CS 1358 Introduction to Programming in Python

Fall Semester 2019

Prof. Pai H. Chou

Self-Check 11

Due Date: Sunday, November 25, 2019, 11:59pm

Answer the following questions to check your understanding of your material. Expect the same kind of questions to show up on your tests.

## 1. Definitions and Short Answers - functions

1. In Python, ArithmeticError is a **base class** of FloatingPointError, OverflowError, and ZeroDivisionError. So,
   1. Does ArithmeticError **inherit from** FloatingPointError ? Or does FloatingPointError inherit from ArithmeticError?
   2. Does FloatingPointError inherit from OverflowError ? Or ZeroDivisionError? Or is there any **inheritance relationship** between them?
   3. Is ZeroDivisionError a **superclass** of ArithmeticError? Or the other way around? [subclass]
2. Let a, f, o, and z respectively denote an instance of ArithmeticError, FloatingPointError, OverflowError, and ZeroDivisionError.
   1. Is a an **instance of** FloatingPointError?
   2. Is f an instance of ArithmeticError?
   3. Does z **inherit from** ZeroDivisionError? Or what is the correct word for the relationship? [is an instance of]
   4. Does o inherit from ArithmeticError?
3. Which of the following evaluates to True?
   1. isinstance(f, ArithmeticError)
   2. isinstance(z, FloatingPointError)
   3. isinstance(a, ZeroDivisionError)
   4. issubclass(OverflowError, ArithmeticError)
   5. issubclass(FloatingPointError, ZeroDivisionError)
   6. issubclass(ArithmeticError, ZeroDivisionError)
4. Suppose you want to define a class named MyList by subclassing from the built-in list class.   
    1 class MyList(list):  
    2 def \_\_repr\_\_(self):  
    3 return self.\_\_class\_\_.\_\_name\_\_ + '(' + \_\_\_\_\_ + ')'  
   1. This class does not define the \_\_init\_\_() method. Does this mean you can't call MyList as a constructor? If you can call MyList as a constructor, what method is actually called? [you can call MyList as a constructor. the arguments are the same as those you pass to a list() constructor. It inherits the constructor from its superclass, namely list's \_\_init\_\_() method.]
   2. By defining the \_\_repr\_\_() method in MyList class, what happens to the base class's (i.e., list class's) \_\_repr\_\_() method? Is it replaced? Or does it continue to exist? [both \_\_repr\_\_() exist. It is just shadowed while inside MyList. That is, MyList's \_\_repr\_\_ will be found first and used, but list's \_\_repr\_\_ continues to exist, unaffected and unmodified.]
   3. How does MyList's \_\_repr\_\_() method invoke its superclass list's \_\_repr\_\_() method to render the actual list content as constructor argument? [super().\_\_repr\_\_()]
5. If you define a find() method in MyList class,
   1. does it need to call the list's find() method? [no]
   2. Does it automatically call list's find() method? [no]
      1. If so, is list's find() method automatically called before or after MyList's find() method? [n/a]
      2. If not, how can MyList's find() call its base class's find()? [super().find(val)]
6. MyList's sort() method is defined as follows:  
    1 class MyList(list):  
    2 # \_\_repr\_\_() and find() not shown  
    3 def sort(self):  
    4 D ={'NoneType': 0, 'int': 1, 'float': 1, 'str': 2,  
    5 'tuple': 3, 'list': 4}  
    6 return super().sort(key=lambda x: \  
    7 (D.get(type(x).\_\_name\_\_, 5), x)),  
    8 reverse=reverse)  
    9 # additional definition not shown  
   1. What does type(x).\_\_name\_\_ do? What is the value of
      1. type(3).\_\_name\_\_?
      2. type((12, 34)).\_\_name\_\_?
      3. type('hello').\_\_name\_\_?
      4. type({}).\_\_name\_\_?
   2. Given the value of D defined on lines 4-5, what does D.get(k, v) do and how is it different from D[k]? What is the value of
      1. D.get(type(3).\_\_name\_\_, 5)
      2. D.get(type((12, 34)).\_\_name\_\_, 5)
      3. D.get(type('hello').\_\_name\_\_, 5)
      4. D.get(type([]).\_\_name\_\_, 5)
      5. D.get(type(3+2j).\_\_name\_\_, 5)
      6. D.get(type({}).\_\_name\_\_, 5)
   3. Let  
      k = lambda x:(D.get(type(x).\_\_name\_\_, 5),x)   
      L = [7.4, 2, 'world', 'bye', (13, 24), (14, 28), None]  
      , then what is the value of  
      list(map(k, L))  
      ?  
      ans: [(1, 7.4), (1, 2), (2, 'world'), (2, 'bye'), (3, (13, 24)), (3, (14, 28)), (0, None)]  
        
