15-112: Introduction to Programming and Computer Science, Spring 2020

Homework 7: Objecct Oriented Programming

Due: Tuesday, March 17, 2020 by 22:00

This assignment has 2 questions, for a total of 50 points.
To start this homework....

1. Create a folder named "week7"
2. Create a file called hw7.py and write all your code in that file.
3. When you have completed and fully tested hw6, submit hw7.py to Autolab. For this hw, you may submit up to 12 times, but only your last submission counts.

Some important notes:

1. After you submit to Autolab, make sure you check your score. If you arenâĂŹt sure how to do this, then ask a CA or Professor.
2. There is no partial credit on Autolab testcases. Your Autolab score is your Autolab score.
3. Read the last bullet point again. Seriously, we wonâĂŹt go back later and increase your Autolab score for any reason. Even if you worked really hard and it was only a minor error...
4. Do not hardcode the test cases in your solutions.
5. Remember the course's academic integrity policy. Solving the homework yourself is your best preparation for exams and quizzes; cheating or short-cutting your learning process in order to improve your homework score will actually hurt your course grade long-term.
6. Your code will be graded for style. Check the style notes on the website for details.

1. [20 points] **Vending Machines** Write the VendingMachine class so that it passes testVendingMachineClass, and uses the OOP constructs we learned this week as appropriate.

   Hint: Make sure you're representing money properly wherever it might appear in the string representation of a Vending Machine. The test cases shown here don't cover every possible case.

```
def testVendingMachineClass():
  print("Testing Vending Machine class...", end="")
  # Vending machines have three main properties:
  # how many bottles they contain, the price of a bottle, and
  # how much money has been paid. A new vending machine starts with no
  # money paid.
  vm1 = VendingMachine(100, 125)
  assert(str(vm1) == "Vending Machine:<100 bottles; $1.25 each; $0 paid>")
  assert(vm1.isEmpty() == False)
  assert(vm1.getBottleCount() == 100)
  assert(vm1.stillOwe() == 125)

  # When the user inserts money, the machine returns a message about their
  # status and any change they need as a tuple.
  assert(vm1.insertMoney(20) == ("Still owe $1.05", 0))
  assert(str(vm1) == "Vending Machine:<100 bottles; $1.25 each; $0.20 paid>")
  assert(vm1.stillOwe() == 105)
  assert(vm1.getBottleCount() == 100)
  assert(vm1.insertMoney(5) == ("Still owe $1", 0))

  # When the user has paid enough money, they get a bottle and
  # the money owed resets.
  assert(vm1.insertMoney(100) == ("Got a bottle!", 0))
  assert(vm1.getBottleCount() == 99)
  assert(vm1.stillOwe() == 125)
  assert(str(vm1) == "Vending Machine:<99 bottles; $1.25 each; $0 paid>")

  # If the user pays too much money, they get their change back with the
  # bottle.
  assert(vm1.insertMoney(500) == ("Got a bottle!", 375))
  assert(vm1.getBottleCount() == 98)
  assert(vm1.stillOwe() == 125)

  # Machines can become empty
  vm2 = VendingMachine(1, 120)
  assert(str(vm2) == "Vending Machine:<1 bottle; $1.20 each; $0 paid>")
  assert(vm2.isEmpty() == False)
  assert(vm2.insertMoney(120) == ("Got a bottle!", 0))
  assert(vm2.getBottleCount() == 0)
```

```
    assert(vm2.isEmpty() == True)

    # Once a machine is empty, it should not accept money until it is restocked.
    assert(str(vm2) == "Vending Machine:<0 bottles; $1.20 each; $0 paid>")
    assert(vm2.insertMoney(25) == ("Machine is empty", 25))
    assert(vm2.insertMoney(120) == ("Machine is empty", 120))
    assert(vm2.stillOwe() == 120)
    vm2.stockMachine(20) # Does not return anything
    assert(vm2.getBottleCount() == 20)
    assert(vm2.isEmpty() == False)
    assert(str(vm2) == "Vending Machine:<20 bottles; $1.20 each; $0 paid>")
    assert(vm2.insertMoney(25) == ("Still owe $0.95", 0))
    assert(vm2.stillOwe() == 95)
    vm2.stockMachine(20)
    assert(vm2.getBottleCount() == 40)

    # We should be able to test machines for basic functionality
    vm3 = VendingMachine(50, 100)
    vm4 = VendingMachine(50, 100)
    vm5 = VendingMachine(20, 100)
    vm6 = VendingMachine(50, 200)
    vm7 = "Vending Machine"
    assert(vm3 == vm4)
    assert(vm3 != vm5)
    assert(vm3 != vm6)
    assert(vm3 != vm7) # should not crash!
    assert(vm4.insertMoney(50) == ("Still owe $0.50", 0))
    assert(vm3 != vm4)
    print("Done!")
testVendingMachineClass()
```

