The CS 5 Black Gazette

CS 5 Students Object to Classes. “They serve no function and we disagree with the methods,” say students.

(Claremont, AP): Students in CS 5 say that they object to classes. “We’re overloaded!” said one student, “and we want to underscore underscore our concerns.” Another student spokesperson said “The professors are def __init__ely hoping this is something that will just float away, but they can’t string us along forever. We have a long list of issues and if the profs don’t understand them, they should look them up in a dictionary,” said a student. “We sure wish the students were mutable!” said one professor. Students and professors eventually agreed on a tuple of ways to __repr__ their relationship.
Classes and operator overloading

class Rational:
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def __add__(self, other):
        newNumerator = self.numerator * other.denominator + self.denominator * other.numerator
        newDenominator = self.denominator * other.denominator
        return Rational(newNumerator, newDenominator)

    def __eq__(self, other):
        return self.numerator*other.denominator == self.denominator*other.numerator

    def __ge__(self, other):
        return self.numerator*other.denominator >= self.denominator*other.numerator

    def __repr__(self):
        return str(self.numerator) + "/" + str(self.denominator)

fuelNeeded = Rational(42, 1000)
tank1 = Rational(36, 1000)
tank2 = Rational(6, 1000)
total = tank1 + tank2  # tank1.__add__(tank2)
if total >= fuelNeeded:  # total.__ge__(fuelNeeded)
Overloaded Operator Naming

+ __add__ + __pos__ == __eq__
- __sub__ - __neg__ != __ne__
* __mul__ __abs__ <= __le__
/ __div__ __int__ >= __ge__
// __floordiv__ __float__ < __lt__
% __mod__ __complex__ > __gt__
** __pow__
OOPs! (Object-Oriented Programs)

```python
>>> assigned = Date(11, 13, 2012)
>>> due = Date(11, 20, 2012)
>>> due - assigned  # due.__sub__(assigned)
7

>>> if due > assigned:  # due.__gt__(assigned)
...     print "Go watch a movie!"
```

class Date:
    def __init__(self, m, d, y):
        self.month = m
        self.day = d
        self.year = y

>>> d = Date(1, 21, 1969)
Another implementation...

class Date:
    def __init__(self, m, d, y):
        self.daysSince1900 = ...

>>> d = Date(1, 21, 1969)

Why would any sane person want ant to store the date as the number of days since January 1, 1900?
Getters and Setters

class Date:
    def __init__(self, m, d, y):
        self.daysSince1900 = ...

    def setDay(self, d):
        if d <= 0 or d > 31:
            ...
        else:
            self.dayCount = ...

>>> d = Date(1, 21, 1969)
>>> d.setDay(28)    # SETTER
>>> x = d.getDay()  # GETTER
Date “Abstraction”

Date
	__init__(self, month, day, year)
setDay(self, day)
setMonth(self, month)
setYear(self, year)
getDay(self)
getMonth(self)
getYear(self)
==, >, <, >=, <=, +, -
The Advantage of Abstraction

Vector, Victor!

```python
class Vector:
    def __init__(self, x, y):
        # Initialize the vector with components x and y.
    def __repr__(self):
        # Return a string representation of the vector.
    def __add__(self, other):
        # Add the other vector to this vector.
    def __sub__(self, other):
        # Subtract another vector from this vector.
    def magnitude(self):
        # Return the magnitude of the vector.
    def normalize(self):
        # Normalize the vector.

In a file called Vector.py

>>> victor = Vector(1, 1)
>>> victor
(1, 1)
>>> roger = Vector(0, 2)
>>> roger
(0, 2)
>>> A = victor + roger
>>> A
(1, 3)
>>> victor.magnitude()
1.4142135
>>> -A ← A.__neg__()  # Subtract another vector from this vector.
(-1, -3)
```
import math

class Vector(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def magnitude(self):
        """ Returns magnitude of vector """
        return math.sqrt(self.x * self.x + self.y * self.y)

    def normalize(self):
        """ Sets vector magnitude to 1 """
        mag = self.magnitude()
        self.x = self.x / mag
        self.y = self.y / mag

    def __add__(self, other):
        return

    def __neg__(self):
        return

    def __sub__(self, other):
        return

    def __repr__(self):
        return
import math

class Vector(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def magnitude(self):
        """ Returns magnitude of vector """
        return math.sqrt(self.x * self.x + self.y * self.y)

    def normalize(self):
        """ Sets vector magnitude to 1 """
        mag = self.magnitude()
        self.x = self.x / mag
        self.y = self.y / mag

