Lab 4: Dictionaries and Classes

Name: ________________________ University Number:____________

Exercise 1: Canadian Postal Codes

AIM:

In Canada, the first, third, and fifth characters in the postal code are letters while the second, fourth, and sixth characters are digits. We can determine the province or territory in which an address resides from the first character of its postal code as shown in the following table. No valid postal codes currently begin with D, F, I, O, Q, U, W, or Z.

Province/Territory	First Character(s)
Newfoundland and Labrador	A
Nova Scotia	B
Prince Edward Island	C
New Brunswick	E
Quebec	G, H, and J
Ontario	K, L, M, N, and P
Manitoba	R
Saskatchewan	S
Alberta	T
British Columbia	V
Nunavut or Northwest Territories	X
Yukon	Y

The second character in a postal code identifies whether the address is rural or urban. If that character is a 0, then the address is rural; otherwise, it is urban. Write a Python program that prompts a postal code from the user and displays the province or territory associated with it, along with whether the address is urban or rural. Your program should use a dictionary to map from the first character of the postal code to the province or territory name. Here are the sample input and output of the program:

```
Enter a Canadian postal code: X0D 0A0
The postal code is for a rural address in Nunavut or Northwest
Territories.
```

```
Enter a Canadian postal code: H8Z 7R6
The postal code is for an urban address in Quebec.
Enter a Canadian postal code: ABB 2L5
The second character in the postal code must be a digit!
Enter a Canadian postal code: Q5N 8P4
The first character in the postal code is invalid!
It cannot be D, F, I, O, Q, U, W, or Z!
```

ALGORITHM:

PROGRAM:

OUTPUT:

Exercise 2: Converting Floating-Point Values to Monetary Amounts

AIM:

Write a Python class `MoneyFmt` for converting floating-point values to monetary amounts. The class has one data attribute of type `float` for the floating-point value called `value`. It also provides the following methods:

(a) `__init__(self, value=0.)` for creating a new instance of the class with the floating-point value initialized,

(b) `update(self, value=None)` for updating the floating-point value,

(c) `__str__(self)` for displaying the floating-point value as a monetary amount to two decimal places with the correct sign,

(d) `__repr__(self)` for returning a string such that `eval` applied to the string recreates the instance,

(e) `isnonzero(self)` for checking if the floating-point value is non-zero where a value with absolute value less than 0.005 is regarded as zero.

Here are the sample input and output for using the class `MoneyFmt`:

```
>>> cash = MoneyFmt(135.79)
>>> cash
MoneyFmt(135.79)
>>> print(cash)
$135.79
>>> cash.update(5000000.2468)
>>> cash
MoneyFmt(5000000.2468)
>>> print(cash)
$5,000,000.25
>>> cash.update(-1386.42)
>>> print(cash)
-$1,386.42
>>> cash.isnonzero()
True
>>> cash.update(-0.0049)
>>> cash.isnonzero()
```

```
False
```

ALGORITHM:

PROGRAM:

OUTPUT:

Exercise 3: Queue

AIM:

A queue behaves like a real-world queue such as a checkout line in a supermarket. The customer at the front of the line is serviced first and other customers can only enter at the back of the line. So the first customer in line is always the first one to be serviced (i. e. it follows the First In First Out principle). Write a Python class `Queue` for representing queues of integers. The class has two data attributes `qlst` and `maxsize` where `qlst` is the list holding the elements of the queue and `maxsize` is the size limit for the queue. It also provides the following methods:

(a) `__init__(self, maxsize=10)` for initializing the queue with an empty list and a size limit,

(b) `enqueue(self, item)` for adding an element to the back of a non-full queue,

(c) `dequeue(self)` for removing an element from the front of a non-empty queue,

(Hint: use the `pop` function)

(d) `isempty(self)` for checking whether the queue is empty,

(e) `isfull(self)` for checking whether the queue is full (i.e. whether the number of elements in the queue is equal to `maxsize`),

(f) `count(self)` for returning the number of elements in the queue,

(g) `__str__(self)` for displaying all the elements in the queue.

