E2_exercises_on_oop

September 6, 2019

0.1 Exercises — Week 2

0.1.1 Introduction to Object-Oriented Programming

These weeks exercises starts you working with classes. If you want a gentler introduction, exercises for Chapter 7 in Langtangen is recommended.

0.1.2 Exercise 1 — Quadratic functions

In this exercise, we will build on the example given in the lectures of implementing 2nd degree polynomials as objects of a custom defined class. A general 2nd degree polynomial, aka, quadratic function, can be written as:

$$f(x) = a_2 x^2 + a_1 x + a_0,$$

where the coefficients, a_2, a_1, and a_0 uniquely defines the polynomial.

Exercise 1a) Defining the `Quadratic` class Create a class, `Quadratic`, that represents a general 2nd degree polynomial. Define the following methods: * A constructor (`__init__`) * A call method (`__call__`) The constructor should take in the three coefficients in order: `a_2`, `a_1`, and `a_0`, and the call method should take the free variable `x`.

You class should be able to handle the following test script: ***

```
f = Quadratic(1, -2, 1)
x = np.linspace(-5, 5, 101)
plt.plot(x, f(x))
plt.show()
```

Implement your solution here:

[ ]:

```

```

Use this to test your implementation:

[ ]:

```
def test_Quadratic():
    f = Quadratic(1, -2, 1)
    assert abs(f(-1) - 4) < 1e-8
    assert abs(f(0)  - 1) < 1e-8
    assert abs(f(1)  - 0) < 1e-8
```

```
test_Quadratic()
```

Exercise 1b) Pretty printing Extend your `Quadratic` class with a string special method (`__str__`) so that you can print a Polynomial object and get the polynomial written out on a readable form. Test by creating a polynomial object and printing it out.

Exercise 1c) Adding together polynomials Adding together two general quadratic functions:

$$f(x) = a_2 x^2 + a_1 x + a_0, \qquad g(x) = b_2 x^2 + b_1 x + b_0,$$

gives a new quadratic function:

$$(f+g)(x) = (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0)$$

Implement this functionality using the addition special method (`__add__`). This method should return a new Quadratic-object, without changing the two that are added together. Your new class should be able to handle the following test script: ***

```
f = Quadratic(1, -2, 1)
g = Quadratic(-1, 6, -3)

h = f + g
print(h)

x = np.linspace(-5, 5, 101)
plt.plot(x, h(x))
plt.show()
```

(Because $a_2 + b_2 = 0$, the resulting plot should be a straight line.)
Implement your solution here:

[ ]:
```

```

Use this to test your implementation:

[ ]:
```
def test_Quadratic_add():
    f = Quadratic(1, -2, 1)
    g = Quadratic(-1, 6, -3)
    h = f + g
    a2, a1, a0 = h.coeffs
    assert a2 == 0
    assert a1 == 4
    assert a0 == -2

test_Quadratic_add()
```

Exercise 1d) Finding the roots The roots of a general quadratic function,

$$f(x) = ax^2 + bx + c = 0,$$

are given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Extend your `Quadratic` function with a method `.roots()` that finds and returns the real roots of the function (ignore the imaginary ones). Return the result as a tuple with 0, 1, or 2 elements.

Test your method on the three polynomials: * $2x^2 - 2x + 2$ * $x^2 - 2x + 1$ * $x^2 - 3x + 2$

Implement your solution here:

[ ]:
```

```

Use this to test your implementation:

[ ]:
```
def test_Quadratic_root():
    f1 = Quadratic(2, -2, 2)
    f2 = Quadratic(1, -2, 1)
    f3 = Quadratic(1, -3, 2)

    assert f1.roots() == ()
    assert abs(f2.roots()[0] - 1) < 1e-8
    assert abs(f3.roots()[0] - 1) < 1e-8 and abs(f3.roots()[1] - 2) < 1e-8

test_Quadratic_root()
```

Exercise 1e) Finding the intersection of two quadratic functions Extend your class with a method that finds and returns the intersection points (if any) between two `Quadratic`-objects. It should work as follows:

```
f = Quadratic(1, -2, 1)
g = Quadratic(2, 3, -2)

print(f.intersect(g))
```

Hint: The intersections are all points solving $f(x) = g(x)$, which can be written as $(f - g)(x) = 0$.

Test your solution by plotting the two functions and their intersections:

$$f(x) = x^2 - 2x + 1, \qquad g(x) = 2x^2 + 3x - 2.$$

0.1.3 Exercise 2 — A class for general polynomials

We now turn to looking at general polynomials of degree n. These can be written as

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

or more compactly as

$$\sum_{k=0}^{n} a_k x^k.$$

We want to make a class that represents such a polynomial, and can take any number of coefficients in. The constructor of such a class could for example take in a list of the coefficients: `[a0, a1, ..., aN]`. However, this list will always have to be of length N, and say we want to specify the polynomial $x^{1000} + 1$, it is highly inefficient to pass in such a long list, as most coefficients are actually 0.

A better approach is to use a dictionary, where we use the index as the key and the coefficient as the value. Doing this, we can then specify only the non-zero coefficients, and simply skip those that are 0. So defining $x^{1000} + 1$ would simply be: `Polynomial({0: 1, 1000: 1})`.

