ENGR 102 – Fall 2022
Lab: Topic 4 (individual) [27 points =100%

Deliverables

Please complete the four programs described below. Each program is an individual assignment, but you may consult your teammates and others as you work. Please submit the following files to zyBooks.
Do not forget to include individual header.

- `largest_number.py` [5 points]
- `roundoff_error.py` [2 points]
- `how_many_gadgets.py` [10 points]
- `calculate_roots.py` [10 points]

Activity #1: Largest number - individual
Using conditional statements (i.e. statements like `if-elif-else`) write a program named largest_number.py that takes as input from the user three (3) numbers from the keyboard and prints the value of the largest number. Your code should output in the format shown below.

Example output (using inputs **1**, **2**, **3**):
```
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
The largest number is 3.0
```

Activity #2: Roundoff error - individual
When performing numerical computations, one of the challenges you can run into is floating-point roundoff error. This occurs when the computer needs to represent a number that would require an infinite number of decimal digits, but rounds them off after some point. That small roundoff error can cause some significant issues. This activity is meant to help you understand floating-point roundoff error a bit more and learn about one way of dealing with it. Create a Python file named roundoff_error.py for the following three-part activity. *Please separate the various parts of your code with the following comment identifying the separate sections (copy/paste into your file with the appropriate letter)*.
```
############ Part A ############
```

Part A: Identifying floating-point problems
First, **type** and run the following program:
```
    a = 1 / 7
    print(f'a = {a}')
    b = a * 7
    print(f'b = a * 7 = {b}')
```
Notice that the value of `a` is rounded off. The value of `b`, if we have no roundoff, should be `1`. Is it? *Make a comment in your code answering the question*.

Now add the following lines:
```
    c = 2 * a
    d = 5 * a
    f = c + d
    print(f'f = 2 * a + 5 * a = {f}')
```
In this case, the value of `f`, if we have no roundoff, should be `1`. Is it? *Make a comment in your code answering the question*.

Finally, add the following lines:

```
from math import sqrt
x = sqrt(1 / 3)
print(f'x = {x}')
y = x * x * 3
print(f'y = x * x * 3 = {y}')
z = x * 3 * x
print(f'z = x * 3 * x = {z}')
```

Again, the values of `y` and `z`, if we have no roundoff error, should be `1` in both cases. Are the values 1? *Make a comment in your code answering the question*.

Was that surprising? You should have seen from those examples that sometimes we will encounter issues due to roundoff error and sometimes we won't. We can't always predict when roundoff error will be obvious.

Part B: Tolerances for Comparisons

In Part A, you should have seen that two different ways of computing values that should be identical might actually produce values that are different, if only by a tiny bit. In some cases, this is not a problem. For example, we usually don't care if speed is incorrect in the 10th decimal place, since we usually can't measure speed that precisely anyway. But, floating point error can become a big problem if comparisons are made with floating-point values.

A common way for dealing with floating-point error is to use **tolerances**. Tolerances let you compare two values that are close, but not identical to each other. Rather than checking whether or not `a == b`, we instead compute the absolute value of the difference in `a` and `b` and see if that quantity (the difference) is within some small distance away from `0` (either above or below). That small value, called the tolerance, is decided upon by the programmer and the value chosen is highly dependent on the problem at hand. If the absolute value of the difference between `a` and `b` is less than the tolerance, then we may consider `a` and `b` to be equal for the problem at hand. The tolerance is commonly abbreviated TOL or EPS (short for epsilon (ε)). 1e-6 is usually a good value for tolerance, but may vary with specific applications.

Add to your code from Part A the following lines to compare values of the variables `b` and `f` using the concept of tolerance:

```
TOL = 1e-10
# check if b and f are equal within specified tolerance
if abs(b - f) < TOL:
    print(f'b and f are equal within tolerance of {TOL}')
else:
    print(f'b and f are NOT equal within tolerance of {TOL}')
```

Add a similar tolerance check to your code for `y` and `z`.

As you write programs, think about your comparisons and decide if you need to use tolerances. If you are making a comparison to check for exact equality and you might have some floating-point error, you will probably want to use tolerances, while if you are just checking which of two things is larger, a tolerance comparison is likely unnecessary. Tolerances are particularly helpful when checking things like whether a denominator is (nearly) 0, and thus a division is likely to create error.

