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

Self-Check 10

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, suppose you have the code
   `return [13, 25, 'hello', 'z']`
   Which of the following are objects?
   a. return
   b. 13
   c. 25
   d. 'hello'
   e. 'z'
   f. [13, 25, 'hello', 'z']
   g. ,

2. In Python, do the following keywords or built-in identifiers refer to objects?
   a. if
   b. print
   c. len
   d. str
   e. ==

3. How can you make a clone of an object?
4. What is a **class**? How is a class related to an **instance**?
5. What is the term for a function call whose name is the name of a class?
6. How is a **method** different from a function?
   methods are invoked on an object in the form of object.method(), whereas a function is not invoked on an object.
7. When you do
   `import os`
   `L = os.listdir()`
   are you making a **method call** with os.listdir(), or are you making a **function call**? Why?

it is a function call, because os is imported to retain its name space, but listdir() is still a function. This i s evidenced by the fact you can do from os import lisdir, then you can invoke the same without the os. prefix.

8. In Python, suppose you havea class defined as

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move_by(self, dx, dy):
        self.x += dx
        self.y += dy
```

   a. How do you **instantiate** a point with a coordinate of (2, 3) and assign it to the variable p? p = Point(2, 3)
   b. What are the two **attributes** created by the constructor of this point? p.x, p.y
   c. The move_by() method defines three parameters (self, dx, dy) but the call takes only two a rguments, such as p.move_by(-2, 7). Why?
      because the method call syntax p.move_by(-2, 7) actually passes the object p to the first param eter as self. It is like Point.move_by(p, -2, 7)
   d. Is it ever okay to declare an instance method **without any parameter**, such as

```
def sayhi():
    print("I am a point")
```

      ?
      no, all instance methods require at least self as the first parameter.

9. How would you define the __repr__ method for the Point class above? What should it display if p = Point(2, 3) and you type p at the interactive prompt?

```
def _repr_(self):
    return self.__class__.__name__ + \
        repr((self.x, self.y))
# But there many other answers
```

10. Suppose in your class definition,

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5      def move_by(self, dx, dy):
6          self.x += dx
7          self.y += dy
8      m = move_by
9      @property
10     def area_of_box(self)
11         return self.x * self.y
```

a. What is the effect of line 8?
   m becomes an alias for move_by inside this class. This means you can call p.m(2, 3) and it is a nother way of making the call p.move_by(2, 3).
b. What type of construct is the @property on line 9? a decorator
c. With line 9, how should you invoke the code for area_of_box on a Point object p?
   you should just invoke it as an attribute, like p.area_of_box without the () method-call syntax.
d. What is the purpose of applying the @property decorator here?
   it makes a read-only derived attribute so you can write code like
   p.area_of_box when reading but can't write to it directly by p.area_of_box = 3

11. Suppose you have

```
>>> p = Point(2, 3)
>>> q = Point(4, 5)
>>> p.z = 7
```

a. What happens when you try to read the value of q.z?
b. if you set q.z = 10, what happens to the value of p.z?
c. Is it okay if you do Point.count = 0 next? If so, what kind of attribute is it called? class att ribute
d. Assuming the assignment Point.count = 0 is allowed, what is the value of p.count? Is it de fined?
e. What is the value of dir(p)? Does it include 'count' as a key in this dict? What about dir (q)?

12. If you want your **constructor** to increment a **class attribute** to count the number of instances created so far,
    a. How should you initialize the **class attribute** count = 0?
       class Point:
           count = 0
           ....
    b. How should you **increment** the class attribute count **in the constructor**? As self.count += 1 or as Point.count += 1? Why?
       Point.count += 1
       because if you do self.count += 1, it is really the same as self.count = self.count + 1, but the ri ght-hand-side of the assignment finds the class attribute, while the left-hand-side always goes i nto the object's own space, so it self.count will be newly created with the value of Point.count + 1 , whereas Point.count is not modified.

13. Why is it better to define setter/getter methods than allowing user code to modify the attributes dire ctly? For instance, suppose you have a DateTime class that allows you to do

```
>>> dt = DateTime(year=2019, month=11, day=11, hour=9, \
...               minute=8, second=7)
>>> dt.set_year(2023)
```

Why would it be preferred, compared to

```
>>> dt.year = 2023
```

?

calling a setter allows the value to be checked before actually assigning to an attribute; e.g., setting a m onth allows the checking if the month is in 1..12. A getter allows some attributes to be calculated from other attributes instead of maintaining each one, and the user shouldn't need to worry about the implem entation.

