Lab2 (Sep 21)

This lab is on object-oriented programming in Python.

Academic honesty and standard:

This is an individual assignment. The work you submit must be your own work. Obviously you should be able to fully explain and describe possible alternatives for any line of code you include in your submission. Do not share any part of your code or your design thinking. Remind yourself of the UBC policies on Academic honesty and standard.

Submissions:

CPEN333B: You will submit one python3 file (.py extension) for the implementation of part1 only. Submit the file to the associated Canvas assignment dropbox by the deadline.
CPEN333A: You will submit two python3 files (.py extension), one for the implementation of part1 and one for the implementation of part2. Make sure to clearly include the words part1 or part2 in the name of your submitted files. Submit the files to the associated Canvas assignment dropbox by the deadline.

Some important reminders:

- Don't forget to include your student name and number as comments at the beginning of each .py file.
- Follow the requested specification exactly. We may use automated tests as a part of grading, so every unit (e.g. functions or methods) must behave as specified, and be testable fully on own and should behave as specified.
- For part1, add code only where it is requested. Do not change the code design, format or skeleton. Do not change the function headers.
- For part2, change is allowed (think out what is needed to be changed and document) to be able to reuse the code for the new problem.
- Write code that is readable in the first place, and include helpful comment statements to document your code design and implementation. The program must be well-documented.
- You are allowed to use inner functions. For consistency, do not add additional functions, variables with global scope, or types. or imports.

Due: Thu Sep 21, by 13:59 PM (common to everyone)

Please plan ahead, no late submissions are accepted.

Write the best code you can. For marking, we will test for functionality first (must work), and we will check the code design, readability, documentation, ..

The setup or preparation for this labs is similar to lab1.

Lab room usage rule for Lab 2:

A TA will be in the lab during your scheduled lab time to provide help, and I do recommend that you use the lab time wisely, however, for Lab 2 we will not record attendance and there is no scheduled demo for it.

Part 1: Implementing a simple GUI-based python program for rational numbers

In this lab, you will implement a simple GUI based app that allows us to *add*, *subtract*, *multiply* and *divide* rational numbers. We represent the rational number type using a python class. The class also has a *toString* method that returns a string representation of the rational number.

Note that:

- A rational number is represented by a numerator and a denominator (both integer) in the form numerator/denominator.
- We use a class to define the rational number type. The class uses two fields to store the numerator and the denominator.
- The user can enter any valid rational number, but we store a rational number in its lowest form. We say a rational number is in its lowest form if both the numerator and denominator have no common factor other than 1. In our class, the numerator stores the sign of *this* rational number.
- A rational number cannot have a denominator of 0. We store such a rational number as usual, but we display it as NaN (not a number) when displaying the result of the calculation on our GUI. Note that we are representing *undefined* also as NaN. The result of a division involving NaN should also be NaN.
- Every integer d has an equivalent rational number d/1. We store such a rational number as usual, but we display only the integer value when displaying the result of the calculations on our GUI.
- Assuming that we have $\boldsymbol{r1} = \frac{a}{b}$ and $\boldsymbol{r2} = \frac{c}{d}$, then recall that:
  - $\boldsymbol{r1} + \boldsymbol{r2} = \frac{ad+bc}{bd}$
  - $\boldsymbol{r1} - \boldsymbol{r2} = \frac{ad-bc}{bd}$
  - $\boldsymbol{r1} * \boldsymbol{r2} = \frac{ac}{bd}$
  - $\boldsymbol{r1}/\boldsymbol{r2} = \frac{ad}{bc}$

Consider the following code template. The provided code has two classes: Rational and GUI.

- Use the GUI class as is. It creates the user interface for your program. It is quite unassuming but does the job.
- The initializer method for the Rational class is also given to you. Use as is.
- You are to complete the Rational class by implementing its remaining methods: add, subtract, multiply, divide and toString.

- All of the add, subtract, multiply and divide methods return a new rational number containing the result. Obviously, they must not mutate any of the operands.

