CSci 1133, Spring 2015
Lab Exercise 11

Introduction to Objects

Object-oriented programming is a powerful computational paradigm in which we approach problem solving from the perspective of how *objects* (Abstract Data Types) interact rather than the step-by-step procedural approach we've been exploring so far. Our exploration begins with the construction of simple *representations* (classes) that Python uses as "templates" or "recipes" for instantiating objects.

Primer on Printing Objects
Let's create an object and explore it a bit. Using your IDE editor, enter the following class:

```
class rational:
    def __init__(self,num=0,den=1):
        self.numerator = num
        if den == 0:
            self.denominator = 1
        else:
            self.denominator = den
```

This defines a class of objects that will represent *rational numbers*, i.e., those that can be expressed as the ratio of two integers. Now, using the command line Python environment, let's instantiate two `rational` objects:

```
>>> num1 = rational(3,4) [ENTER]
>>> num2 = rational(1,3) [ENTER]
```

Let's see what they look like:

```
>>> num1 [ENTER]
```

Hmmm… that's not what we wanted. Python is telling us that the "value" of `num1` is a `main.rational` object. Perhaps if we print the value…

```
>>> print(num1) [ENTER]
```

...still not what we wanted. How do we actually display the value of the object??! Since this is a thing we've created ourselves, Python has no idea what we mean by "printing" a `rational` object. Like nearly everything in object-land, we have to tell Python explicitly what we mean. It turns out many functions in Python have anticipated that we might be creating weird objects of our own design, so *polymorphism* (we'll learn about it later) is employed to provide a "hook" into the future…

For now, you need to know that if some Python function needs a "value" to represent your object, it will obtain a string representation by calling the `__str__()` method for your class.

If we want functions such as `print` to display the "value" of a `rational` object, we must supply a method named `__str__()` in our class definition. Add the following method to the rational class definition:

```
def __str__(self):
    return str(self.numerator) + '/' + str(self.denominator)
```

Now let's try it again:

```
>>> num1 = rational(3,4) [ENTER]
>>> print(num1) [ENTER]
```

Voila! Anytime we use `print` to display a rational number, it will display the current value as a "fraction"

Warm-up

1). Modify the `__str__()` method for the `rational` class to display fractions such as 3/1 and 4/1 as "whole" numbers rather than fractions. Also, if the numerator is zero, display 0 rather than 0/*n*:

Stretch

1). Construct a class named `measure` that will keep track of measured distances in *feet* and *inches*. Include a constructor that will initialize two instance variables to hold integer values for the feet and inches respectively. Your constructor should allow the following initialization forms:

`measure()`	*zero feet, zero inches*
`measure(4,5)`	*four feet, five inches*
`measure(48)`	*forty-eight inches* (or *four feet*)

Stored measurements should be *normalized* such that the number of inches is always < 12. If the number of inches is greater than 12, the number of feet should be adjusted to maintain this property.

2). Provide an overloaded `str()` operator in the `measure` class that will return values as a string in the following format:

feet ' *inches* "

where *feet* is the integer value of the number of feet followed by a single apostrophe, and *inches* is the integer value of the number of feet followed by two apostrophes. Do not show values of zero unless *both* values are zero, in which case show 0". For example, if the number of inches is zero, you should only show the value for feet. Similarly, if the number of feet is zero you should only show the value for inches.

3). Overload the addition (+) and subtraction (-) operators to add/subtract two `measure` objects respectively.

4). Test your class implementation by writing a function that includes the following Python statements:

```
m1 = measure()
m2 = measure(4,11)
m3 = measure(6,10)
print( m1 )
print( m2+m3 )
print( m3-m2 )
```

Workout

1). **Vector Class**

A *vector* is simply an ordered list of scalar values. Vectors are used in a broad spectrum of scientific fields. For example, points in space are generally represented using a 3-dimensional vector containing the [*x*, *y*, *z*] coordinates. The elements of a vector simultaneously describe both a *direction* and *magnitude* (length). So they are useful to represent quantitative data such as force, velocity, acceleration, etc.