2. **Equations**

   Write the Polynomial and Quadratic classes so that they pass testEquationClasses and use the OOP constructs we learned this week as appropriate.

   (a) [20 points] Polynomial Class should be designed so that it will pass the following testcases.

```
def testPolynomialBasics():
  # we'll use a very simple str format...
  assert(str(Polynomial([1,2,3])) == "Polynomial(coeffs=[1, 2, 3])")
  p1 = Polynomial([2, -3, 5])  # 2x**2 -3x + 5
  assert(p1.degree() == 2)

  # p.coeff(i) returns the coefficient for x**i
  assert(p1.coeff(0) == 5)
  assert(p1.coeff(1) == -3)
  assert(p1.coeff(2) == 2)

  # p.evalAt(x) returns the polynomial evaluated at that value of x
  assert(p1.evalAt(0) == 5)
  assert(p1.evalAt(2) == 7)

def testPolynomialEq():
  assert(Polynomial([1,2,3]) == Polynomial([1,2,3]))
  assert(Polynomial([1,2,3]) != Polynomial([1,2,3,0]))
  assert(Polynomial([1,2,3]) != Polynomial([1,2,0,3]))
  assert(Polynomial([1,2,3]) != Polynomial([1,-2,3]))
  assert(Polynomial([1,2,3]) != 42)
  assert(Polynomial([1,2,3]) != "Wahoo!")
  # A polynomial of degree 0 has to equal the same non-Polynomial numeric!
  assert(Polynomial([42]) == 42)

def testPolynomialConstructor():
  # If the list is empty, treat it the same as [0]
  assert(Polynomial([]) == Polynomial([0]))
  assert(Polynomial([]) != Polynomial([1]))
  # In fact, disregard all leading 0's in a polynomial
  assert(Polynomial([0,0,0,1,2]) == Polynomial([1,2]))
  assert(Polynomial([0,0,0,1,2]).degree() == 1)

  # Require that the constructor be non-destructive
  coeffs = [0,0,0,1,2]
  assert(Polynomial(coeffs) == Polynomial([1,2]))
  assert(coeffs == [0,0,0,1,2])
  # Require that the constructor also accept tuples of coefficients
  coeffs = (0, 0, 0, 1, 2)
  assert(Polynomial(coeffs) == Polynomial([1,2]))

def testPolynomialMath():
  p1 = Polynomial([2, -3, 5])  # 2x**2 -3x + 5

  # p.scaled(scale) returns a new polynomial with all the
  # coefficients multiplied by the given scale
  p2 = p1.scaled(10) # 20x**2 - 30x + 50
  assert(isinstance(p2, Polynomial))
  assert(p2.evalAt(0) == 50)
  assert(p2.evalAt(2) == 70)

  # p.derivative() will return a new polynomial that is the derivative
  # of the original, using the power rule:
  # More info: https://www.mathsisfun.com/calculus/power-rule.html
  p3 = p1.derivative() # 4x - 3
  assert(type(p3) == Polynomial)
  assert(str(p3) == "Polynomial(coeffs=[4, -3])")
  assert(p3.evalAt(0) == -3)
  assert(p3.evalAt(2) == 5)

  # we can add polynomials together, which will add the coefficients
  # of any terms with the same degree, and return a new polynomial
  p4 = p1.addPolynomial(p3) # (2x**2 -3x + 5) + (4x - 3) == (2x**2 + x + 2)
  assert(type(p4) == Polynomial)
  assert(str(p4) == "Polynomial(coeffs=[2, 1, 2])")
  assert(p1 == Polynomial([2, -3, 5]))
  assert(p4.evalAt(2) == 12)
  assert(p4.evalAt(5) == 57)
  # can't add a string and a polynomial!
  assert(p1.addPolynomial("woo") == None)

  # lastly, we can multiple polynomials together, which will multiply the
  # coefficients of two polynomials and return a new polynomial with the
  # correct coefficients.
  # More info: https://www.mathsisfun.com/algebra/polynomials-multiplying.html

  p1 = Polynomial([1, 3])
  p2 = Polynomial([1, -3])
  p3 = Polynomial([1, 0, -9])
  assert(p1.multiplyPolynomial(p2) == p3) # (x + 3)(x - 3) == (x**2 - 9)
  assert(p1 == Polynomial([1, 3]))
  # (x**2 + 2)(x**4 + 3x**2) == (x**6 + 5x**4 + 6x**2)
  p1 = Polynomial([1,0,2])
  p2 = Polynomial([1,0,3,0,0])
  p3 = Polynomial([1,0,5,0,6,0,0])
  assert(p1.multiplyPolynomial(p2) == p3)

def testPolynomialClass():
  print('Testing Polynomial class...', end='')
  testPolynomialBasics()
  testPolynomialEq()
  testPolynomialConstructor()
  testPolynomialMath()
  print('Passed!')
testPolynomialClass()
```