See the code for the comments, and watch this video as a demo for the app first:
https://youtu.be/DU63sVj5hHU (https://youtu.be/DU63sVj5hHU)

(https://youtu.be/DU63sVj5hHU)

```
#Lab 2
#student name:
#student number:

from tkinter import *
#do not import any more modules

#do not change the skeleton of the program. Only add code where it is requested.
class Rational:
    """ this class implements the rational number type
        it stores the rational number in its lowest from
        two data fields:
            numerator and denominator
            (numerator stores the sign of the rational)
        Operation:
            add, subtract, multiply and divide
            toString
    """
    def __init__(self, numerator: int, denominator: int) -> None:
        """initizer stores the rational number in the lowest form"""
        def greatestCommonDivisor(n: int, d: int):
            """local method for the greatest common divisor calculation"""
            n1 = abs(n)
            d1 = abs(d)
            result = 1
            k=1
            while k <= n1 and k <= d1:
                if n1 % k == 0 and d1 % k == 0:
                    result = k
                k += 1
            return result
        #rational number must be in the lowest form
        gcd: int = greatestCommonDivisor(numerator, denominator)
        #numerator stores the sign of the rational
        signFactor: int = 1 if denominator > 0 else -1
        self.numerator = signFactor * numerator // gcd
        self.denominator = abs(denominator) // gcd

    def add(self, secondRational):
        """adds 'this' rational to secondRational
           returns the result as a rational number (type Rational)
        """
        # to implement

    def subtract(self, secondRational):
        """subtracts secondRational from 'this' rational to
           returns the result as a rational number (type Rational)
        """
```

```
        # to implement

    def multiply(self, secondRational):
        """multiplies 'this' rational to secondRational
           returns the result as a rational number (type Rational)
        """
        # to implement

    def divide(self, secondRational):
        """divides 'this' rational by secondRational
           returns the result as a rational number (type Rational)
        """
        # to implement

    def toString(self):
        """ returns a string representation of 'this' rational
            the format is: numerator/denominator
            if 'this' rational is an integer, it must not show any denominator
            if denominator is 0, it just returns "NaN" (not a number)
        """
        # to implement

class GUI:
    """ this class implements the GUI for our program
        use as is.
        The add, subtract, multiply and divide methods invoke the corresponding
        methods from the Rational class to calculate the result to display.
    """
    def __init__(self):
        """ The initializer creates the main window, label and entry widgets,
            and starts the GUI mainloop.
        """
        window = Tk()
        window.title("RN") #Rational Numbers

        # Labels and entries for the first rational number
        frame1 = Frame(window)
        frame1.grid(row = 1, column = 1, pady = 10)
        Label(frame1, text = "Rational 1:").pack(side = LEFT)
        self.rational1Numerator = StringVar()
        Entry(frame1, width = 5, textvariable = self.rational1Numerator,
              justify = RIGHT, font=('Calibri 13')).pack(side = LEFT)
        Label(frame1, text = "/").pack(side = LEFT)
        self.rational1Denominator = StringVar()
        Entry(frame1, width = 5, textvariable = self.rational1Denominator,
              justify = RIGHT, font=('Calibri 13')).pack(side = LEFT)

        # Labels and entries for the second rational number
        frame2 = Frame(window)
        frame2.grid(row = 3, column = 1, pady = 10)
        Label(frame2, text = "Rational 2:").pack(side = LEFT)
        self.rational2Numerator = StringVar()
        Entry(frame2, width = 5, textvariable = self.rational2Numerator,
              justify = RIGHT, font=('Calibri 13')).pack(side = LEFT)
        Label(frame2, text = "/").pack(side = LEFT)
        self.rational2Denominator = StringVar()
        Entry(frame2, width = 5, textvariable = self.rational2Denominator,
              justify = RIGHT, font=('Calibri 13')).pack(side = LEFT)

        # Labels and entries for the result rational number
        # an entry widget is used as the output here
        frame3 = Frame(window)
        frame3.grid(row = 4, column = 1, pady = 10)
        Label(frame3, text = "Result:     ").pack(side = LEFT)
        self.result = StringVar()
        Entry(frame3, width = 10, textvariable = self.result,
              justify = RIGHT, font=('Calibri 13')).pack(side = LEFT)
```

```
        # Buttons for add, subtract, multiply and divide
        frame4 = Frame(window) # Create and add a frame to window
        frame4.grid(row = 5, column = 1, pady = 5, sticky = E)
        Button(frame4, text = "Add", command = self.add).pack(
            side = LEFT)
        Button(frame4, text = "Subtract",
               command = self.subtract).pack(side = LEFT)
        Button(frame4, text = "Multiply",
               command = self.multiply).pack(side = LEFT)
        Button(frame4, text = "Divide",
               command = self.divide).pack(side = LEFT)

        mainloop()

    def add(self):
        (rational1, rational2) = self.getBothRational()
        result = rational1.add(rational2)
        self.result.set(result.toString())

    def subtract(self):
        (rational1, rational2) = self.getBothRational()
        result = rational1.subtract(rational2)
        self.result.set(result.toString())

    def multiply(self):
        (rational1, rational2) = self.getBothRational()
        result = rational1.multiply(rational2)
        self.result.set(result.toString())

    def divide(self):
        (rational1, rational2) = self.getBothRational()
        result = rational1.divide(rational2)
        self.result.set(result.toString())

    def getBothRational(self):
        """ Helper method used by add, subtract, multiply and divide methods"""
        try:
            numerator1 = eval(self.rational1Numerator.get())
            denominator1 = eval(self.rational1Denominator.get())
            rational1 = Rational(numerator1, denominator1)

            numerator2 = eval(self.rational2Numerator.get())
            denominator2 = eval(self.rational2Denominator.get())
            rational2 = Rational(numerator2, denominator2)
            return (rational1, rational2)
        except:
            return(Rational(0,0), Rational(0,0)) #if an entry value is missing, cause NaN

if __name__ == "__main__": GUI()
```

