Assignment 4: Modeling with Classes: Complex and Polynomials

EC602 Design by Software

Fall 2017

Contents

1 Introduction

1.1 Assignment Goals

The assignment goals are to

- illustrate the advantages of modeling data using classes
- introduce complex numbers
- introduce python's class mechanism

1.2 Group Size

For this assignment, the maximum group size is 3.

1.3 Due Date

This assignment is due 2017-10-03 at midnight.

1.4 Submission Link

You can submit here: week 4 submit link

Please use the checker locally before submitting.

1.5 Points

This assignment is worth 6 points.

1.6 The checker

Here is the checker: modeling_checker.py

This checker requires a new version of ec602lib.py. Please download it.

2 Background: Modeling data with Classes

The idea of a `class` is useful not only as a way of structuring code, but as a way of thinking about:

- modeling
- abstraction

- defining specifications
- defining expectations
- defining requirements
- organization
- relationships between parts of a larger system

More concretely, we can used classes to

- organize and collect related information
- define methods that operate on the object
- allow for integration into the language using operators.

In the C++ standard library and in Python's standard library most code is provided in the form of families of related classes.

2.1 Classes and Objects

A class is like a type.

An object is like a variable.

In python, everything is an object.

3 Example: Polynomials

In week 3, we developed code that worked with polynomials.

However, our model for a polynomial (using a list or vector) has several issues

- the ordering needs to be understood by the caller of add_poly() and multiply_poly()
- there is no way to represent negative powers of x
- inefficient in the case of sparse polynomials like $2x^{10001} + 1$
- functions are all separate and un-organized.
- can't type C = A + B , or D = A * B

4 Example: Complex Numbers

In python, the built in class `complex` models complex numbers.

You may use `j` as the imaginary number (not `i`), like this:

```
c = 3 + 4j
```

or

```
d = 4.3 + 4.2j
```

or define using the initializer

```
f = complex(3,4)
```

4.1 The `Complex` class.

Read this section (and the code provided) carefully and fully: it will be necessary to complete the assignment.

Here is a python re-implementation of the built-in `complex` class.

The new class is called `Complex`.

The functions starting with two underscores `__` are special names that allow python to map operations to them. So, for example, if `z1` and `z2` are of type `Complex`, then

```
total = z1 + z2
```

is equivalent to

```
total = z1.__add__(z2)
```

and is also equivalent to

```
total = Complex.__add__(z1,z2)
```

Some people call this *syntactic sugar* since it makes the code *sweeter*.

Two underscores is pronounced *dunder* in the python world, short for *Double UNDERscore*.

Note that the class `Complex` is incomplete. Some operations have not been implemented.

```
class Complex():
    "Complex(real,[imag]) -> a complex number"

    def __init__(self,real=0,imag=0):
        self.real=real
        self.imag=imag

    def __abs__(self):
        "abs(self)"
        return (self.real**2 +self.imag**2)**0.5

    def __add__(self,value):
        "Return self+value."
        if hasattr(value,'imag'):
            return Complex(self.real+value.real,self.imag+value.imag)
```

```
        else:
            return Complex(self.real+value,self.imag)

    def __mul__(s,v):
        "Return s*v."
        if hasattr(v,'imag'):
            x = s.real * v.real - s.imag * v.imag
            y = s.real * v.imag + v.real * s.imag
            return Complex(x,y)
        else:
            return Complex(v*s.real,v*s.imag)

    def __rmul__(s,v):
        "Return s*v"
        if hasattr(v,'imag'):
            x = s.real * v.real - s.imag * v.imag
            y = s.real * v.imag + v.real * s.imag
            return Complex(x,y)
        else:
            return Complex(v*s.real,v*s.imag)

    def __radd__(self,value):
        "Return self+value"
        if hasattr(value,'imag'):
            return Complex(self.real+value.real,self.imag+value.imag)
        else:
            return Complex(self.real+value,self.imag)

    def __str__(self):
        if self.real==0:
            return "{}j".format(self.imag)
        else:
            sign="-" if self.imag<0 else "+"
            return "({}{}{}j)".format(self.real,sign,abs(self.imag))

    def __repr__(self):
        return str(self)

    def __pow__(self,value):
        "Return self ** value"
        raise NotImplementedError('not done yet')
```

