EECE 230X – Introduction to Computation and Programming
Programming Assignment 11

- This programming assignment consists of 4 problems and one optional problem
- Prerequisites: Topics 13 and 14:
  - Problems 1,2,3: Topic 13
  - Problems 4,5: Topic 14
- Related material: classes and object oriented programming, Stacks and Queues

Problem 1. Complex number class

A complex number is represented by two real numbers x and y as $x + iy$, where $i = \sqrt{-1}$. Design the class `ComplexNumber` which defines complex numbers as an Abstract Data Type (ADT).

Include the data attributes x and y and the method attributes:

- `__init__(self,`$x = 0$`,`$y = 0$`)`, which takes x and y as input arguments with zero default values. This method should return the exception “Bad Input!” if the type of x or y is not `int` or `float`.
- `conj(self)`, which returns the conjugate of `self` (the conjugate of $x + iy$ is $x - iy$)
- `norm(self)`, which returns the norm of `self` (the norm of $x + iy$ is $\sqrt{x^2 + y^2}$)
- Binary operators: `__add__`, `__sub__`, `__mul__`, `__truediv__`
- Unary operator: `__neg__(self)`, which returns $-$`self`
- Special method `__str__`, which represents the complex number as a string, as shown in the below test program

Test program:

```
r = ComplexNumber(12)
z = ComplexNumber(1.5,2)
t = ComplexNumber(2.2,3)
print(r)
print(z+t)
print(-z)
print(z*t)
print(z/t)
print(z.conj())
print(z.norm())
```

Output:

```
(12+0j)
(3.7+5j)
(-1.5-2j)
(-2.6999999999999997+8.9j)
(0.6719653179190752-0.007225433526011535j)
(1.5-2j)
2.5
```

Note: Python has a built in type for complex numbers called `complex`.

Problem 2. Point, circle, and rectangles classes

a) **Point and circle classes.** In this part, you will design abstract data types for planar points and circles.

A planar point p is represented by two real numbers x and y: x-coordinate $p.x$ and y-coordinate $p.y$. Design the class `Point` which defines a planar point as an Abstract Data Type (ADT). Include the data attributes x and y and the method attributes:

- `__init__`, which takes x and y as input arguments with zero default values. This method should return the exception "Bad Input!" if the type of x or y is not `int` or `float`.
- Special method `__str__`, which represents the point as a string of the form `"(x,y)"`

A circle C is represented by a `Point center` and a nonnegative number `radius`. Using the class `Point`, design the class `Circle` which defines a circle as an Abstract Data Type (ADT). Include the data attributes `center` and `radius` and the method attributes:

- `__init__`, which takes `center` and `radius` as input arguments, with default values `center=Point(0,0)` and `radius=1`. This method should return the exception "Bad Input!" `center` is not of type `Point`, `radius` is not of type `int` or `float`, or `radius` is not nonnegative.
- `__str__`, which represents the point as a string of the form `"((center.x,center.y),radius)"`, as shown in the below test program.
- `diameter`, which returns the diameter (i.e., $2 \times$ `radius`)
- `perimeter`, which returns the perimeter (i.e., $2\pi \times$ `radius`)
- `area`, which returns the area (i.e., $\pi \times$ `radius`^{2})
- `contains`, which checks if a given point `pt` is inside the circle. This method should return the exception "Bad Input!" if the type of `pt` is not `Point`
- `intersect`, which checks if the *disk associated with the circle* (i.e., the circle and its interior) has a nonempty intersection with the disk associated with another given circle `other`. This method should return the exception "Bad Input!" if the type of `other` is not `Circle`.

Test program:	*Output*:
`C1 = Circle()`	
`C2 = Circle(Point(1,0.5),0.75)`	
`C3 = Circle(Point(10,5),2)`	
`print(C1)`	`((0,0),1)`
`print(C2)`	`((1,0.5),0.75)`
`print(C3)`	`((10,5),2)`
`print(C1.diameter())`	`2`
`print(C2.perimeter())`	`4.71238898038469`
`print(C3.area())`	`12.566370614359172`
`print(C1.contains(Point(0.5,0.5)))`	`True`
`print(C1.contains(Point(5,5)))`	`False`
`print(C1.intersect(C2))`	`True`
`print(C1.intersect(C3))`	`False`

b) **Rectangle class.** In this part, you will design an abstract data type for rectangles based on the planar `Point` class in Part (a).

A rectangle R is represented by two points p and q, where p is the lower left corner R and q is the upper right corner of R. Using the class `Point`, design the class `Rectangle` which defines a rectangle as an Abstract Data Type (ADT). Include the data attributes p and q and the method attributes:

- `__init__`, which takes p and q as input arguments. This method should return the exception "Bad Input!" if the following condition does not hold: type of p and q is `Point`, $p.x \leq q.x$, and $p.y \leq q.y$.
- Special method `__str__`, which represents the rectangle as a string of the form `"(p=(a,b),q=(c,d))"`.
- `height`, which returns the height
- `width`, which returns the width
- `perimeter`, which returns the perimeter
- `area`, which returns the area
- `contains`, which checks if a given point `pt` is inside the rectangle. This method should return the exception "Bad Input!" if the type of `pt` is not `Point`

Test program:	*Output:*
`Include a statement to initialize a rectangle R whose lower left corner is (1,2) and upper right corner is (3.2,4)`	
`print(R)`	`(p=(1,2),q=(3.2,4))`
`print(R.width())`	`2.2`
`print(R.height())`	`2`
`print(R.area())`	`4.4`
`print(R.perimeter())`	`8.4`
`print(R.contains(Point(1.5,3)))`	`True`
`print(R.contains(Point(10.5,3)))`	`False`

Problem 3. Vector class inherited from list

In this problem, by a vector v we mean an n-dimensional vector v represented as a list of n real coordinates $[x_0, x_1, \ldots, x_{n-1}]$.