      and what is the result of   
      list(sorted(map(k, L)))  
      ?  
      ans: [(0, None), (1, 2), (1, 7.4), (2, 'bye'), (2, 'world'), (3, (13, 24)), (3, (14, 28))]
7. In the revised version of MyList class's sort method,   
    1 class MyList(list):  
    2 # \_\_repr\_\_() and find() not shown  
    3 def sort(self, key=None, reverse=False):  
    4 D ={'NoneType': 0, 'int': 1, 'float': 1, 'str': 2,  
    5 'tuple': 3, 'list': 4}  
    6 return super().sort(key=lambda x: \  
    7 (D.get(type(x).\_\_name\_\_, 5), \  
    8 key(x) if key is not None else x),\  
    9 reverse=reverse)  
   10 # additional definition not shown  
   1. What is the meaning of line 9, reverse=reverse in this context?  
      [ans: this is to pass a parameter by name. The callee list's parameter named reverse (left hand side of the equal sign) gets the value of the expression reverse, which is the caller's (i.e., MyList's sort()'s) parameter named reverse.
   2. If the caller does not pass a key parameter on line 3, then what is the value of the expression  
      lambda x: key(x) if key is not None else x  
      ? [lambda x: x]
   3. if the caller passes a callable object to the key parameter on line 3, then what is the value of the expression  
      lambda x: key(x) if key is not None else x  
      ? [lambda x: key(x)]
8. In the ColorPoint class:  
     
    1 class Point:  
    2 def \_\_init\_\_(self, x, y):  
    3 self.\_x = x  
    4 self.\_y = y  
    5 def \_\_repr\_\_(self):  
    6 return \_\_class\_\_.\_\_name\_\_ + \  
    7 repr((self.\_x, self.\_y))  
    8 class ColorPoint(Point):  
    9 def \_\_init\_\_(self, x, y, color):  
   10 super().\_\_init\_\_(x, y)  
   11 self.\_color = color  
   12 def \_\_repr\_\_(self):  
   13 return \_\_class\_\_.\_\_name\_\_ +\  
   14 repr((self.\_x, self.\_y, self.\_color))  
   1. What is the purpose of line 10? [calls the base class's constructor on the object being constructed to initialize the base class's attributes]
   2. After line 10, what attributes are defined in self? [self.\_x and self.\_y]
   3. If ColorPoint class doesn't define its own \_\_repr\_\_ but instead chooses to inherit it, what will be printed on the line below?  
        