Note that in the method `enqueue`, the provided element would be added to the queue only if it is an integer. Here are the sample input and output for using the class `Queue`:

```
>>> a = Queue(3)
>>> a.isempty()
True
>>> a.enqueue(1.2)
You can only add integers to the queue!
>>> a.enqueue(1)
>>> a.enqueue(2)
>>> a.enqueue(3)
>>> a.enqueue(4)
The queue is full and no new element can be added!
>>> a.isfull()
True
```

```
>>> a.count()
Number of elements in the queue = 3
>>> a.dequeue()
1
>>> print(a)
Element 1 in the queue = 2
Element 2 in the queue = 3
>>> a.dequeue()
2
>>> a.dequeue()
3
>>> a.dequeue()
The queue is empty and there is no front element!
```

ALGORITHM:

PROGRAM:

OUTPUT:

Exercise 4: Quadrilaterals, Trapezoids, Parallelograms, and Squares

AIM:

Suppose a Python class `Point` is defined in the program `point.py` as follows:

```
class Point:
    """ A class representing geometric points """

    def __init__(self, x, y):
        """ Initialize the point with its x, y coordinates """
        self.x = x
        self.y = y

    def getx(self):
        """ Return the x coordinate of the point """
        return self.x

    def gety(self):
        """ Return the y coordinate of the point """
        return self.y

    def __str__(self):
        """ Return a string representation of the point """
        return "({:.1f}, {:.1f})".format(self.x, self.y)
```

Write a Python class `Quadrilateral` for representing quadrilaterals. This class has four data attributes `pt1`, `pt2`, `pt3`, `pt4` of class `Points` for the four endpoints of the quadrilateral. In addition to the constructor for initializing the instance, it provides the methods `getpt1(self)`, `getpt2(self)`, `getpt3(self)`, `getpt4(self)` for returning the individual endpoints of the quadrilateral, `getcoorstr(self)` for returning a string containing the coordinates of the endpoints of the quadrilateral, and `__str__(self)` for returning a string representation of the quadrilateral with the coordinates of its endpoints obtained by calling the method `getcoorstr`. Next, write a Python class `Trapezoid` for representing trapezoids by inheriting from the class `Quadrilateral`. It overrides the method `__str__(self)` for returning a string representation of the trapezoid with the coordinates of its endpoints, its height, and its area. It also adds the methods `getheight(self)` for returning the height of the trapezoid, `getsumoftwosides(self)` for returning the sum of the length of the two parallel sides of the trapezoid, and `getarea(self)` for returning the area of the trapezoid, assuming the points on each parallel side to have the same *y* coordinates. Next, write a Python class `Parallelogram` for representing parallelograms by inheriting from the class `Trapezoid`. It overrides the method `__str__(self)` for returning a string representation of the parallelogram with the coordinates of its endpoints, its width, its height, and its area. It also adds the methods `getwidth(self)` for returning the width of the parallelogram. Finally, write a Python class `Square` for representing squares by inheriting from the class `Parallelogram`. It overrides the method `__str__(self)` for returning a string representation of the square with the coordinates of its endpoints, its side, and its area. Assume the constructors of these classes are called with valid input. Here are the sample input and output for using these classes:

```
>>> a = Quadrilateral(1.1, 1.2, 6.6, 2.8, 6.2, 9.9, 2.2, 7.4)
>>> print(a)
Coordinates of the Quadrilateral are:
(1.1, 1.2), (6.6, 2.8), (6.2, 9.9), (2.2, 7.4)
>>> b = Trapezoid(0.0, 0.0, 10.0, 0.0, 8.0, 5.0, 3.3, 5.0)
>>> b.getsumoftwosides()
14.7
>>> print(b)
Coordinates of the Trapezoid are:
(0.0, 0.0), (10.0, 0.0), (8.0, 5.0), (3.3, 5.0)
Height = 5.0
Area = 36.75
>>> c = Parallelogram(5.0, 5.0, 11.0, 5.0, 12.0, 20.0, 6.0, 20.0)
>>> print(c)
Coordinates of the Parallelogram are:
(5.0, 5.0), (11.0, 5.0), (12.0, 20.0), (6.0, 20.0)
Width = 6.0
Height = 15.0
Area = 90.00
>>> d = Square(4.0, 0.0, 8.0, 0.0, 8.0, 4.0, 4.0, 4.0)
>>> print(d)
Coordinates of the Square are:
(4.0, 0.0), (8.0, 0.0), (8.0, 4.0), (4.0, 4.0)
Side = 4.0
Area = 16.00
```

ALGORITHM:

PROGRAM:

OUTPUT: