Design Lab 1 *6.01 – Fall 2011*

Object-Oriented Programming

Goals:

- Get familiar with the 6.01 environment and on-line Tutor
- Practice concepts of software engineering: Primitives, Combination, Abstraction, Patterns
- Design and implement an abstract method to operate on polynomials

1 Introduction

Welcome to your first 6.01 design lab! These labs are generally intended to let you explore concepts introduced in the lecture, working together with a lab partner. Each design lab consists of instructions for setup, and a set of problems, which are done with robots, instrumentation, computers, and electronics components in the laboratory.

Each problem is specified by its **objectives**, the documentation of the **resources** provided, and occasionally detailed **guidance** of the steps involved. Problem answers are generally entered into the 6.01 Online Tutor. Some problem specifications will be provided entirely on the Tutor.

1.1 Setup

Due to the late start of classes this term, today's design lab is a bit unusual. Design labs are generally done with partners, but this one should be done individually.

Laptops: Please use a lab laptop unless you have already installed Python 2.6.x on yours. If you have not installed Python, please do so after class.

Python: **If you have already worked through** the Python programming tutor and/or have had other Python experience, then go ahead and do the problems below. **If not**, then please work through the Python tutor.

Some of the software and design labs contain the command athrun 6.01 getFiles. Please disregard this instruction; the same files are available on the 6.01 OCW Scholar site as a .zip file, labeled Code for [Design or Software Lab number].

2 PCAP Exercises

The following are three problems covering concepts of Primitives, Combination, Abstraction, and Patterns.

2.1 Fibonacci

Objective: Write the definition of a Python procedure `fib`, such that `fib(n)` returns the n^{th} Fibonacci number. Recall the definition of $\text{fib}(n)$:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1 \end{cases}$$

Resources:

- `~/Desktop/6.01/designLab01/designLab01Work.py`: a template for your `fib` procedure implementation
- Tutor problem **Wk.1.3.1**: where your answer should be entered

Detailed guidance : We recommend using **idle**, an integrated development enviroment for Python. In the Terminal window, type
`idle &`
You can type Python expressions in `idle`'s Python Shell window.
You can write your programs in a file and test them using `Run Module`.
For example:

- Click `idle`'s File menu, select New Window, and write the following line in the window:
  `print 'Hello World'`
- Click `idle`'s File menu, select Save as, and type `~/Desktop/6.01/designLab01/test.py` in the filename box.

- Click `idle`'s Run menu, then select Run Module.
- Look at the Python Shell window: you should see `Hello World`.

Now look at problem **Wk.1.3.1** on the Tutor.

- Click Idle's `File` menu, select `Open`, navigate to `~/Desktop/6.01/designLab01/` and choose the file name `designLab01Work.py`.
- Complete the definition in this window. After the definition include some test cases (remember to print the result). So, your file should look like:

```
def fib (n):
  ...
print fib(1)
print fib(6)
 ...
```

- Click Idle's `File` menu, select `Save`
- Click Idle's `Run`, select `Run Module` (or use `F5`).
- Look at the `Python Shell` window for your results.
- Debug `fib` in `idle` until it is correct. When something goes wrong, read the error message carefully to what the problem was. If it doesn't make sense to you, ask a staff member for help.

Wk.1.3.1	Check your results by copying the text of your procedure from `idle` and pasting it into the tutor problem **Wk.1.3.1**. Submit your answer when it passes all the tests.

2.2 Object-Oriented Practice

Wk.1.3.2	Get some practice with object-oriented concepts in this tutor problem.

Wk.1.3.3	Get some more practice with object-oriented concepts in this tutor problem.