(b) [10 points] The Quadratic class should inherit from the Polynomial class and should pass the following testcases.

```
import math
def testQuadraticClass():

  print("Testing Quadratic class...", end="")
  # Quadratic should inherit properly from Polynomial
  q1 = Quadratic([3,2,1])  # 3x^2 + 2x + 1
  assert(type(q1) == Quadratic)
  assert(isinstance(q1, Quadratic) and isinstance(q1, Polynomial))
  assert(q1.evalAt(10) == 321)
  assert(str(q1) == "Quadratic(a=3, b=2, c=1)")

  # We use the quadratic formula to find the function's roots.
  # More info: https://www.mathsisfun.com/quadratic-equation-solver.html

  # the discriminant is b**2 - 4ac
  assert(q1.discriminant() == -8)
  # use the discriminant to determine how many real roots (zeroes) exist
  assert(q1.numberOfRealRoots() == 0)
  assert(q1.getRealRoots() == [ ])

  # Once again, with a double root
  q2 = Quadratic([1,-6,9])
  assert(q2.discriminant() == 0)
  assert(q2.numberOfRealRoots() == 1)
  [root] = q2.getRealRoots()
  assert(math.isclose(root, 3))
  assert(str(q2) == "Quadratic(a=1, b=-6, c=9)")
```

```
    # And again with two roots
    q3 = Quadratic([1,1,-6])
    assert(q3.discriminant() == 25)
    assert(q3.numberOfRealRoots() == 2)
    [root1, root2] = q3.getRealRoots() # smaller one first
    assert(math.isclose(root1, -3) and math.isclose(root2, 2))

    # Creating a non-quadratic "Quadratic" is an error
    ok = False # the exception turns this to True!
    try: q = Quadratic([1,2,3,4]) # this is cubic, should fail!
    except: ok = True
    assert(ok)
    # one more time, with a line, which is sub-quadratic, so also fails
    ok = False
    try: q = Quadratic([2,3])
    except: ok = True
    assert(ok)

    # And make sure that these methods were defined in the Quadratic class
    # and not in the Polynomial class (we'll just check a couple of them...)
    assert('evalAt' in Polynomial.__dict__)
    assert('evalAt' not in Quadratic.__dict__)
    assert('discriminant' in Quadratic.__dict__)
    assert('discriminant' not in Polynomial.__dict__)
    print("Passed!")

testQuadraticClass()
```