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __neg__(self):
        return Vector(-self.x, -self.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __repr__(self):
        return "{" + str(self.x) + "," + str(self.y) + "}"
Inheritance (inheritance)

class Person:
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return 23 <= time <= 7  # Using 24 hour time!

    def __repr__(self):
        return self.firstName + " " + self.lastName

>>> ran = Person("Ran", "Libeskind-Hadas")
>>> ran
Ran Libeskind-Hadas
>>> ran.asleep(2)
True
class Person:
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return 23 <= time <= 7  # Using 24 hour time!

    def __repr__(self):
        return self.firstName + " " + self.lastName

class Student(Person):
    def __init__(self, first, last, major):
        Person.__init__(self, first, last)
        self.major = major

    def asleep(self, time):
        return 3 <= time <= 11

    def __repr__(self):
        return Person.__repr__(self) + ", " + self.major + " major"

>>> s = Student("Anna", "Litik", "CS")
>>> s
Anna Litik, CS major
>>> s.asleep(2)
False
class Person:
    def __init__(self, first, last):
        self.firstName = first
        self.lastName = last

    def asleep(self, time):
        return 0 <= time <= 7

    def __repr__(self):
        return self.firstName + " " + self.lastName

class Student(Person):
    def __init__(self, first, last, major):
        Person.__init__(self, first, last)
        self.major = major

    def asleep(self, time):
        return 3 <= time <= 11

    def __repr__(self):
        return Person.__repr__(self) + ", " + self.major + " major"

class Mudder(Student):
    def __init__(self, first, last, major, dorm):
        Student.__init__(self, first, last, major)
        self.dormitory = dorm

    def asleep(self, time):
        return False

>>> W = Mudder("Wally", "Wart", "Dermatology", "North")
>>> W
>>> W.asleep()
Objects are self-ish

def rename(self, newF, newL):
    self.firstName = newName
    self.lastName = newL

def rename(self, newName):
    self = Person(newF, newL)
In Python, everything is a class!

```python
class int:
    def __init__(self, str):
```
In Python, everything is a class!

class list:
    def __init__(self):
    def __repr__(self):
    def __append__(self, item):
    def __add__(self, other):
    def __getitem__(self, index):
    def __setitem__(self, index, value):

>>> x = []
>>> x.append(42)
>>> x
[42]
>>> x[0]
42
>>> x[0] = 67

>>> x = list()
>>> x.append(42)
>>> x.__repr__()  
[42]
>>> x.__getitem__(0)
42
>>> x.__setitem__(0, 67)
Polite lists

```python
>>> p = polite()
Thank you for using polite lists!
>>> p.append(42)
>>> p.append(50)
>>> p[0] += 1
>>> p
<Polite list: 43 50 >
>>> p[3] = 100
Pardon me, but your index is out of bounds
>>> p[3]
Pardon me, but your index is out of bounds
```
class polite(list):

def __init__(self):
    print("Thank you for using polite lists!")

def __repr__(self):
    output = "<Polite list: "
    return output

def __setitem__(self, index, value):
    if index >= len(self):
        print("Pardon me, but your index is out of bounds")
    else:
        list.__setitem__(self, index, value)

def __getitem__(self, index):
    if index >= len(self):
        print("Pardon me, but your index is out of bounds")
    else:
        return list.__getitem__(self, index)
class polite(list):
    
def __init__(self):
        print("Thank you for using polite lists!")
    
def __repr__(self):
        output = "<Polite list: 
        for x in self:
            output = output + str(x) + " "
        output += ">
        return output
    
def __setitem__(self, index, value):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            list.__setitem__(self, index, value)
    
def __getitem__(self, index):
        if index >= len(self):
            print("Pardon me, but your index is out of bounds")
        else:
            return list.__getitem__(self, index)
>>> p = polite()
Thank you for using polite lists!

>>> for x in range(0, 10): p.append(x)

>>> q = polite()
Thank you for using polite lists!

>>> for y in range(5, 100): q.append(y)

>>> r = p - q
← r = p.__sub__(q)
Thank you for using polite lists!

>>> r
<Polite list: 0 1 2 3 4 >

Worksheet!
>>> p = polite()
Thank you for using polite lists!

>>> for x in range(0, 10): p.append(x)

>>> q = polite()
Thank you for using polite lists!

>>> for y in range(5, 100): q.append(y)

>>> r = p - q
leness r = p.__sub__(q)
Thank you for using polite lists!

