MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007

Work for Week 3

Issued: Thursday, Feb. 22

This handout contains:

- Software Lab for Thursday, February 22
- Post-lab exercises due Tuesday, February 27

Thursday Software Lab: OOPractice

This week, we'll get practice with object-oriented programming, and build some tools that will be useful in future labs.

We'll start with building a class that illustrates the value of objects as abstract data types. Then we'll exercise encapsulation of state and inheritance. Finally, we'll put them together.

Vector arithmetic

Start by defining a class of two-dimensional vectors, called `V2`, that supports the following operations:

- Create a new vector out of two real numbers: `v = V2(1.1, 2.2)`
- Add two V2s to get a new V2; don't change either of the vectors being added. You can think of this as `V2.add(v, V2(1.0, 1.0))` or `v.add(V2(1.0, 1.0))`.
- Multiply a V2 by a scalar (real or int) and return a new V2. You can think of this as `V2.mult(v, 6.0)` or `v.mult(6)`.

Question 1. Define the basic parts of your class, with an `__init__` method and a `__str__` method (see the lecture notes for more information about this), so that if you do

```
print V2(1.1, 2.2)
```

something nice and informative is printed out.

Question 2. Implement the `add` and `mul` operators, and demonstrate them to your LA.

A cool thing about Python is that you can *overload* the arithmetic operators. So, for example, if you add the following method to your `V2` class

```
def __add__(self, v):
    return self.add(v)
```

or, if you prefer,

```
    def __add__(self, v):
        return V2.add(self, v)
```

then you can do

```
>>> print V2(1.1, 2.2) + V2(3.3, 4.4)
V2(4.4,6.6)
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example.

Question 3. Add the `__add__` method, as well as methods named `__mul__` and `__div__` to your class. The `__mul__` method should call your `mul` method to multiply the vector by a scalar. The `__div__` method should divide a vector by a scalar. Debug them on some test cases, and then demonstrate them to your LA.

Question 4. Also, add two accessor methods, `getX` and `getY` that return the x and y components of your vector, respectively. For example,

```
>>> v = V2(1, 2)
>>> v.getX()
1
>>> v.getY()
2
```

Question 5. Submit this code as your answer to the first tutor question for problem set 3.

Question 6. Be sure to save this code in a file called `V2.py` and mail or `scp` it to yourself, because you'll need it later in the lab.

Rocket science

In this problem, we will explore object-oriented programming in the context of a very simple simulated physics world. Our world will be populated with instances of class `Body`. Each of these instances has attributes `mass`, `position`, and velocity. For now, think of position and velocity as being one-dimensional quantities, but we'll actually change that later.

Instances of class `Body` support three methods

- `__init__` sets the mass, initial position and initial velocity of the body.
- `updatePosition` simulates a forward step in the dynamics of this world for an amount of time `dt`. As a result, the position of the object is changed, by moving it according to its current velocity.
- In order to get anything to happen, we have to be able to apply an acceleration. The `applyAcceleration` method takes as an argument an `acceleration` and an amount of time and increases the velocity appropriately.
- We also add an `applyForce` method, which just divides the force by the mass of the object and applies that as an acceleration, for a specified increment of time.

Here is the code:

```
class Body:
    def __init__(self, mass, initPosition, initVelocity):
        self.mass = float(mass)
        self.position = initPosition
        self.velocity = initVelocity

    def updatePosition(self, dt):
        self.position = self.position + self.velocity * dt

    def applyAcceleration(self, a, dt):
        self.velocity = self.velocity + a * dt

    def applyForce(self, force, dt):
        self.applyAcceleration(force/self.mass, dt)
```

Once we have the basic `Body` class, we can take advantage of inheritance to make some special cases. We'll start by adding gravity, as a constant acceleration; a body with gravity has the gravitational acceleration applied to it on every step, just before it updates its position. Here is the code for the `BodyWithGravity` class.

```
class BodyWithGravity(Body):
    def __init__(self, mass, initPosition, initVelocity, gravityVector):
        self.gravityVector = gravityVector
        Body.__init__(self, mass, initPosition, initVelocity)

    def updatePosition(self, dt):
        self.applyAcceleration(self.gravityVector, dt)
        Body.updatePosition(self, dt)
```

It adds another attribute, `gravityVector`, which stores the gravity vector on our world.[1] It needs to change the way `__init__` works, because when we make a new body with gravity, we have to specify the gravity vector. So we include an `__init__` method in the `BodyWithGravity` class, which *overrides* the original `__init__` method from `Body`. If you look inside this method, it does something kind of tricky: it needs to do its own business, which is assigning the gravity vector, but then it needs to be sure that the base class, `Body`, has its `__init__` method called as well. So, we call it explicitly.

We also override the `updatePosition` method, in a similar way. It applies the gravity force, then calls the `updatePosition` method from the `Body` class.

Question 7. Explain to your LA why we can't just call `self.updatePosition` here.

Your job is to implement a class called `Rocket`, which is a subclass of `BodyWithGravity`. A rocket

[1] Why a vector?? In our initial experiments with this class, we'll actually live in a one-dimensional world, in which the only direction is up/down, so the gravity "vector" will just be a negative scalar (that is, pointing down). But later in the assignment, we'll extend to a two-dimensional world.

has two new attributes, `fuel` and a `thrustVector`.[2] Fuel weighs one unit of mass per unit of fuel. The `thrustVector` specifies the direction and force of the thrust generated by the rocket (we're assuming it applies a constant force until it runs out of fuel). Every time the position is updated, if there is any fuel remaining, the following things should happen:

- Fuel is consumed; we'll assume it consumes one unit of fuel per unit time, but remember that each position update is meant to account for `dt` amount of time.
- The mass is decreased by the amount of fuel consumed.
- The `thrustVector` is applied.
- The position is updated.
- If there is no fuel remaining, then it's just a falling body.

If we were running this with a very small `dt`, then it wouldn't much matter what order we did these steps in. As it is, because we'd like to compare your results to ours after a few steps, please do them in this order.

To create a rocket, we'll need to initialize all of these things. Here is a call that should create a rocket with initial mass 20, fuel 10, initial position 0, initial velocity 0, gravity vector -1, and thrust vector 100.

