Intro to Computing Lab 11 Name:_____

Objective: To practice implementing and using classes in Python.

The Assignment Overview

For this lab, you will be learning how to use and implement classes. In part A, you'll examine a Fraction class that stores the numerator and the denominator as exact values. This will give you a more involved example to model from than the simple Coin class discussed in class. In part B, you'll implement a generalized Die class. Your Die class will have methods for constructing, rolling, getting the current roll of the die, adding two dice, and comparing dice. In part C, you will then use this Die class to write a Python program that simulates the rolling of two ten-sided dice 1,000 times. Your program should output the results of your simulation.

For today's lab you'll need several files, so start the lab by copying the directory P:\Math-CS\810-051-fienup\common\lab11 to your P: space (or desktop or flash drive).

Part A: Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are public, but name mangling can provide some private protection if your identifier starts with two underscores, e.g, `self.__sideup`
- all data types are objects, so they can be used as inherited base classes
- most built-in operators (+, -, *, <, >, ==, etc.) can be redefined for a class. This makes programming with objects a lot more intuitive. For example suppose we have two Die objects: die1 & die2, and we want to add up their combined rolls. We could use accessor methods to do this:

  diceTotal = die1.getRoll() + die2.getRoll()

  Here, the getRoll method returns an integer (type int), so the '+' operator being used above is the one for ints. But, it might be nice to "overload" the + operator to work with Die objects, so the programmer could add dice directly as in:

  diceTotal = die1 + die2
- objects are passed by reference
- all classes have a set of standard methods provided, but may not work properly (__.str__, __.doc__, etc.)
- The general syntax of class definition is:

class MyClass [(superClass1 [, superClass2]*)]:
    "Document comment which becomes the __.doc__ attribute for the class"
    def __init__(*self*, [param [, param]*):
        "Document comment for constructor method with *self* be referencing to the object itself"
        #__init__body

    # defs of other class methods and assignments to class attributes

end class MyClass

Consider the following Fraction class (next page) that stores the numerator and the denominator as exact values. Fraction objects can be created and used as:

```
>>> myFraction = Fraction(3,4)
>>> anotherFraction = Fraction(11, 8)
>>> fraction3 = myFraction + anotherFraction
```

a) Modify the Fraction class to include a method that overloads the '*' operator to perform multiplication, called __mul__.

b) Test your code by calling __mul__ to assign `productFraction` the value of `myFraction` and `anotherFraction`?

```
def gcd(m, n):
    "Calculates the greatest-common-divisor using Euclid's Algorithm"
    # Assumes the m and n are greater than zero!!!
    while m%n != 0:
        oldm = m
        oldn = n
        m = oldn
        n = oldm % oldn

    return n
# end def gcd

class Fraction:
    "Simple Fraction class to allow rational number"
    def __init__(self, top, bottom):
        "Constructor for the Fraction class that takes a \
        numerator and denominator as parameters."
        self.num = top
        self.den = bottom
    # end def __init__

    def __str__(self):
        "Overrides the standard method, converts a Fraction to a string"
        return str(self.num)+"/"+str(self.den)
    # end def __str__

    def show(self):
        "Displays a Fractional by printing it"
        print self.num,"/",self.den
    # end def show

    def __add__(self, rhs_fraction):
        "Overrides the '+' operator for Fractions"
        newnum = self.num * rhs_fraction.den + self.den*rhs_fraction.num
        newden = self.den * rhs_fraction.den
        common = gcd(newnum, newden)
        return Fraction(newnum/common, newden/common)
    # end def __add__

    def __cmp__(self, rhs_fraction):
        "Overrides the '__cmp__' operator for Fractions, to allow for \
        a deep comparison of two Fractions"
        num1 = self.num*rhs_fraction.den
        num2 = rhs_fraction.num*self.den

        if num1 < num2:
            return -1
        elif num1 == num2:
            return 0
        else:
            return 1
    # end def __cmp__
# end class Fraction
```

After you have implemented and tested your __mul__ method in the Fraction class, PRINT the modified fraction.py file to be turned in at the end of lab.

Part B: For this part of this lab, you will be developing a generalized Die class that can have any number of sides. The detailed description of this class is in the table below. Put your Die class in its own file called die.py to make a "`die`" module. You'll want import the `random` module for rolling the die randomly.

In a separate file called testDie.py, include a list of commands to thoroughly test your Die class. Your testDie script will need to `import die`. An advantage of the interactive nature of Python is that you can test each Die method as you write them. This allows for good incremental development.

Detail Descriptions of the Die Class Methods		
Method	**Example Usage**	**Description**
`__init__`	`myDie = Die(6)`	Constructs a die with a specified number of sides and randomly rolls it
`__cmp__`	`if myDie == otherDie:`	Allows the comparison operations (>, <, ==, etc.) to work correctly for Die objects. To implement this you need to return -1 if the left-hand-side die's roll is less than the right-hand-side die's roll, or return 0 if they are equal, or return +1 if the left-hand-side die's roll is greater than the right-hand-side die's roll. (see the Fraction class for an example of __cmp__)
`__add__`	sum = myDie + otherDie	Allows the direct addition of Die objects, and returns the integer sum of there current values.
`__str__`	Indirectly as: `myDie.__str__()` or indirectly as: `print myDie`	Returns a string representation for the Die. By overriding the default __str__ method the "print" statement will work correctly.
`roll`	`myDie.roll()`	Rolls the die randomly and return the value rolled
`getRoll`	`myDie.getRoll()`	Returns the current value of the die
`show`	`myDie.show()`	Displays the die's value to standard output

After you have implemented AND fully tested you Die class, PRINT the die.py and testDie.py file to be turned in at the end of lab.

Part C: Using your Die class, write a Python program called dieSimulation.py that simulates the rolling of two ten-sided dice 1,000 times. Your program should output the results of your simulation by printing the percentage of time the dice totaled 2, 3, 4, ..., 20.

For extra credit, you can also include a histogram (bar chart) of asterisks representing the percentages for each possible roll. You may orient the histogram either horizontally or vertically.

After you have implemented your dieSimulation, PRINT it AND its output. In Windows, IDLE has a File | Print Window menu options that you can use.

If you complete all of the activities within the lab period, you should turn in the following printouts.

- a hard-copy of fraction.py from Part A
- a hard-copy of die.py and testDie.py from Part B
- a hard-copy of dieSimulation.py, and
- the output produced by running the dieSimulation.py program.