>>> r
<Polite list: 0 1 2 3 4>

def __sub__(self, other):
    output = polite()
    for x in self:
        if not x in other:
            output.append(x)
    return output
class Person:
    def __init__(self, ...):
        blah, blah, blah

class Infantryman(Person):
    def __init__(self, ...):
        blah, blah, blah
class Person:
    def __init__(self,…):
        blah, blah, blah

class Infantryman(Person):
    def __init__(self, …):
        blah, blah, blah

class Kangaroo(Infantryman):
    def __init__(self, …):

class Person:
    def __init__(self, firstName, lastName):
    def __repr__(self):
    def asleep(self):

class Student(Person):
    def __init__(self, firstName, lastName, school="Claremont Colleges", year=None, major="undeclared"):
        self.Person.__init__(firstName, lastName)
        self.school = school
        self.year = year
        self.major = major

>>> Alice = Student("Alice", "Chi", "HMC", "firstyear")
>>> Bigbird = Student("Big", "Bird", year="senior", major="birds")
>>> Elmo = Student("Elmo")
Millisoft “Shapes”
(HW 10!)

>>> r = Rectangle(100, 50, center=Vector(200, 0), color="blue")
>>> c = Circle(radius=30, color = "red")  # default center (0,0)
>>> r.rotate(45)  # 45 degree ccw rotation about origin
>>> r.render()  # draw it!
>>> c.render()  # draw it!

Radius = 30
(200,0) 50
100
You’ll write the Shapes class!

```python
from Shapes import *

def demo():
    myRectangle = Rectangle(130, 50, center = Vector(0, 50), color = "red")
    eye0 = Circle(Vector(-80, 100), 15, color = "blue")
    eye1 = Circle(Vector(-40, 100), 15, color = "blue")
    eye2 = Circle(Vector(0, 100), 15, color = "blue")
    eye3 = Circle(Vector(40, 100), 15, color = "blue")
    eye4 = Circle(Vector(80, 100), 15, color = "blue")
    L = [myRectangle, eye0, eye1, eye2, eye3, eye4]
    for part in L:
        part.render()
```
class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!

    def render(self):
        code here...

    def erase(self):
        code here...

    def rotate(self, theta):
        code here...

    def scale(self, stretch):
        code here...

    def translate(self, shiftVector):
        code here...

class Circle(Shape):

class Rectangle(Shape):

class Square(Rectangle):

Inheriting from Shape means we don’t have to reinvent the wheel!

I don’t plan to ever call the Shape factory to make a Shape. But I will call the Rectangle, Square, and Circle factories!

An “abstract” class
class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!

    def render(self): code here...
    def erase(self): code here...
    def rotate(self, theta): code here...
    def scale(self, stretch): code here...
    def translate(self, shiftVector): code here...

class Rectangle(Shape):
    def __init__(self, width, height, center=Vector(0, 0), color="black"):
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width, center=Vector(0, 0), color="black"):
        Rectangle.__init__(self, width, width, center, color)
import math
import turtle
from Matrix import *
from Vector import *

class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!
    
def render(self):
    def erase(self):
    def rotate(self, theta):
    def translate(self, shiftVector):
    def scale(self, stretch):

class Rectangle(Shape):

class Square(Rectangle):

class Circle(Shape):

Specific shapes inherit from Shape!
First a few words about matrices...

Matrix-vector multiplication...

\[
\begin{bmatrix}
  a & b \\
  c & d
\end{bmatrix} \cdot \begin{bmatrix}
  x \\
  y
\end{bmatrix} = \begin{bmatrix}
  ax + by \\
  cx + dy
\end{bmatrix}
\]

Matrix-Matrix multiplication...

\[
\begin{bmatrix}
  a_1 & b_1 \\
  c_1 & d_1
\end{bmatrix} \times \begin{bmatrix}
  a_2 & b_2 \\
  c_2 & d_2
\end{bmatrix} = \begin{bmatrix}
  a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\
  c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2
\end{bmatrix}
\]
class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!

def render(self): code here...
def erase(self): code here...
def rotate(self, theta): code here...
def scale(self, stretch): code here...
def translate(self, shiftVector): code here...

class Circle(Shape):

class Rectangle(Shape):

class Square(Rectangle):

$$R = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}$$

Rotate about origin for now. We’ll extend this to rotating about an arbitrary point shortly...
class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!