```
>>> r1 = Rocket(20, 10, 0, 0, -1, 100)
```

Assume the mass argument includes the mass of the fuel as well as the rocket.

Question 8. What behavior would this rocket display?

```
>>> r1 = Rocket(20, 10, 0, 0, -1, 0)
>>> r1.updatePosition(1.0)
```

Question 9. What about this one?

```
>>> r1 = Rocket(20, 10, 0, 0, 0, 1)
>>> r1.updatePosition(1.0)
```

Now, make the `Rocket` class, and define `__init__` and `updatePosition` methods for it. You might find it handy to print out the state of the rocket (position, velocity, mass fuel) whenever the position is updated.

Question 10. Make some other test cases and demonstrate them to your LA, and argue why the answers you're getting are right.

Question 11. Submit this code as your answer to the second tutor question for problem set 3.

Into the Second Dimension

Now, we'll combine the first two parts of this assignment, so that we can make rockets in two dimensions. Add the following statement to the top of the file that your `Rocket` class is in:

[2] As with gravity, for now, in a one-dimensional world, you can think of it as always being a positive number, in units of gravitys (thrust is up, gravity is down; you could do it the other way, if you wanted, to flout convention, and it would be logically fine, but your answers probably wouldn't work when checked by the tutor.)

```
from V2 import *
```

Be sure that your `V2.py` file is in the same directory as your rocket file. This will have the effect of importing all of the entries from the environment associated with the `V2` module into this environment. So, you can write `V2(1.0, 2.0)` in your rocket file, and it will make an instance of your `V2` class.

Let the first component of our two-dimensional space be x (think of the rocket flying across your screen), and the second one be y. Gravity should apply in the $-y$ direction and thrust in the $+y$ direction. Now, all our positions, velocities, and accelerations will be two-dimensional vectors, represented as `V2`s.

Question 12. Construct a rocket in two dimensions with zero gravity and zero thrust in the x dimension. Verify that it behaves the same as one of your rockets from the previous section. **You don't need to write any code to do this!!**

Question 13. Now construct a rocket that thrusts at a 45-degree angle. Run it for a while. Does its trajectory make sense?

Question 14. Explain to your LA why you didn't have to write any code to do this, and what you'd have to do to make it work in three dimensions.

Post-Lab Exercises: Due before lab on February 27

Polynomial Arithmetic

Write a class called `Polynomial` that supports arithmetic on polynomials. You can structure it any way you'd like, but it should be correct and be beautiful. It should support, at least, an interaction like this:

```
>>> p1 = Polynomial([3, 2, 1])
>>> print p1
poly[3, 2, 1]
>>> p2 = Polynomial([100, 200])
>>> print p1 + p2
poly[3.0, 102, 201]
>>> print p1 * p1
poly[9.0, 12.0, 10.0, 4.0, 1.0]
>>> print (p1 * p1) + p1
poly[9.0, 12.0, 13.0, 6.0, 2.0]
>>> p1.val(1)
6.0
>>> p1.val(10)
321.0
>>> p1.roots()
[(-0.33333333333333331+0.47140452079103173j),
 (-0.33333333333333331-0.47140452079103173j)]
>>> p3 = Polynomial([3, 2, -1])
```

```
>>> p3.roots()
[(0.33333333333333331+0j), (-1+0j)]
>>> (p1 * p2).roots()
Order too high to solve for roots.  Try Newton.
```

The constructor accepts a list of coefficients, highest-order first. You only have to compute the roots if is quadratic or less, otherwise `roots` can generate an error message.[3]

As for beauty, try to do things as simply as possible. Don't do anything twice. If you need some extra procedures to help you do your work, you can put them in the same file as your class definition, but outside the class (so, put them at the end of the file, with no indentation). For instance, LPK found it useful to write helper functions to: add two lists of numbers of the same length, to extend a list of numbers to a particular length by adding zeros at the front, and to compute quadratic roots (actually, she just pasted in her old version). To evaluate the polynomial at a particular input `x`, you can use a procedure that you defined last week. If you want a little more exercise, try writng a version of Horner's rule that is not recursive (but is still very small and beautiful).

What to turn in

Question 15. Submit this code as your answer to the third tutor question for problem set 3.

Question 16. In addition, turn in a hardcopy of your code for this class. Explain, in comments, your strategy for representing a polynomial internally, as well as for addition and multiplication.

Exploration

There are lots of things you can do to extend this assignment.

- Make three-dimensional rockets
- Use our graphing code to plot the trajectory of a 1D rocket over time, or a 2D rocket through space.
- Make an anti-missile-missile.
- Add Newton's method to the polynomial class. Try to be smart about finding starting points (and all the roots).

[3] But we'd be delighted if you wanted to implement higher-order root-finding as an extension.