14. In DateTime class shown on slide #35, the attributes are named with an underscore in front, such a s _year, _month, _day, etc. What is the reason for this?
This is a naming convention saying these are private attributes and that they should not be accessed dire ctly by anyone outside the class.

15. In the DateTime class, the check_and_set() method is defined to be

```
def check_and_set(self, field_name, field_value, L, U):
    if not (L <= field_value <= U):
        raise ValueError(f'{field_name} must be {L}..{U}')
    self.__dict__['_'+field_name] = field_value
```

a. What is the purpose of self.__dict__['_'+field_name] = field_value? If field_name is 'year', self is p, and field_value is 2010, then what attribute of p gets assigned the value 2010?
it assigns field_value to an attribute whose name is field_name prepended with _. For instance, if field_name is "year", then calling p.check_and_set('year', 2010, 0, 3000) will do p._year = 2 010 after checking that 2010 is indeed between 0 and 3000.

b. Why is it a good idea to write check_and_set() as a method that is called by set_year(), set _month(), set_day(), etc methods to assign value via self.__dict__[] instead of assigning to the attributes self._year, self._month, self._day directly?
this is called "code factoring", which means you don't want to keep multiple copies of the sam e code if you can say it just once. This way, it could make your code shorter, but more importa ntly, if you have to make changes, you just have to fix it once instead of fixing many places.

16. Assume you have a **getter** and a **setter** for the instance attribute _month in the DateTime class:

```
1  class DateTime:
2      def get_month(self):
3          ...
4      def set_month(self, mo):
5          ...
6      month = property(lambda self: self.get_month(),
7                       lambda self, v: self.set_month(v))
```

What is the effect of lines 6-7? Suppose you have a variable dt which is an instance of DateTime, what **method gets called** when you do

```
print(dt.month)
```

and

```
dt.month = 5
```

?

it defines a property named month, which calls the setter and getter function depending on if it is assign ed to or read from. In case of print(dt.month), it calls dt.get_month() to get the month value; in case of dt.month=5, it calls the dt.set_month(5) method to do the setting.

17. Which of the following correctly describes a **class method**? Assume you want to declare one named set_year_range() for the DateTime class and it takes parameters for the lower and upper bounds.
    a. you need to use the **decorator** @classmethod on the line immediately before def set_year_range() method definition to make it a class method
       correct, the @classmethod is required
    b. As long as you have the @classmethod decorator, the class method works just like an instance method because you would declare it as
       def set_year_range(self, lower, upper):
       and self refers to the **instance** that you invoke the method on.
       no, you should also change self to cls so it refers to the class of the class method
    c. In addition to @classmethod decorator, you also need to declare the **first parameter** as cls instead of self because it refers to the **class object** instead of the instance object
       yes, class method takes cls as first parameter.
    d. Even though you want cls to refer to the **class object**, you still invoke the class method on an **instance object** (e.g., named p)
       p.set_year_range(100, 3000)
       and Python will pass the class object for p as the cls parameter.
       yes, always invoke on the instance object
18. Consider the leap-year function

```
def leap(year):
    return (year % 400 == 0) or \
           ((year % 4 == 0) and (year % 100 != 0))
```

and you would like to define it as a method inside the class DateTime rather than as a function outside the class.
    a. define it as an **instance method**

```
def leap(self, year):
    # same code as the function
```

    b. define it as a **class method**

```
@classmethod
def leap(cls, year):
    # same code as the function
```

    c. define it as a **static method**

```
@staticmethod
def leap(year):
    # same code as the function
```

    d. Is there any difference in how you would call the leap method?
       no, they are called in exactly the same way, even though their parameter lists are defined differently
    e. Which of the three methods would be the preferred way and why?
       static method would be preferred in this case because it does not depend on any attributes in the instance object or the class object but the functionality is very relevant to the DateTime class. Defining it inside the DateTime class instead of as a function reduces "name pollution".

