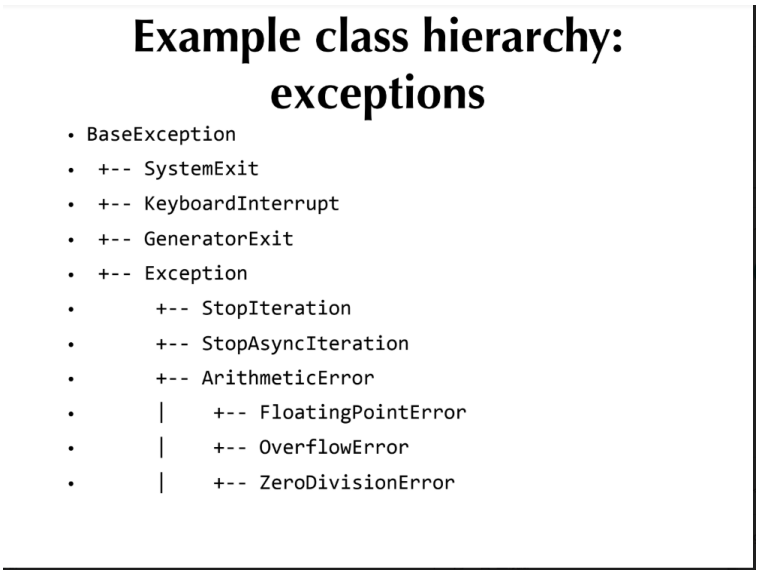
EECS 1210 Introduction to Programming in Python
Sping Semester 2024
Prof. Pai H. Chou

Self-Check 11

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,
   a. Does ArithmeticError **inherit from** FloatingPointError ? Or does FloatingPointError inherit from ArithmeticError?
   b. Does FloatingPointError inherit from OverflowError ? Or ZeroDivisionError? Or is there any **inheritance relationship** between them?
   c. 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.
   a. Is a an **instance of** FloatingPointError? no
   b. Is f an instance of ArithmeticError? yes
   c. Does z **inherit from** ZeroDivisionError? Or what is the correct word for the relationship? [is an instance of]
   d. Does o inherit from ArithmeticError? [is an instance of]

3. Which of the following evaluates to True?
    a. isinstance(f, ArithmeticError) True
    b. isinstance(z, FloatingPointError) False
    c. isinstance(a, ZeroDivisionError) False
    d. issubclass(OverflowError, ArithmeticError)True
    e. issubclass(FloatingPointError, ZeroDivisionError) False
    f. issubclass(ArithmeticError, ZeroDivisionError) False

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__ + '(' + _____ + ')'
```

    a. This class does not define the __init__() method. Does this mean you can't call MyList a s 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 li st() constructor. It inherits the constructor from its superclass, namely list's __init__() met hod.]
    b. By defining the __repr__() method in MyList class, what happens to the base class's (i.e., l ist 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.]
    c. How does MyList's __repr__() method invoke its superclass list's __repr__() method to re nder the actual list content as constructor argument? [super().__repr__()]

5. If you define a find() method in MyList class,
    a. does it need to call the list's find() method? [no]
    b. Does it automatically call list's find() method? [no]
        i. If so, is list's find() method automatically called before or after MyList's find() me thod? [n/a]
        ii. 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
```

a. What does type(x).__name__ do? What is the value of
   i. type(3).__name__? int
   ii. type((12, 34)).__name__? tuple
   iii. type('hello').__name__?str
   iv. type({}).__name__?dict

b. 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
   i. D.get(type(3).__name__, 5)1
   ii. D.get(type((12, 34)).__name__, 5)3
   iii. D.get(type('hello').__name__, 5)2
   iv. D.get(type([]).__name__, 5)4
   v. D.get(type(3+2j).__name__, 5)5
   vi. D.get(type({}).__name__, 5)5

c. 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
```

a. 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 h and 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.]

b. If the caller does not pass a key parameter on line 3, then what is the value of the expressi on
lambda x: key(x) if key is not None else x
? [lambda x: x]

c. 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))
```

a. What is the purpose of line 10? [calls the base class's constructor on the object being cons tructed to initialize the base class's attributes]

b. After line 10, what attributes are defined in self? [self._x and self._y]

c. 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?
Point(4, 5)

9. What is the meaning of **polymorphism** in a programming language like Python? Does it mean *an o bject can take on different names*? Or does it mean *a name can refer to one of different possible o bjects*?後者

10. Why is the built-in str() considered an **overloaded function**?
11. What is the meaning of **operator overloading**? 例如數字+跟字串+
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?
    a. x.__add__(2)valid
    b. 2.__add__(x) 2加括號才行
    c. x.__add__(2.) float(x).__add__(2.)
    d. 2..__add__(x)valid
14. In the Vector class,

```
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)
```

    a. When x = Vector(1, 2, 3) is called, what is the value of parameter v on line 3?(1,2,3)
    b. 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)]
    c. By the time line 13 z = x + y finishes execution, how many times has the Vector constructor been called? 1次
    d. Does line 17 print True or False? Explain.
       false 是新的instance了
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?

不會新創一個物件,x重覆使用

16. Assume x and y refer to the two Vector instances. In order to support the following operator synta x, what special methods must be defined in the Vector class, and what is the equivalent *object.met hod*(*args*) syntax? Fill in the last column of the table below.

operator syntax	example	equivalent object.method(args) synta x
indexing	x[3]	x.__getitem_(3)
slicing	x[2:5]	x.__getitem__(slice(2,5))
indexed assignment	x[1] = 5	x.__setitem_(1,5)
sliced assignment	x[0:2] = (8, 6)	x.__setitem__(slice(0,2),(8,6))

17. Given that binary operators << and >> have **lower precedence** than binary operators + and -, and al l are **left associative**, in what order does Python evaluate the expression

m + n << p - q >> r

? is it

a. ((m + n) << (p - q)) >> r
b. (m + (n << p)) - (q >> r)
c. (m + n) << ((p - q) >> r)
d. m + ((n << p) - (q >> r))

or some other way?

18. 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

a. list(str(x))
b. list(repr(x))

?

```
>>> x = 'hello\n'
>>> list(str(x))
['h', 'e', 'l', 'l', 'o', '\n']
>>> list(repr(x))
["'", 'h', 'e', 'l', 'l', 'o', '\\', 'n', "'"]
>>>
```

19. If you overload the __len__() special method in your Vector class, how does Python expect the use r to call it on an instance v? (Hint: not v.__len__())

len(v)

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
>>>
```

2. (Difficulty: ★★★☆☆) Define a NewTemp class by subclassing from the Temperature class from last week so that it can support
   a. operator overloading for + and -. The unit of the operation defaults to the unit of the left-hand-side.
   b. 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')
```

3. (Difficulty: ★★★★☆) Write a NewList class by inheriting from the built-in list class to support the following operations:
   a. list multiplication (also known as cross-product) by overloading the @ operator (define th e `__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')])
      ```

   b. 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.
   c. alternative base index (e.g., starting from index 1 instead of index 0). However, negative i ndex remains the same.

      ```
      >>> L = NewList(['a', 'b', 'c', 'd', 'e'])
      >>> L[1]
      'a'
      >>> L[5]
      'e'
      ```

   d. 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]

4.