Python Practice 3: Using loops to solve useful tasks in Python

This lab class gives you a first chance to use loop constructs in Python, and will illustrate how programs can be used to solve some maths and engineering problems. First, recall the basic form of a Python *while*-loop, as shown schematically on the right. Here, the loop continues to repeat so long as the condition is satisfied (i.e. returns `True`):

```
while CONDITION:
    CODE-BLOCK
```

Type the example on the right into a file and run it to see what it does. Now modify it, to see what happens if the condition is false before the loop begins. Is the code-block executed once or not at all? What happens if the condition is never going to become false, e.g. if the test was `i < 100`? (**NOTE**: you can interrupt a code run by clicking the "terminate" icon (a small red circle, with X inside) at the top right-hand side of the shell.)

```
i = 55
while i > 10:
    print(i)
    i = i * 0.8
```

1 Introductory exercises

Create a file for your solutions to the next two exercises.

factorial

The *triangular number* of a positive integer n is the sum of values from n down to one, i.e. $n+(n-1)+\ldots 2+1$. The function on the right uses a *while*-loop to compute this value. Type this definition into a code file, and test it. Now create your own definition for a *factorial* function. Recall that the factorial of a positive integer n is the *product* of the values from n down to 1, i.e. $n\times(n-1)\times\ldots\times 1$. The factorial of 0 is a special case, whose value is 1 by definition.

```
def triangular(n):
    trinum = 0
    while n > 0:
        trinum = trinum + n
        n = n - 1
    return trinum
```

the guessing game

The code file `guessing_game.py` (available from the module homepage in usual way), contains the skeleton of a function definition for a *guessing game*. Save this into your own filespace and inspect it. The code imports a function `randint` (from the `random` module), which is used to generate a random integer, e.g. a call `randint(1,20)` returns a random integer in the range 1 to 20 (where 1 and 20 are also possible values). The function has parameters for (i) the number of guesses allowed, and (ii) the range from which the code picks a number to be guessed.

Complete the definition of the `guess` function, so that it behaves as illustrated in the dialogues below, i.e. telling the player how many guesses they have left, reading in their guess, telling them if their guess right, and if not, whether it is *too low* or *too high*. You code will use a *while*-loop to make sure the player is only given the allowed number of attempts.

NOTE: a tricky issue is handling the case where the player guesses correctly while further attempts remain, as here the loop must *terminate early*. Since this is a function, a `return` statement could be used to end the entire function call, but then the final "`GAME OVER`" statement of the definition would not be reached. Instead, you can use the special *loop-control* command `break`, which causes the loop to end, with execution proceeding to the statements that follow the loop.

```
>>> guess(3,10)
Welcome! Can you guess my secret number?
You have 3 guesses remaining
Make a guess: 4
No - too low!
You have 2 guesses remaining
Make a guess: 7
Well done! You got it right.
GAME OVER: thanks for playing. Bye.
>>>
```

```
>>> guess(3,20)
Welcome! Can you guess my secret number?
You have 3 guesses remaining
Make a guess: 12
No - too low!
You have 2 guesses remaining
Make a guess: 16
No - too high!
You have 1 guesses remaining
Make a guess: 14
No - too low!
No more guesses - bad luck!
GAME OVER: thanks for playing. Bye.
>>>
```

As you test your definition, you might ponder the following questions: (1) Is there an *optimal strategy* to follow, i.e. that gives the best chance of success? (2) Is there a number of guesses that can *guarentee* success on a given range size (one that is less than the range size itself)?

2 Solving equations using Newton's method

Create a new code file `lab3_sec2.py`, for your solutions to this section's exercises.

Newton's method is an effective way to find the roots of an equation, and you can find out about it on Wikipedia: `http://en.wikipedia.org/wiki/Newton's_method`. We begin by implementing one of the examples described there in Python: *calculating the square root of a number.*

To solve an equation $f(x) = 0$, we need to find the value of x for which the function $f(x)$ evaluates to zero. In Newton's method we start off with an approximation x_0 of x, and then use the function f and its first derivative f' to produce a better approximation x_1. This process repeats with each x_{i+1} being computed from its predecessor x_i as follows: $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

This continues until the value produced is a satisfactory approximation of the required root. This type of iteration is very common in engineering problems (such as finite element analysis where large systems of linear equations are solved to within an error tolerance). From the point of view of programming, this is an ideal candidate for a `while` loop, because we need to keep calculating successive approximations while our current approximation is outside the error tolerance.