The full program along with some test code is here: model_complex.py

Another test program is provided here: fancy_tester.py

5 Just for fun: Mandelbrot sets.

To illustrate complex numbers and what python can do, we are including two programs that plot the Mandelbrot set

Here is the source code for `plot_mandelbrot_two.py`

```
import numpy as np
import matplotlib.pyplot as plt
def bounded_box(c):
    return ((-2.0 < c.real <2.0) and (-2.0 < c.imag < 2.0))

def bounded_circle(c):
    return abs(c) < 2.0

def mandelbrot(c,maxiteration=12,boundedfcn=bounded_box):
    z,count=0,0
    while boundedfcn(z) and count<maxiteration:
        z = z*z + c
        count += 1
    return count

X = Y = np.r_[-2.1:2.1:1001j]

XX,YY = np.meshgrid(X,X)
Z = XX+1j*YY

array_mandelbrot = np.vectorize(mandelbrot)

M = array_mandelbrot(Z)
plt.imshow(M, interpolation='nearest', cmap='jet')
plt.axis('off')
plt.show()
```

Try them out here:

- plot_mandelbrot_one.py
- plot_mandelbrot_two.py

6 The assignment

Write a python program that implements polynomials by defining a class Polynomial.

You may not import any modules.

Here are the requirements:

- implement a constructor which takes a sequence and assigns the coefficients in the natural (descending order). So `Polynomial([4,-9,5.6])` should make $4x^2 - 9x + 5.6$
- implement addition and subtraction of polynomials using `+` and `-`
- implement multiplication of polynomials using `*`
- implement testing for equality of polynomials using `==`
- implement an efficient mechanism for handling sparse polynomials
- implement negative powers in the polynomial, i.e. you should be able to handle $p(x) = x^{-1}$
- implement evaluation of the polynomial using a `eval` method, like this: p.eval(2.1)
- implement accessing and modifying the coefficients using `[]`. So `p[2]` should be the coefficient of x^2 and `p[8] = 12` should set the coefficent of x^8 to 12.
- implement a derivative method `p.deriv()` which returns the derivative of the polynomial.

During testing, the coefficients and variable `x` may be integers, float, or complex. All three cases should work.

Your program should be *importable* which means it does absolutely nothing when imported except define the class `Polynomial`. Here is the basic template for doing this shown below:

```
class Polynomial():
    pass

def main():
    pass

if __name__=="__main__":
    main()
```

Your program should be called `modeling.py`

When we test your code, it will be done like this:

```
from modeling import Polynomial
```

and so no code gets executed except that the class `Polynomial` will be defined.

7 Notes and Hints

7.1 Output and testing

You do not need to implement any output facilities for your class, but it will be very difficult for you to test your own code if you do not.

Furthermore, the checker will use the functions `str` and `repr` (which you can implement with `__str__` and `__repr__`) to provide you with useful information about what is or is not working.

7.2 Dealing with negative polynomials.

The constructor does not provide any mechanism for creating a Polynomial with negative exponents. This can only be done using `[]`, like this

```
q = Polynomial( [] )
q[-2] = 5.1
```

The first line creates `q`, a Polynomial with no values. Hence $q(x) = 0$. The second line sets the coefficient of x^{-2} to `5.1`, so $q(x) = 5.1x^{-2}$

7.3 Dealing with missing exponents.

Using the constructor, you must specify missing coefficients by adding zeros to the sequence. For example, to construct $3.1x^2$, you could use

```
p = Polynomial([3.1, 0, 0])
```

or

```
p = Polynomial([])
p[2] = 3.1
```

7.4 About sequences

In python, a *sequence* is any collection that has elements that can be accessed by index starting at 0 all the way up to N-1 where N is the length of the list.

Examples of sequences are

- string
- list
- tuple
- numpy.ndarray

If `s` is a sequence, then `s[i]` is the element at position `i`, the valid values for `i` being 0 through `N-1`. Negative numbers are also allowed, they specify count from the end. So `s[-1]` is the last element in the sequence.

The input to the constructor of `Polynomial` must be a sequence with numeric values.