2. Programming

1. (Difficulty: ★★☆☆☆) Define a class for a polynomial for a single variable x with integer coeff icients and powers. That is,

   $f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + \ ...$

   Your Polynomial constructor would take variable-length arguments for the coefficients from the 0t h order and up. For instance,

   $f(x) = 3 + 5 x + 4 x^2 + 7 x^3 + x^4$

   is represented by

   `>>> f = Polynomial(3, 5, 4, 7, 1)`

   It should support a method named evaluate(xvalue):

   `>>> f.evaluate(3)`

   `324`

   because $3 + 5 * 3 + 4 * 3^2 + 7 * 3^3 + 3^4$

   $= 3 + 15 + 36 + 189 + 81$

   $= 324$

   Your Polynomial class may look like this:

```
class Polynomial:
    def __init__(self, *coeff):
        # your code here to remember to coefficients
        #
    def evaluate(self, xvalue):
        # return the sum of coefficient_i * xvalue ^ i
```

   Extra information:

   If you define a special method named `__call__`, then the object instance can be called just like a fu nction. To do this, you can simply do

```
    __call__ = evaluate
    # indent it at the same level as the def for the methods
```

   this way, you define `__call__` to be another name for the evaluate method, but because it is a speci al symbol, Python lets you say

   `>>> f(3)`

   `324`

   which is a more concise way than saying `f.evaluate(3)`.

ANS:

```
class Polynomial:
    coefficient = list()
    def __init__(self, *coeff):
        self.coefficient = coeff
        return
    def evaluate(self, xvalue):
        val = 0
        for index, value in enumerate(self.coefficient):
            val += value * (xvalue**index)
        return val
    __call__ = evaluate

f = Polynomial(3, 5, 4, 7, 1)

f(3)
```

2. (Difficulty: ★★★☆☆) Define a class for Temperature. The requirements are
   a. The constructor should take two arguments (degree, unit):
   b. degree is an int or float
      i. The constructor needs to check if degree is an int or float; if not, raise a TypeErro r.
   c. unit defaults to 'C' for Celsius, but it can be 'F' for Fahrenheit
      i. The constructor needs to check if the unit is an allowed character; if 9not, raise a ValueError. Actually, lower case 'c' and 'f' are also accepted, but they should be c onverted to the upper case.
   d. The __repr__() method should return a string that, when printed, is a constructor call that yields the same value as the object.
   e. Define an instance method named get_temp(). It should return a tuple (*degree*, *unit*). It t akes one optional argument for the unit, which should be either 'C' (default) or 'F', in the s ame way as the constructor.
   f. Define a **property** named degree. You should define two methods
      i. get_degree(), which returns the value of the _degree attribute
      ii. set_degree(), which assigns the parameter value to the _degree attribute
      iii. use degree = @property(...) to make degree a property
   g. Define a **class method** named set_format() that takes a formatting string to be used by the subsequent __repr__() calls to format the degree.

```
>>> c = Temperature(10)
>>> d = Temperature(68, 'F')
>>> c
'10.0 C'
>>> d
'68.0 F'
>>> c.get_temp()
```

```
(10, 'C')
>>> c.get_temp('F')
(50.0, 'F')
>>> d.get_temp()
(68, 'F')
>>> d.get_temp('C')
(20.0, 'C')
>>> c.degree
10
>>> c.degree = 50
>>> c.get_temp()
(50, 'C')
>>> c.get_temp('F')
(122.0, 'F')
>>> c
'50.0 C'
>>> c.set_format('%d')  # this is a class method call
>>> c
'50 C'
>>> c.set_format('%.3f')
>>> c
'50.000 C'
>>> d.set_format('%.3f')
>>> d
'68.000 F'
```

```python
class Temperature:
    def __init__(self, degree, unit='C'):
        if not type(degree) in {int, float}:
            raise TypeError('degree must be int or float')
        if not unit in {'C', 'F'}:
            raise ValueError("unit must be 'C' or 'F'")
        self._degree = degree
        self._unit = unit
        self._format = ".1f"
    def __repr__(self):
        return f"'{self._degree:{self._format}} {self._unit}'"
    def get_temp(self, unit=''):
        if unit == '':
            return (self._degree, self._unit)
        elif self._unit == 'F' and unit == 'C': # F to C
            return ((self._degree - 32) * 5 / 9, unit)
        elif self._unit == 'C' and unit == 'F': # C to F
            return ((self._degree * 9 / 5) + 32, unit)
    def get_degree(self):
        return self._degree
    def set_degree(self, value):
        self._degree = value
    def set_format(self,string_format):
        string_format = string_format.split("%")[1]
        self._format = string_format
        print(f"'{self._degree:{string_format}} {self._unit}'")
    degree = property(
        fget=get_degree,
        fset=set_degree
    )
```