Part 2: Implementing a simple GUI-based python program for complex numbers

[see the Submissions section at the top of this page on whether you need to do part2 as well or not]

In this part of the lab, you will reuse the whole code from the template in part 1 of the lab and modify it to implement a program that allows arithmetic operations involving complex numbers (**https://en.wikipedia.org/wiki/Complex_number (https://en.wikipedia.org/wiki/Complex_number)**).

Note that:

- You need to try to keep as much as the structure of the code from the template of part 1 as possible, but you are welcome to modify/improve wherever needed in the code for this part (regardless, do include comments to explain, justify and document).
- You are to use descriptive identifiers (variables, ...). It is fine obviously to just rename when applicable (on VS Code, right click on that variable and choose "Rename Symbol"; if you choose "Change All Occurrences" instead, be careful as it behaves differently).
- Make sure you modify and correct all comments and docstrings.
- You can keep the look of the GUI (with proper modifications and relabeling) or/and you can improve it.
- We only consider complex numbers in the Cartesian notation (that is `a + bi`), where `i` is the imaginary unit.
- A real number is a complex number whose imaginary part is 0.
- You are to implement addition, subtraction, multiplication and division of two complex numbers.
- Assuming that we have $a = x + yi$ and $b = u + vi$, recall that:
  - $a + b = (x + u) + (y + v)i$
  - $a - b = (x - u_+ (y - v)i$
  - $a * b = (xu - yv) + (xv + yu)i$
  - $\frac{a}{b} = \frac{(xu+yv)+(yu-xv)i}{u^2+v^2}$
- When we enter the two complex numbers as the operands (e.g. `a` and `b`), we just need to enter the numbers for their real and imaginary parts. You must represent + and i, similar to how we represented / in part 1. But when we display the result, obviously we need to include + and i (again similar to our representation of rational number where we did include /). It is up to you to correctly represent the number, considering that the sign of the real and/or the imaginary part of the result can be positive or negative. For example, you can display a resulting number like `-1.2-2i` or `-1.2 + (-2)i` ... (be consistent and user-friendly)
- Similar to part 1, we have to enter both the real and imaginary parts of an operand. For example, if any part is 0, we must enter that 0 explicitly like any other number. If either is missing (not entered by the user), we display NaN for the result (when any of the add, ... is clicked). You should find a way to implement it.
  - Also, a divide by 0 results in NaN.
- If the imaginary part of the result is 1, do not display that 1 (for example, display `-1 + i` and not `-1 + 1i`)
- If the result is a real number (that is when we have $x + 0i$), you should only display the value x. If the result is a purely imaginary number (that is when we have $0 + yi$), you should only display the value yi.
- You can use a tool (like the following online tool: **https://www.symbolab.com/solver/complex-numbers-calculator (https://www.symbolab.com/solver/complex-numbers-calculator)**) to test your result.

To get full mark:

- Your program must work. There won't be any credit for a code that does not.
- Your program must be correct. Fully test it.
- Your program must follow the specification (implement what is requested). Your program must not do any unnecessarily or repetitive work.
- Your program must use good code style, be readable, and include "useful" comments statements.

See the submission dropbox for the marking detail.