Programming Project 9

Part 1 - Newton's Method

Newton's method is a means of finding progressively more accurate roots of any real function. While finding the roots of equations requires the use of derivatives, finding the root of a real number is quite simple. Newton's method can be reduced to the following simple equation:

$$next\_x = \frac{x + \frac{a}{x}}{2}$$

where a is the number we want to find the square root of, and x is any starting number. next_x becomes x for the next iteration of the equation.

We will use this equation to create an encapsulated function square_root(a, x) which finds the square root of a number to some error of our choosing.

> *Note:* We compare with an error rather than with the actual solution (math.sqrt(a)) in case the solution is a floating point number. Floating point numbers cannot be directly compared because some numbers simply can't be represented by the IEEE floating point standard. This standard represents real numbers with infinitely many decimal places by using a formulaic representation. This allows a computer to store an infinitely large number without using an infinite amount of memory! IEEE floats have increasingly larger gaps between the numbers they can represent as those numbers get further from 0. Because of these gaps, most rational numbers (1/3) and irrational numbers (sqrt(2)) can't be represented exactly using floating point representations they can only be approximated.

Set error = 0.000001. This ensures that the root returned by our square_root function will be within 0.000001 of the true value of the root. To accomplish this, you must loop, using the equation above, until the difference between your y value is less than error away from math.sqrt(a). Recall that the difference between two numbers can be found using abs(num1 - num2).

Your function should return the square root that it found. It should also print the number of iterations (passes through the loop) it required to find that root.

Example output:

```
print square_root(9, 5)
Number of iterations: 2
3.0
print square_root(9, 7000)
Number of iterations: 12
3.0
print square_root(5, 3)
Number of iterations: 5
```

```
2.2360679775

print square_root(5, 7000)

Number of iterations: 16

2.2360679775
```

Note that the number of iterations increases the more your initial estimate differs from the actual solution, but not greatly. Newton's method is highly effective in reducing estimates down to the actual root.

Part 2 – Testing

Now, we will write a test to compare the result of our square_root function to that of the built-in math.sqrt() function. Write a function named test_square_root that prints a table like this:

```
n  square_root        math.sqrt          difference         is difference < 0.000001?
1  1.0                1.0                0.0                1
2  1.41421356237      1.41421356237      2.22044604925e-16  1
3  1.73205080757      1.73205080757      0.0                1
4  2.0                2.0                0.0                1
5  2.2360679775       2.2360679775       1.88293824976e-13  1
6  2.44948974278      2.44948974278      6.2172489379e-15   1
7  2.64575131106      2.64575131106      0.0                1
8  2.82842712475      2.82842712475      4.4408920985e-16   1
9  3.0                3.0                0.0                1
```

To print this table, comment out the line that prints the number of iterations taken by square_root if necessary. Use the following formatting to print the table:

```
import sys

sys.stdout.write("{:<3}{:<20}{:<20}{:<20}{:<20}\n".format('n', "square_root",
                                    "math.sqrt", "difference", "is difference < 0.000001?"))
```

This uses the underlying C implementation of standard output to console. It's a simple way of printing a table, but we could also use the pprint pretty printing module, old- (ex., '%#.17g' % 1.2) or new-style (ex., string.format()) string formatting.

In the column "is difference < 0.000001?" I'm printing 1 for True and 0 for False. You can print the boolean values (True and False) if you'd prefer.

Note that in Test-Driven Development (TDD), this test would have been written *before* the square_root function was written. In the case of TDD, we would likely have asserted that our solution is within error of the built-in modules. This allows us to create a set of specifications that our function must meet, and we know our function is finished once it fulfills all of those specifications.

Part 3 - Iteration Redux

Now that we've learned how to iteratively improve an approximation, we're going to implement the following formula in code:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!\,(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Remember that summation is just a loop. Note that this is a sum from 0 to infinity. This is because it's attempting to find an exact irrational number, which is infinitely long. Since computers can't represent numbers to infinite precision, we're going to use an error, just like we did in part one. Use an error of 0.000001.

Similar to the testing we did in part two, compare your result to the built in math.pi, ensuring that your error is less than 0.000001.

Exercises adapted from Downey, Allen. *Think Python*. Sebastopol, CA: O'Reilly, 2012. Print.