2.3 Two-dimensional vectors

Objective: Define a Python class `V2`, which represents two-dimensional vectors and supports the following operations:

- Create a new vector out of two real numbers: `v = V2(1.1, 2.2)`
- Convert a vector to a string (with the `__str__` method)
- Access the components (with the `getX` and `getY` methods)
- Add two `V2`s to get a new `V2` (with `add` and `__add__` methods)
- Multiply a `V2` by a scalar (real or int) and return a new `V2` (with the `mul` and `__mul__` methods)

Resources:

- `~/Desktop/6.01/designLab01/designLab01Work.py`: a template for your `V2` class implementation
- Tutor problem **Wk.1.3.4**: where your answer should be entered

Detailed guidance : Open file `designLab01Work.py` and do the following:

Step 1. Define the basic parts of your class, with an `__init__` method and a `__str__` method, so that if you do

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

it prints

```
V2[1.1, 2.2]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example. Remember that `str(x)` turns `x`, whatever it is, into a string. Don't worry about making this beautiful!

Step 2. Write two accessor methods, `getX` and `getY` that return the `x` and `y` components of your vector, respectively. For example,

```
>>> v = V2(1.0, 2.0)
>>> v.getX()
1.0
>>> v.getY()
2.0
```

Step 3. Define the `add` and `mul` methods, so that you get the following behavior:

```
>>> a = V2(1.0, 2.0)
>>> b = V2(2.2, 3.3)
>>> print a.add(b)
V2[3.2, 5.3]
>>> print a.mul(2)
V2[2.0, 4.0]
>>> print a.add(b).mul(-1)
V2[-3.2, -5.3]
```

Step 4. 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)
```

then you can do

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

Add to the class the `__add__` method, which should call your `add` method to add vectors, and the `__mul__` method, which should call your `mul` method to multiply the vector by a scalar. The scalar will always be the second argument.

Test your implementation in `idle` until it seems correct to you.

Wk.1.3.4 Check your results by copying the text of your procedure from `idle` and pasting it into the tutor problem **Wk.1.3.4**.

3 Polynomial Class

Objective: Define a Python class `Polynomial` which provides methods for performing algebraic operations on polynomials. Your class should behave as described in the following sample transcript:

```
>>> p1 = Polynomial([1, 2, 3])
>>> p1
1.000 z**2 + 2.000 z + 3.000
>>> p2 = Polynomial([100, 200])
>>> p1.add(p2)
1.000 z**2 + 102.000 z + 203.000
>>> p1 + p2
1.000 z**2 + 102.000 z + 203.000
>>> p1(1)
6.0
>>> p1(-1)
2.0
>>> (p1 + p2)(10)
1323.0
>>> p1.mul(p1)
1.000 z**4 + 4.000 z**3 + 10.000 z**2 + 12.000 z + 9.000
>>> p1 * p1
1.000 z**4 + 4.000 z**3 + 10.000 z**2 + 12.000 z + 9.000
>>> p1 * p2 + p1
100.000 z**3 + 401.000 z**2 + 702.000 z + 603.000
>>> p1.roots()
[(-1+1.4142135623730947j), (-1-1.4142135623730947j)]
>>> p2.roots()
[-2.0]
>>> p3 = Polynomial([3, 2, -1])
>>> p3.roots()
[-1.0, 0.33333333333333331]
>>> (p1 * p1).roots()
Order too high to solve for roots.
```

Resources:

- `~/Desktop/6.01/designLab01/designLab01Work.py`: contains a template definition of the `Polynomial` class
- Tutor Problem **Wk.1.3.5**: exercise about representations of polynomials
- Tutor Problem **Wk.1.3.6**: where your answer should be entered

Detailed guidance : The following two subsections provide steps to follow for implementing the `Polynomial` class.