      >>> p = Point(2, 3)  
      >>> p  
      Point(2, 3)  
      >>> q = ColorPoint(4, 5, 'black')  
      >>> q  
      \_\_\_\_\_\_\_\_\_\_\_\_  
      Does it print Point(4, 5)? ColorPoint(4, 5)? or something else?
9. What is the meaning of **polymorphism** in a programming language like Python? Does it mean *an object can take on different names*? Or does it mean *a name can refer to one of different possible objects*?
10. Why is the built-in str() considered an **overloaded function**?
11. What is the meaning of **operator overloading**?
12. To overload operators +, -, \*, / for the Point class above, what do you have to declare?  
    [def \_\_add\_\_(self, B); def \_\_sub\_\_(self, B); def \_\_mul\_\_(self, B); def \_\_div\_\_(self, B)]
13. Assume x = 3, what are the values of the following expressions, if valid? If not valid, why not, and how can it be fixed?
    1. x.\_\_add\_\_(2)
    2. 2.\_\_add\_\_(x)
    3. x.\_\_add\_\_(2.)
    4. 2..\_\_add\_\_(x)
14. In the Vector class,  
     1 import operator as op   
     2 class Vector:   
     3 def \_\_init\_\_(self, \*v):   
     4 self.\_v = list(v) # covert tuple to list   
     5 def \_\_repr\_\_(self):   
     6 return \_\_class\_\_.\_\_name\_\_+repr(tuple(self.\_v))   
     7 def \_\_add\_\_(self, right):   
     8 return Vector(\*map(op.add, self.\_v, right.\_v))    
     # op.add is same as lambda x,y: x+y   
     9 def \_\_sub\_\_(self, right):   
    10 return Vector(\*map(op.sub, self.\_v, right.\_v))  
    11 x = Vector(1, 2, 3)  
    12 y = Vector(4, 5, 6)  
    13 z = x + y  
    14 i = id(x)  
    15 x += y  
    16 j = id(x)  
    17 print(i == j)  
    1. When x = Vector(1, 2, 3) is called, what is the value of parameter v on line 3?
    2. What is the equivalent method syntax when line 13 z = x + y is executed? In other words, the statement can be written in the form of  
       z = *object*.*method*(*arg*)  
       What are the *object*, *method*, and *arg*? [z = x.\_\_add\_\_(y)]
    3. By the time line 13 z = x + y finishes execution, how many times has the Vector constructor been called?
    4. Does line 17 print True or False? Explain.
15. In the previous problem, Python understands how to execute line 15  
    15 x += y   
    as  
    x = x.\_\_add\_\_(y)   
    So why would you ever need to overload the \_\_iadd\_\_(self, other) method? Isn't it redundant?
16. Assume x and y refer to the two Vector instances. In order to support the following operator syntax, what special methods must be defined in the Vector class, and what is the equivalent *object*.*method*(*args*) syntax? Fill in the last column of the table below.

| operator syntax | example | equivalent object.method(args) syntax |
| --- | --- | --- |
| indexing | x[3] |  |
| slicing | x[2:5] |  |
| indexed assignment | x[1] = 5 |  |
| sliced assignment | x[0:2] = (8, 6) |  |

1. Given that binary operators << and >> have **lower precedence** than binary operators + and -, and all are **left associative**, in what order does Python evaluate the expression  
   m + n << p - q >> r  
   ? is it
   1. ((m + n) << (p - q)) >> r
   2. (m + (n << p)) - (q >> r)
   3. (m + n) << ((p - q) >> r)
   4. m + ((n << p) - (q >> r))

or some other way?

1. Both str() and repr() return a string of an object. What is their difference? Suppose you have x = 'hello\n', what is the value of
   1. list(str(x))
   2. list(repr(x))

?

1. If you overload the \_\_len\_\_() special method in your Vector class, how does Python expect the user to call it on an instance v? (Hint: not v.\_\_len\_\_())

## 2. Programming

1. (Difficulty: ★★☆☆☆) Extend the Polynomial class from last week. To recall, it models a polynomial for a single variable *x* with integer coefficients and powers. That is,  
   *f*(*x*) = *a*0 + *a*1 *x* + *a*2 *x*2 + *a*3 *x*3 + *a*4 *x*4 + ...  
     
   The constructor takes variable-length arguments for the coefficients for polynomials to the 0, 1, 2, … degrees.