2.1 Using Newton's method to compute square roots

Applying this method to find the square root of A, we set the function f to be:

$$f(x) = x^2 - A$$

i.e. so that $f(x) = 0$ when x is the desired square root value.

This in turn gives first derivative and 'approximation' functions as follows:

$$f'(x) = 2x \qquad\qquad x_{i+1} = x_i - \frac{x_i^2 - A}{2x_i}$$

Our pseudocode for solving the problem might be:

```
1. start with value A for which to solve (parameter)
2. set an initial value for our approximation x
3. set error tolerance E
4. while difference between $x^2$ and A is greater than E
    4.1 calculate a new value of x
5. display the result
```

Working in your solutions file `lab3_sec2.py`, define a Python function `MySqrt` to compute square roots using Newton's method. (We call it `MySqrt` to avoid confusion with the function `sqrt` that comes with python's `math` library.) This has one parameter `A`, the value whose square root we seek. Your definition, then, might start as follows:

```
def MySqrt(A):
    x = 1.0      # initial approximation value
    errorTolerance = 0.001
         :
         :
```

Here we have chosen an initial value of 1. For square roots Newton's method is *stable*, so it does not matter too much what you choose as an initial value. We have chosen an error tolerance of 0.001, which will be fine for approximating the square root of numbers larger than 1, but may be no good for smaller numbers. (Why not?)

The tricky bit of the code is how to implement the `while` loop condition. Our pseudocode says:

```
4. while difference between $x^2$ and A is greater than E
```

which perhaps suggests a `while` statement in Python such as:

```
while (x * x - A) >= errorTolerance:
```

Note that we have used brackets to make the code easier to read. The problem with this condition is that, at any stage, the current approximation `x` might be either an *underestimate* or *overestimate* of the true value. If it is an underestimate, then the result of `(x * x - A)` will be *negative*, and hence a lower value than `errorTolerance`, but this is not the point. What matters is the *magnitude* of `(x * x - A)` — a small negative value might satisfy the requirement of being *within tolerance*. What we really want to compare against `errorTolerance` is the *absolute* value of `(x * x - A)`, regardless of whether it is positive or negative. The python function `abs` computes the absolute value of a number (i.e. is such that `abs(1)` and `abs(-1)` both return 1).

The other problem we have is how to calculate the new value of `x` with each repeat of the `while` loop. Directly implementing the equation for Newton's method above might give us the following, where `xNew` is the new approximation, computed in terms of the old approximation `x`:

```
xNew = x - (x * x - A) / (2 * x)
```

The new value now becomes the current approximation, so we might extend the above to become:

```
        xNew = x - (x * x - A) / (2 * x)
        x = xNew
```

In practice, this is equivalent to just saying:

```
        x = x - (x * x - A) / (2 * x)
```

as the calculation on the r.h.s. of “`=`” here is made using the value of `x` at the time that the statement is executed, and only when the calculation is done is the result value assigned to the variable indicated to the left of “`=`”, which here happens to be `x`.

We have now seen enough for you to complete your definition of `MySqrt`. When it has computed its result, your definition should *return* the value computed (rather than just printing it).

Whilst you are developing your definition, include a `print` statement immediately after the `while` statement that prints the current value of `x` at the start of each loop, to help you check that your code is running as you expect. This can then be removed when your definition is complete and working. Test your definition thoroughly, trying a range of different input values.

2.2 Using Newton's method to compute cube roots

Define a function MyCubeRoot, which uses Newton's method to approximate cube root. The approach will be similar to that used in `MySqrt`, except with different functions f and f', i.e.:

$$f(x) = x^3 - A \qquad\qquad f'(x) = 3x^2$$