3.1 Representation

We can represent a polynomial as a list of coefficients starting with the highest-order term. For example, here are some polynomials and their representations as lists:

$x^4 - 7x^3 + 10x^2 - 4x + 6$	`[1, -7, 10, -4, 6]`
$3x^3$	`[3, 0, 0, 0]`
8	[8]

Wk.1.3.5 It is a little bit tricky to implement addition and multiplication of polynomials. Do tutor problem **Wk.1.3.5** before you start programming, and **be sure you understand the results in the example transcript provided above.**

3.2 Operations

Edit the template definition of the `Polynomial` class in `designLab01Work.py`, and add the following:

- An attribute called `coeffs`, which is the list of coefficients used to create the instance. It must have this name or the tests in the tutor will fail.
- `__init__(self, coefficients)`: initializes the `coeffs` attribute to be a list of *floating-point* coefficient values.
- `coeff(self, i)`: returns the coefficient of the x^i term of the polynomial. For example, if the polynomial is $x^4 - 7x^3 + 10x^2 - 4x + 6$, then `coeff(self, 3)` is `-7`.
- `add(self, other)`: returns a new `Polynomial` representing the sum of `Polynomials` `self` and `other`. **Be sure that performing any operation on polynomials, e.g. `p1 + p2`, *does not change* the original value of `p1` or `p2`.**
- `mul(self, other)`: returns a new `Polynomial` representing the product of `Polynomials` `self` and `other`
- `__str__(self)`: converts a `Polynomial` into a string. Do the simplest thing that shows the coefficients; remember that `str(x)` turns `x`, whatever it is, into a string. After you're done with everything else, go back and change your `__str__` method to print polynomials out as they are shown in the transcript at the end. This is not required; do it only if you have time and interest.

- `val(self, v)`: returns the numerical result of evaluating the polynomial when `x` equals `v`.
- `roots(self)`: returns a list containing the root or roots of first or second order polynomials (for orders other than 1 and 2, just print an error message saying that you don't handle them). If the roots are real-valued, then return the roots as floats. If a root has a non-zero imaginary part, then return it as a complex number. Python has built-in facilities for handling complex numbers. For example, `complex(3,2)` represents a complex number whose real part is 3 and whose imaginary part is 2. This same complex number could also be written as `3+2j`. The real part of a complex number `z` can be obtained with `z.real`. You can take square roots by using a fractional exponent. For example `3**0.5` represents the square root of 3. Similarly `complex(3,2)**0.5` represents the square root of the complex number $3 + 2j$. Python tries to return a real-valued result when it raises a real number to a fractional power. It returns a complex-valued result when it raises a complex number to a fractional power. Therefore `(-4)**0.5` generates an error, while `complex(-4,0)**0.5` returns an answer that is close to 2j.

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).

3.3 Operator overloading

In order to use expressions like `p1 + p2`, `p1 * p2`, and `p1(3)`, for addition, multiplication, and evaluation, respectively, define the specially-named methods `__add__`, `__mul__`, and `__call__`. So for example, include

```
def __add__(self, other):
    return self.add(other)
def __mul__(self, other):
    return self.mul(other)
def __call__(self, x):
    return self.val(x)
```

Also, in order to have your polynomials printed out nicely by the Python shell, you can add this line to your class:

```
def __repr__(self):
    return str(self)
```

which says that the the shell should print the string returned by the `__str__` method.

Wk.1.3.6 After you have debugged in idle, check and submit your results by copying the text of your class and associated definitions from idle and pasting it into the tutor problem **Wk.1.3.6**.

3.4 Optional

There's a particularly elegant way to implement the `val` method, using *Horner's Rule*. For computing the value of a polynomial, it structures the computation of

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

as

$$(\cdots(a_n x + a_{n-1})x + \cdots + a_1)\, x + a_0$$

In other words, we start with a_n, multiply the entire result by x, add a_{n-1}, multiply by x, and so on, until we reach a_0. For example, we'd evaluate $8x^3 - 3x^2 + 4x + 1$ as

$$((8 \cdot x - 3) \cdot x + 4) \cdot x + 1$$

For fun, try implementing `val` with Horner's rule. Think about how many multiplication operations it takes to evaluate a polynomial using Horner's rule, compared to the usual way.