The supported operations include

* adding or subtracting two polynomial functions to make another polynomial function by overloading the + and - operators. (i.e., define \_\_add\_\_ and \_\_sub\_\_ special methods)
* evaluating a polynomial function for a given value of x
* scaling a polynomial by implementing the \_\_imul\_\_ special method

>>> f = Polynomial(3, 2, 0, 5, 4)  
>>> g = Polynomial(7, 4, 1)  
>>> f + g  
Polynomial(10, 6, 1, 5, 4)  
>>> g - f  
Polynomial(4, 2, 1, -5, -4)  
>>> f \*= 2  
Polynomial(6, 4, 0, 10, 8)  
>>> f(-1)  
-28  
>>>

1. (Difficulty: ★★★☆☆) Define a NewTemp class by subclassing from the Temperature class from last week so that it can support
   1. operator overloading for + and -. The unit of the operation defaults to the unit of the left-hand-side.
   2. changing units, including 'C' (Celsius), 'F' (Fahrenheit)

Note: define \_\_add\_\_(self, RHS) and \_\_sub\_\_(self, RHS) methods to overload the + and - operators. You must check the RHS (= "right hand side") parameter's type to make sure it is an instance of Temperature (base class is okay -- doesn't have to be NewTemp), or it could be a number (int or float). If it is a Temperature, convert it to the same unit as self's unit before adding or subtracting. If it is a number (int or float), simply assume it is of the same unit.  
  
>>> t = NewTemp(20, 'C')  
>>> t + 3  
NewTemp(23, 'C')  
>>> u = NewTemp(30, 'C')  
>>> t + u  
NewTemp(50, 'C')  
>>> t - u  
NewTemp(-20, 'C')  
>>> t.unit  
'C'  
>>> t.unit = 'F'  
>>> t  
NewTemp(68.0, 'F')  
>>> t + u  
NewTemp(122.0, 'F')

1. (Difficulty: ★★★★☆) Write a NewList class by inheriting from the built-in list class to support the following operations:
   1. list multiplication (also known as cross-product) by overloading the @ operator (define the \_\_matmul\_\_(self, RHS) special method)  
      >>> NewList([6,7,8]) @ NewList(['a', 'b'])  
      NewList([(6, 'a'), (6, 'b'), (7, 'a'), (7, 'b'), (8, 'a'), (8, 'b')])
   2. scalar multiplication, to be distinguished from list repetition. e.g.,   
      >>> NewList([6, 7, 8]) \* 2   
      NewList([6, 7, 8, 6, 7, 8])   
      as in a regular list, but  
      >>> 2 \* NewList([6, 7, 8])   
       NewList([12, 14, 16])  
      >>> 3 \* NewList(['a', 'b', 'c'])  
      NewList(['aaa', 'bbb', 'ccc'])  
      Hint: define the \_\_rmul\_\_(self, scalar) special method, where scalar is a number.
   3. alternative base index (e.g., starting from index 1 instead of index 0). However, negative index remains the same.  
      >>> L = NewList(['a', 'b', 'c', 'd', 'e'])  
      >>> L[1]  
      'a'  
      >>> L[5]  
      'e'
   4. inclusive limit instead of exclusive (e.g., L[2:5] refers to L[2],..., **up to and including** L[5], whereas a regular list is up to **but not including** L[5]). This should work for downward (e.g., negative) stepping and slicing.  
      >>> L[2:3]  
      NewList(['b', 'c'])  
      >>> L[4:2:-1]  
      NewList(['d', 'c', 'b'])  
      >>> L[-1]  
      'e'

Hint: to implement c and d, you will need to overload all operators that may use indexing or slicing. This means

\_\_getitem\_\_(self, itemref) -- called by L[i], L[i:j] or L[i:j:k],

\_\_setitem\_\_(self, itemref, val) -- called to do L[i] = val, L[i:j] = val, L[i:j:k]=val

\_\_delitem\_\_(self, itemref) -- called to do del L[i], del L[i:j], or del L[i:j:k]