Design the class `Vector` as a subclass of `list`. As a subclass of `list`, `Vector` doesn't have new data attributes.

Override the following special methods of `list`:

- `__init__(self,other)`, which invokes `list.__init__` to initialize `self` to `other` after asserting that `len(other)` is nonzero and the types of all entries of `other` are `int` or `float`. Below is the code of `__init__(self,other)`:

```
def __init__(self,other):
    assert len(other)!=0, "Invalid Input!"
    for e in other:
        assert type(e)==int or type(e)==float,  "Invalid Input!"
    list.__init__(self,other)
```

- `__str__` to represents the vector as a string of the form "$\langle x_0, x_1, \ldots, x_{n-1}\rangle$"
- Overloaded binary operator `__add__(self,other)` to perform the pointwise addition of two vectors:

$$\langle x_0, x_1, \ldots, x_{n-1}\rangle + \langle y_0, y_1, \ldots, y_{n-1}\rangle = \langle x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}\rangle.$$

  If `self` and `other` are not of the same length (i.e., dimension), this function should raise the exception "Invalid Input!".

Include the following new methods:

- Binary operator `__sub__` to perform pointwise subtraction. As in vector addition, raise the exception "Invalid Input!" in case of dimensional mismatch.
- Binary operator `__mul__` to perform the scalar product:

$$\langle x_0, x_1, \ldots, x_{n-1}\rangle \cdot \langle y_0, y_1, \ldots, y_{n-1}\rangle = x_0y_0 + x_1y_1 + \ldots + x_{n-1}y_{n-1}.$$

  As in vector addition, raise the exception "Invalid Input!" in case of dimensional mismatch. Note that unlike `__add__` and `__sub__`, `__mul__` returns a real number (i.e., not a vector).

- Unary operator `__neg(self)__`, which returns $-$`self`:

$$-\langle x_0, x_1, \ldots, x_{n-1}\rangle = \langle -x_0, -x_1, \ldots, -x_{n-1}\rangle$$

- `norm(self)`, which returns the euclidean norm |`self`| of `self`:

$$|\langle x_0, x_1, \ldots, x_{n-1}\rangle| = \sqrt{x_0^2 + x_1^2 + \ldots + x_{n-1}^2}$$

Include also the following functions (not methods):

- `zeros(n)`, which given an integer `n`, returns an all zeros vector of dimension n.
- `ones(n)`, which given an integer `n`, returns an all ones vector of dimension n.

Test program:	*Output:*
`v = Vector([1,2.3])`	
`w = zeros(5)`	
`u = ones(2)`	
`print(v)`	`<1,2.3>`
`print(w)`	`<0,0,0,0,0>`
`print(v[1])`	`2.3`
`print(v+v+v)`	`<3,6.8999999999999995>`
`print(v*u)`	`3.3`
`print(v.norm())`	`2.5079872407968904`
`w[2]=3.5`	
`v = copy.deepcopy(w)`	
`print(v)`	`<0,0,3.5,0,0>`
`print(-v)`	`<0,0,-3.5,0,0>`
`w[0]=15.5`	
`w[4]=-12`	
`print(w-v)`	`<15.5,0,0.0,0,-12>`
`print(-w+v)`	`<-15.5,0,0.0,0,12>`

Problem 4. Unbounded circular queue inherited from bounded circular queue

We implemented in Topic 14 the circular queue class `Queue`. One drawback of this implementation is that we have to set `maxSize` at initialization and live with it. One solution is to double the list size if maxSize reached instead of raising the exception "Queue Full".

Derive the class `UnboundedQueue` from the class `Queue` as follows:

- All the method of `Queue` except `Queue.enqueue` are inherited by `UnboundedQueue`
- Add new "pseudo-private" method `__doubleList(self)`, which when invoked doubles the list.

  (*Hint:* First, create a new list `newL` of size `2*self.maxSize`. Then copy the used content of `self.L` starting from index `self.head` into `newL` starting from index 0. Then update `self.head`, `self.tail`, `self.L`, and `self.maxSize`.)
- Override the method `enqueue(self,val)` by first calling `__doubleList(self)` if `self.isFull()` and then invoking `Queue.enqueue(self,val)`.

The time of dequeue remains $O(1)$. The time of enqueue is not anymore $O(1)$ in the worst case, but it is $O(1)$ in the amortized sense: the cost of a sequence of n `UnboundedQueue.enqueue` operation is $O(n)$. The analysis is similar to the `list.append` amortized analysis in Topic 8.

Test program:	*Output:*
`Q = UnboundedQueue(5)`	
`for i in range(5):`	
`    Q.enqueue(i)`	
`print(Q)`	`[0,1,2,3,4]`
`Q.dequeue()`	
`print(Q)`	`[1,2,3,4]`
`Q.dequeue()`	
`print(Q)`	`[2,3,4]`
`for i in range(5):`	
`    Q.enqueue(i)`	
`print(Q)`	`[2,3,4,0,1,2,3,4]`
`for i in range(10):`	
`    Q.enqueue(i)`	
`print(Q)`	`[2,3,4,0,1,2,3,4,0,1,2,3,4,5,6,7,8,9]`

Problem 5 ($\star$) (Optional). Iterative Randomized Quick Sort using a stack

Since recursion is implemented using a stack, any recursive function has an equivalent iterative implementation using a stack.

Use a stack to implement an iterative version of Randomized Quick Sort. Minimize the stack use.