Exercise 2a) Defining the Polynomial class Define the `Polynomial` class with the following methods * A constructor (`__init__`) that takes in the coefficients of the polynomial as a dictionary * A call method (`__call__`) that computes f(x) for a given x * A string method (`__str__`) for informative printing of the polynomial

Your class should be able to handle the following test script

---

```
coeffs = {0: 1, 5:-1, 10:1}
f = Polynomial(coeffs)

print(f)

x = linspace(-1, 1, 101)
plt.plot(x, f(x))
plt.show()
```

Implement your solution here:

[ ]:

```

```

Exercise 2b): Adding general polynomials together We now want to be able to add together two general polynomial objects, which should produce a new general polynomial object. Mathematically, this is just an extension of the 2nd degree polynomial case which we saw in exercise (1). If we have

$$f(x) = \sum_{k=0}^{m} a_k x^k, \qquad g(x) = \sum_{k=0}^{n} b_k x^k,$$

the sum will be defined by

$$(f+g)(x) = \sum_{k=0}^{\max(m,n)} (a_k + b_k)x^k.$$

Thus, if we add together two polynomials of degree m and n, then the sum will have degree $\max(m,n)$, i.e., the largest of the two.

Extend your class to add this functionality using the addition special method (`__add__`).

The class should handle the following test case: ***

```
f = Polynomial({0:1, 5:-7, 10:1})
g = Polynomial({5:7, 10:1, 15:-3})

print(f+g)
```

Which should produce the output: $-3x^{15} + 2x^{10} + 1$

Implement your solution here:

[ ]:
```

```

Hint: You will need to create a new coefficient dictionary for the new polynomial and add in the coefficients from the two polynomials. This can be slightly tricky getting the keys right. Here `collections.defaultdict` can be useful, but it isn't necessary.

Exercise 2c) Defining a `AddableDictionary` class The previous exercise would have been a lot simpler, if we could simply add two dictionary objects together as follows:

```
a = {0: 2, 1: 3, 2: 4}
b = {0: -1, 1:3, 2: 3, 3: 2}
c = a + b
```

However, if you try to do this, you get an exception: > `TypeError: unsupported operand type(s) for +: 'dict' and 'dict'`

This means that there is no addition special method defined for dictionaries. However, we can extend the normal dictionary class to include this by adding a special method as follows

```
class AddableDict(dict):
    def __add__(self, other):
        ...
```

Add the necessary code, so that our new `AddableDict` class can add two dictionaries together as follows:

```
a = AddableDict({0: 2, 1: 3, 2: 4})
b = AddableDict({0: -1, 1:3, 2: 3, 3: 2})
print(a + b)
```

And give the ouput: `{0: 1, 1: 6, 2: 7, 3: 2}`.

Implement your solution here:

[ ]:
```

```

Use this to test your implementation:

[ ]:
```python
def test_AddableDict():
    a = AddableDict({0: 2, 1: 3, 2: 4})
    b = AddableDict({0: -1, 1:3, 2: 3, 3: 2})
    c = a + b
    assert c[0] == 1
    assert c[1] == 6
    assert c[2] == 7
    assert c[3] == 2

test_AddableDict()
```

Having made the `AddableDict class`, go back and change the Polynomial constructor, so that even if the user sends in the coefficients as a normal dictionary, it is converted to an `AddableDict` inside the Polynomial. Having done this, rewrite `Polynomial.__add__`, which should be trivial.

Exercise 2d) Derivative of a polynomial It is also the case that the derivative of a polynomial is a polynomial, if we have

$$f(x) = \sum_{k=0}^{m} a_k x^k,$$

then we get

$$f'(x) = \sum_{k=1}^{m} (a_k \cdot k) x^{k-1},$$

which can be written as

$$f'(x) = \sum_{k=0}^{m-1} b_k x^k,$$

where $b_k = (k+1)a_{k+1}$.

Implement a method, `derivative`, that returns the function $f'(x)$ as a new Polynomial object. Test your function by finding the derivative of

$$f(x) = x^{10} - 3x^6 + 2x^2 + 1.$$

Implement your solution here:

[ ]:
```

```

Use this to test your implementation:

[ ]:
```python
def test_derivative():
    f = Polynomial({10:1, 6:-3, 2:2, 0:1})
    f_deriv = f.derivative()
    assert f_deriv.coeffs == {9:10, 5:-18, 1:4}

test_derivative()
```

Exercise 2e) Multiplying polynomials It is also the case that the *product* of two polynomials form a new polynomial. If we again define

$$f(x) = \sum_{k=0}^{m} a_k x^k, \qquad g(x) = \sum_{k=0}^{n} b_k x^k,$$

then the product is given by

$$(f \cdot g)(x) = \left(\sum_{i=0}^{m} a_i x^i \right) \cdot \left(\sum_{j=0}^{n} b_j x^j \right) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i b_j x^{i+j}$$

Implement this functionality using the multiplication special method (`__mul__`). To acomplish this, you will need two nested for-loops over the coefficient dictionaries.

Test your implementation with the code block ***

```
f = Polynomial({2: 4, 1: 1})
g = Polynomial({3: 3, 0: 1})
print(f*g)
```

Which should give the output:

$$(4x^2 + x)(3x^3 + 1) = 12x^5 + 3x^4 + 4x^2 + x$$

Implement your solution here:

[ ]:
```

```

Use this to test your implementation:

[ ]:
```
def test_Polynomial_mul():
    f = Polynomial({2: 4, 1: 1})
    g = Polynomial({3: 3, 0: 1})
    h = f*g
    assert h.coeffs == {5:12, 4:3, 2:4, 1:1}

test_Polynomial_mul()
```

0.1.4 Exercise 3 - Quantum Harmonic Oscillator in One Dimension

The quantum harmonic oscillator wave function, $\psi_n(x)$ is a solution to the time-independent Scrödinger equation

$$\hat{H}\psi_n(x) = E_n \psi_n(x),$$

where $\hat{H}$ is the quantum harmonic oscillator hamiltonian and E_n is the energy at the n-th level (n must be a positive integer, $n = 0, 1, 2, \ldots$).

In this excersise you will implement a `class HOWF` which represents a quantum harmonic oscillator wave function in one dimension.