Part C: Illusions of Precision

Computers operate in base 2 (binary). No matter how many digits you are willing to use, some numbers still cannot be represented exactly, like the number 0.1. Most people aren't aware that 0.1 is stored as an approximation because Python keeps the number of digits manageable by displaying a rounded value. Even though the printed result of a calculation may look exact, know that the actual stored value is the nearest representable binary fraction.

Add to your code from Part B the following lines:

```
m = 0.1
print(f'm = {m}')
n = 3 * m
print(f'n = 3 * m = 0.3 {n==0.3}')
p = 7 * m
print(f'p = 7 * m = 0.7 {p==0.7}')
q = n + p
print(f'q = n + p = 1 {q==1}')
```

Did the results surprise you? If you rewrote your program for Lab Topic 1 Activity 3 using variables and successive divisions for x (`x = 1 + 1/10`, `x = 1 + 1/100`, etc) would you expect to see the same output as your original program? Check out this link to the Python documentation: https://docs.python.org/3/tutorial/floatingpoint.html. This info helps explain these issues. As the document states, ". . .this is not a bug in Python, and it is not a bug in your code either." Instead, the problems stem from the way a floating-point number is represented by the hardware. Awareness of these issues may save you a lot debugging effort in the future. Also check out this link: Binary Tutorial - 5. Binary Fractions and Floating Point
https://ryanstutorials.net/binary-tutorial/binary-floating-point.php

Activity #3: How many gadgets - individual
Assume a machine during its initial testing phase produces 5 gadgets a day. After 10 days of testing (starting on day 11), it begins to run at full speed, producing 50 gadgets per day. After 50 days at full speed (days 11-60), it gradually starts becoming less productive, and produces 1 fewer gadget per day, (49 gadgets on day 61, 48 on day 62, etc) until on day 101 it stops producing gadgets. Write a program named how_many_gadgets.py that reads in a day (as a number) from the keyboard and reports the total number of gadgets produced from the initial testing phase up to and including the day entered. For example, entering 3 would report 15 gadgets.

Your code should also . . .

- Check for inappropriate day numbers and message the user accordingly
- Echo the input in the output when you report the number of gadgets
- Use the output format shown below

Note: Do **NOT** use a loop. Part of the challenge in this program is for YOU to work out the model for how to compute gadgets produced in total, given the above information. Hint: Solve this problem by hand on paper first, then translate your solution to Python. **A graph or diagram is extremely helpful**. This approach is much easier than debugging some lousy Python code written before you understand the problem.

Example output (using input **3**):
```
Please enter a positive value for day: 3
The total number of gadgets produced on day 3 is 15
```

Example output (using input **-1**):
```
Please enter a positive value for day: -1
You entered an invalid number!
```

Activity #4: Calculate roots - individual
The roots of a quadratic equation are the values of x at which the equation evaluates to 0. The well-known quadratic formula is often used to find these roots. Write a program named calculate_roots.py that takes as input the coefficients A, B, and C and outputs the roots of that equation. Be aware of the following:

- Use the output format shown below
- If the roots have an imaginary component (complex), use i when representing the imaginary term in the output. For example, you may output "$3.0 + 7.0i$" as a root.
- Be sure to handle the cases in which any or all coefficients are equal to zero
    - If `A != 0`, there could be 2 real distinct roots, 2 real identical roots (one root of multiplicity 2), or 2 complex roots
        - If there are two roots, output the larger root first
    - If `A = 0`, we are left with $Bx + C = 0$, a linear equation with one root
    - If `A = B = 0`, we are left with `C = 0`, so if the user entered a non-zero value of `C`, write a message to the screen indicating this error

Example output (using inputs **1**, **2**, **1**):
```
Please enter the coefficient A: 1
Please enter the coefficient B: 2
Please enter the coefficient C: 1
The root is x = -1.0
```

Example output (using inputs **1**, **2**, **-3**):
```
Please enter the coefficient A: 1
Please enter the coefficient B: 2
Please enter the coefficient C: -3
The roots are x = 1.0 and x = -3.0
```

Example output (using inputs **0**, **0**, **1**):
```
Please enter the coefficient A: 0
Please enter the coefficient B: 0
Please enter the coefficient C: 1
You entered an invalid combination of coefficients!
```