Construct a class named `vec3` that will represent 3-dimensional vectors. The `vec3` class should maintain the scalar elements as a Python list. Include the following:

- A constructor to initialize a `vec3` object from a *list* of values (default to the vector [0,0,0])
- An overloaded `str()` operator that will format and return a string as follows:

  $[v_1, v_2, v_3]$
- An accessor method named `vlist` that will return the vector elements as a list.
- A mutator method named `setValues` that will take a *list* argument and update the vector elements
- An overloaded `float()` operator that will return a floating-point value representing the *magnitude* of the vector. The vector *magnitude* is computed as:

  $$\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$$
- An overloaded addition (+) operator that will return a `vec3` object representing the sum of two vectors. The sum of two vectors is the sum of their respective elements. For example, given two vectors: $[a_1, a_2, a_3, \ldots, a_n]$ and $[b_1, b_2, b_3, \ldots, b_n]$, the sum is:

  $[\ a_1+b_1, a_2+b_2, a_3+b_3, \ldots, a_n+b_n\ ]$
- An overloaded division (/) operator that will return the vector quotient of a vector divided by a scalar value. The quotient is a vector formed by dividing each element of the vector argument by the scalar value.

Write a short Python program that will input two 3-dimensional "force" vectors and a mass value, and then compute the resultant acceleration. Display the resulting acceleration as both a vector and its magnitude.

Recall from Newton's second law: force = mass * acceleration.

Challenge

Here's an interesting challenge problem. Try it if you have extra time or would like additional practice outside of lab.

1). **Polynomials Again**

In this problem, you will construct a class named `Poly` to represent polynomials. Recall that a polynomial of *degree n* is represented as the sum of n+1 terms as follows:

$$a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \ldots + a_nx + a_{n+1}$$

where a_i are the n+1 *coefficients* of the polynomial. *Degree*-2 polynomials are also known as *quadratic* polynomials. For example, the following expression represents a *quadratic* polynomial:

$$36.7x^2 - 23.2x - 4$$

The *variable* in a polynomial simply acts as a "placeholder" for the coefficients. Therefore the only truly useful information about polynomials is the array of coefficients and their corresponding exponents. A simple way to implement the polynomial class is to use a list of *floats* to store the coefficient values and simply use the ordinal (index) of the list element to represent the exponent of the corresponding term. For example, the polynomial $x^3 + x + 2$ would be represented as an array of coefficients like this:

```
coefficient = [2,1,0,1]
```

where the 1st element is the coefficient for the x^0 term, the second element is the coefficient for the x^1 term, the third element represents the coefficient for the x^2 term, and so on. Note that since the x^2 term is missing from the polynomial expansion above, we simply represent it using a zero coefficient.

Part 1: **Basic Polynomial Class**

Construct a class named `Poly` to represent and maintain polynomials using the array-of-coefficients method described above. Include a list instance variable to represent the polynomial Your class definition should include the following:

a. A constructor that will initialize the polynomial value using a list of coefficients. It should default to the scalar value 0.

b. An accessor method named `degree` that will return the degree of the polynomial (highest exponent corresponding to a non-zero coefficient).

c. A mutator method named `addTerm` that will take two values: an exponent value and a coefficient and add the term to the polynomial (inserting zero terms as needed).

d. __str__() method that will display a polynomial object. Polynomials should be displayed using the following symbolic format:

*a*x^3 + *b*x^2 + *c*x – *d*

where '^' character represents exponentiation and *a*, *b*, *c* and *d* represent coefficient values from the list. Pay careful attention to the output format and obey the following constraints:

- Do not display any terms with zero coefficients.
- Do not display coefficients equal to 1.
- Do not display a '+' sign for the highest order term (but do include a '-' sign if the coefficient is negative).
- Use minus '-' signs in place of '+' for negative terms.
- Do not display `x^0` for the last term (just the coefficient).

For example, the polynomial $36.7x^2 - 23.2x - 4$ would be displayed as:

```
36.7x^2 - 23.2x - 4
```

e. Write a mutator method named `integrate` that will symbolically integrate the calling object. For example, the integral of a quadratic polynomial $dy/dt = ax^2 + bx + c\ dx$ is $y = ax^3/3 + bx^2/2 + cx + C$ (where C is an indeterminate constant). For this function, assume the integration constant C is zero.

Again, be sure to thoroughly test your methods using a suitable test program.

Part2: **Even More FUN with Polynomials**

a. Modify the constructor for the Poly class to accept a string consisting of a polynomial representation in the same form that the `__str__()` method produced.

b. Add an accessor method that will evaluate the polynomial for any value of x.