3. (Difficulty: ★★★★☆) Write a Python program that models the mother-side relationship in a family.

```python
class Person:
    def __init__(self, name):
        # your code here
    def __repr__(self):
        # your code here

    @property
    def name(self):
        # your code here. read-only property

    @property
    def children(self):
        # your code here. read-only property

    def add_children(self, *children):
        # your code here.
```

```
        # construct each child, linked with mother and sisters

    @property
    def mother(self):
        # your code here. read-only property

    @property
    def sisters(self):
        # your code here. read-only property
        # mother's daughters minus self

    @property
    def aunts(self):
        # your code here: return list of aunts. read-only
        # mother's sisters

    @property
    def grandchildren(self):
        # your code here: list of ALL grandchildren. read-only
        # trick is how to combine lists from daughters'
        # daughters.

    @property
    def family_tree(self):
        # make a dictionary for the family tree
        # from self to descendants but not to ancestors
        # hint: recursion
        # read-only property.
```

```
>>> p = Person('Wilma')  # constructor call
>>> p.children              # no children initially
[]
>>> p.add_children('Mary', 'Ann', 'Jill', 'Jane')
>>> p.children           # husband & wife have same children
[Person('Mary'), Person('Ann'), Person('Jill'), Person('Jane')]
>>> mary, ann, jill, jane = p.children
>>> mary
Person('Mary')
>>> mary.mother
Person('Wilma')
>>> mary.sisters
[Person('Ann'), Person('Jill'), Person('Jane')]
```

```
>>> mary.children
[]
>>> mary.add_children('Lynn', 'Cindy')
>>> lynn, cindy = mary.children
>>> lynn.aunts
[Person('Ann'), Person('Jill'), Person('Jane')]
>>> lynn.grandmother
[Person('Wilma')]
>>> jill.add_children('Kate')
>>> p.family_tree
{'Wilma': {'Mary': {'Lynn': {}, 'Cindy': {}}, 'Ann': {}, 'Jill': {'Kate':{}}, 'Jane': {}}}
>>> p.grandchildren
[Person('Lynn'), Person('Cindy'), Person('Kate')]
>>>
```

```python
class Person:
    def __init__(self, name):
        # your code here
        self._name = name
        self._children = []
        self._mother = None
        self._sister = []
        self._aunts = []
        self._grandchildren = []
    def __repr__(self):
        # your code here
        return self.__class__.__name__ + f"('{self._name}')"
    @property
    def name(self):
        # your code here. read-only property
        return self._name
    @property
    def children(self):
        # your code here. read-only property
        return self._children
    def add_children(self, *children):
        # your code here.
        # construct each child, linked with mother and sisters
        self._children = [Person(i) for i in children]
        for i in self._children:
            i._mother = self
    @property
    def mother(self):
        # your code here. read-only property
        return self._mother

    @property
    def sisters(self):
        # your code here. read-only property
        # mother's daughters minus self
        for i in self._mother.children:
            if i != self:
                self._sister.append(i)
        return self._sister

    @property
    def aunts(self):
        # your code here: return list of aunts. read-only
        # mother's sisters
        return self._mother._sister

    @property
    def grandchildren(self):
        # your code here: list of ALL grandchildren. read-only
        # trick is how to combine lists from daughters'
        # daughters.
        for i in self._children:
            self._grandchildren.extend(i._children)
        return self._grandchildren

```

```python

    @property
    def family_tree(self):
        # make a dictionary for the family tree
        # from self to descendants but not to ancestors
        # hint: recursion
        # read-only property.
        return {self._name:self.find_tree()}

    def find_tree(self):
        ans = {}
        if len(self._children) == 0:
            return {}
        else:
            temp = {}
            for i in self._children:
                temp[i._name] = i.find_tree()
            return temp
```

(之後會再更新)