CS5001-5003, Homework 4, Fall 2022

1. Design and implement a Python 3 **function** (i.e., not a complete *program with a main()),* in a file called **root.py**, to compute the ***nth*** *root* of a number. Your file should contain just the top-of-file documentation string, one function definition, and other associated documentation.

The function **root** should accept 2 required parameters and one optional[1] parameter, as follows:

- **n**, an integer > 1 representing which root to calculate;
- **number**, an integer *or* float >= 1 whose root is to be calculated;
- (optional) **accuracy**, a float between 0.0 and 1.0, defaulting to 0.001.

If any of the actual parameters provided by the caller violate these constraints, the function should simply return -1. (This signals an error, since all valid solutions to the problem, as stated, are positive). All such information should be included in the documentation string[2] for your function.

If **n** == 2, root should return the square root of **number**. If **n** == 3, it should return the cube root; and so on. Given valid arguments, **root** should return a ***positive float*** which, when raised to the **n**th power, will fall **within accuracy** of **number**.[3] By definition, the **n**th root of 1 is 1.

Here are some examples of acceptable input/output behavior for this part.

```
>>> root(2, 4)
  2.0

>>> root(3, 27.0, 0.01)
  2.9999

>>> root(100, 1)
  1.0

>>> root(5, 32)
  2.0

>>> root(2, -1)
  -1
```

Naturally, use good, "Pythonic" coding style, including choosing mnemonic, snake_case variable names and complying with PEP8 style rules, such as indenting 4 spaces or leaving 2 blank lines around function definitions. Additionally, there are some **constraints on how you solve this**.

[1] For a reminder about defining functions with optional parameters with default values, see https://realpython.com/python-optional-arguments/#using-python-optional-arguments-with-default-values.
[2] For example, see https://www.pythontutorial.net/python-basics/python-function-docstrings/.
[3] Hint: abs(number – guess) < accuracy

Your solution ***must not*** involve raising any quantity to a fractional power (*e.g.,* **number****0.5). Also, you ***must not*** **import math** or other Python libraries. Instead, you are expected to use a "guess and check" approach, that:

- does not merely guess randomly;
- avoids repeated guesses;
- converges on sufficiently accurate result *faster* than just trying every increment of 0.001.

Here is a sketch of the basic idea. Start by establishing upper and lower bounds for the desired root. For example, you know your first guess has to be at least 1 and not more than **number**. Check whether each guess is too low, too high, or within the accuracy required, by raising the guess to the **n**th power. If it is too low, increase your lower bound. If it is too high, decrease your upper bound. Keep your next guess within your current upper and lower bounds. Continue narrowing it down. Remember that children's game, "I'm thinking of a number between 1 and 100 – can you guess my number?" If you play that game starting from 1 and just continue increasing by 1 until you guess right, you will eventually guess the number (unless the child is cheating!), but it will take an average of 50 guesses. A better algorithm should require far fewer guesses, on average. Try your best to **develop your own solution** rather than spending your time googling for someone else's work.

2. In addition to the Python documentation strings for your file and function, as required in part **1**, please provide extensive inline commentary for your code. These comments should be intended for *another software engineer* who has been asked to modify your code, or add new features, whereas doc strings are intended for the *end user* of your function. Although excessive documentation can often be almost as bad as insufficient documentation, for the purposes of this question, only, we are looking for an average of ***1 inline comment per every 2 lines of code***. Remember that this is to help your instructors ensure that you really understand what your code is doing. Such a high density of comments would usually be inappropriate in an actual software engineering job assignment. We are not looking for perfect English grammar, but we are looking for clarity in explaining what your program is doing, the implicit assumptions it is making, and so on.

3. In a *separate file* called **root_test.py**, provide a "test suite" for **root.py**. Be sure to test for the examples shown above, as well as "edge"/"corner"/"boundary" cases[4], various values of **n** (*e.g.,* 7th root?), ints vs. floats (where appropriate), invalid arguments, and different settings for **accuracy**, including invocations of **root**() that provide a specific value for **accuracy** and invocations that just take the default of 0.001. For each test, print a summary, including arguments provided, **expected** return value, and **actual** return value. Provide counts of **how many tests failed** as well as **how many tests were run**. Minimize repetitive coding by including a separate function, **run_test()**, with suitable arguments and return value(s), to run your tests, which should be called by **main()**.

[4] Hint: https://softwareengineering.stackexchange.com/questions/125587/what-are-the-difference-between-an-edge-case-a-corner-case-a-base-case-and-a-b.

Homework 4 Grading Rubric

Question1	Points	Total
Your file should contain just the top-of-file documentation string, one function definition, and other associated documentation	5	50
The function root should accept 2 required parameters and one optional parameter, as follows:		
~~**n**, an integer > 1 representing which root to calculate;~~	2	
~~**number**, an integer *or* float >= 1 whose root is to be calculated~~;	2	
accuracy, a float between 0.0 and 1.0, defaulting to 0.001.	2	
If any of the actual parameters provided by the caller violate these constraints, the function should simply return -1.	4	
Style: use good, "Pythonic" coding style, including choosing mnemonic, snake_case variable names and complying with PEP8 style rules, such as indenting 4 spaces or leaving 2 blank lines around function definitions.	10	
Code does not involve raising any quantity to a fractional power (e.g., number**0.5)	2	
Code does not **import math** or other Python libraries	2	
Code uses a "guess and check" approach, that:		
• does not merely guess randomly;	2	
• avoids repeated guesses;	2	
• converges on sufficiently accurate result *faster* than just trying every increment of 0.001.	2	
Code complies without errors	5	
Code is correct/produces accurate solutions	10	
Question 2	Point	Total
Average 1 inline comment per every 2 lines of code	10	20
Comments demonstrate high-level understanding of code	10	
Question 3	Points	Total
Each test prints a summary, including arguments provided, expected return value and actual return value	5	30
Code contains counts of **how many tests failed** as well as **how many tests were run**	5	

Code contains run_test() function, with suitable arguments and return value(s), to run your tests	2	
run_test() function called by main()	2	
Code contains tests for:		
root(2,4) == 2.0	2	
root(3,27.0,0.01) == 2.9999	2	
root(100,1) == 1.0	2	
root(5,32) == 2.0	2	
root(2,-1) == -1	2	
Code checks for "edge"/"corner"/"boundary" cases, various values of n, ints vs. floats, invalid arguments, and different settings for accuracy, including invocations of root() that provide a specific value for accuracy and invocations that just take the default of 0.001	6	