    def render(self): code here...
    def erase(self): code here...
    def rotate(self, theta): code here...
    def scale(self, stretch):
    def translate(self, shiftVector): code here...

class Circle(Shape):

class Rectangle(Shape):

class Square(Rectangle):

Assume object centered at origin for now. We’ll generalize this shortly.
class Shape:
    def __init__(self):
        self.points = []  # List of Vectors!

    def render(self):
        code here...
    def erase(self):
        code here...
    def rotate(self, theta):
        code here...
    def scale(self, stretch):
        code here...
    def translate(self, shiftVector):
        code here...

class Circle(Shape):

class Rectangle(Shape):

class Square(Rectangle):
A Matrix Class

>>> m = Matrix(1, 2, 3, 4)
>>> m
1 2
3 4
>>> m.get(0, 1)  # Why do things this way?
2
>>> v = Vector(1, 20)
>>> v
(1, 20)
>>> v[0]
???
>>> w = m * v  # w = m.__mul__(v)
>>> w
???

I thought that Linear Algebra was the Matrix Class!

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def magnitude(self):
        blah, blah, blah
        return ...

    def __repr__(self):
        return "\((", + str(self.x) + ",\)"
        + str(self.y) + ")"

    def __add__(self, other):
        blah, blah, blah
        return Vector(blah, blah)

    def __sub__(self, other):
        blah, blah, blah
        return Vector(blah, blah)
Matrix Class

from Vector import *

def Matrix:
    """2x2 matrix class"""

def __init__(self, a11=0, a12=0, a21=0, a22=0)
    self.array = [[a11, a12], [a21, a22]]

def __repr__(self)

def set(self, row, column, value):

def get(self, row, column):

def __mul__(self, other):
    """other may be a matrix OR a vector and returns the product of self and other.""

    Blah, blah, blah
    if other.__class__.__name__ == "Matrix":
        blah, blah, blah
    else:  # it’s HOPEFULLY a Vector!

>>> m = Matrix(1, 2, 3, 4)
>>> n = Matrix(1, 1, 2, 2)
>>> v = Vector(1, 3)
>>> p = m * n  # m.__mul__(n)
>>> q = m * v  # m.__mul__(v)

I was trying to anticipate how this would be done!

Notice that we store the matrix in a “funny” way. Why?!
import math  # has math.cos(angle) and math.sin(angle)

class Shape:
    def __init__(self):
        self.points = []

    def render(self):

    def translate(self, shiftVector):

    def rotate(self, theta):
        """ Rotate shape by theta degrees """
        theta = math.radians(theta)  # Python thinks in radians!

class Rectangle(Shape):
    def __init__(self, width, height, center=Vector(0, 0), color="black"):
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width, center=Vector(0, 0), color="black"):
        Rectangle.__init__(self, width, width, center, color)
class Shape:
    def __init__(self):
        self.points = []

    def render(self):
    def translate(self, shiftVector):

    def rotate(self, theta):
        """Rotate shape by theta degrees """
        theta = math.radians(theta)  # Python thinks in radians!
        RotationMatrix = Matrix(math.cos(theta), -1*math.sin(theta), math.sin(theta), math.cos(theta))
        NewPoints = []
        for vector in self.points:
            newvector = RotationMatrix * vector
            NewPoints.append(newvector)
        self.points = NewPoints

class Rectangle(Shape):
    def __init__(self, width, height, center = Vector(0, 0), color = "black"):  
        SW = Vector(center.x - width/2.0, center.y - height/2.0)
        NW = Vector(center.x - width/2.0, center.y + height/2.0)
        NE = Vector(center.x + width/2.0, center.y + height/2.0)
        SE = Vector(center.x + width/2.0, center.y - height/2.0)
        self.points = [SW, NW, NE, SE]
        self.color = color

class Square(Rectangle):
    def __init__(self, width, center = Vector(0, 0), color = "black"):  
        Rectangle.__init__(self, width, width, center, center, color)
class Shape:
    def __init__(self):
        self.points = []

    def render(self):
        
    def rotate(self, theta):

class Circle (Shape):
    def __init__(self, center=Vector(0,0), radius=10, color="black")
        self.center = center
        self.radius = radius
        self.color = color

The turtle draws circles using turtle.circle(radius)

Should we still use the Shape class render, rotate, translate, etc.?
Rotate, scale, flip

def rotate(self, theta, about=Vector(0, 0)):

def flip(self, vector1, vector2):
Draw a pretty picture...

... and deposit it as a .png or .jpg here...