going to look at a really important function - factorial of n (also written as n!). The factorial of a positive integer is that number multiplied by all the integers below it.[1]

So, for example, the factorial of 5 is:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

As you may know, before digital computers were invented, the word “computer” referred to humans who operated simple mathematical machines (or perhaps used slide-rules!) to perform calculations. Even though the definition of factorial is very simple, calculating it “by hand” is quite hard work. The values increase very sharply. For instance:

$$20! = 2{,}432{,}902{,}008{,}176{,}640{,}000$$

Perhaps that helps explain why it’s a good idea to get the computer to work out n! by itself.

```
# factorial.py
# using a recursive function to calculate n!

def factorial(x):
    """This function keeps calling itself, until it gets to 0."""
    if num == 0:
        return 1
    else:
        return x * factorial(x-1)

# main
n = int(input("Factorial of: "))
print(str(n) + "! = " + str(factorial(n)))
```

As you can see, this is a recursive function. When factorial calls itself, it has to “remember” that whatever result it gets back needs to be multiplied by the current value of x. The process is shown below. As you can see it gets wider and wider the deeper the recursion goes. All this information has to be stored somewhere (usually in “the stack”). So eventually you will run out of space.

factorial(5)
5 * factorial(4)
5 * 4 * factorial(3)
5 * 4 * 3 * factorial(2)
5 * 4 * 3 * 2 * factorial(1)
5 * 4 * 3 * 2 * 1 * factorial(0)

When x = 0, the function returns 1. This value is then fed back up the chain of function calls:

5* 4 * 3 * 2 * 1 * 1 = 120

[1] The factorial of zero is considered to be 1.

When dealing with a very large number, this can cause problems.

Try running your program and asking for 999!

Task

I want you to re-write the function to use an iterative approach. An iterative version will work out a running total as it goes, so it will not need to store all of those other pieces of data in the stack.

HINT: use a while or for loop to go through all the integers between 1 and n.

Now try running your program and see if it can cope with 1000!.