CS 1358 Introduction to Programming in Python

Fall Semester 2019

Prof. Pai H. Chou

Self-Check 10

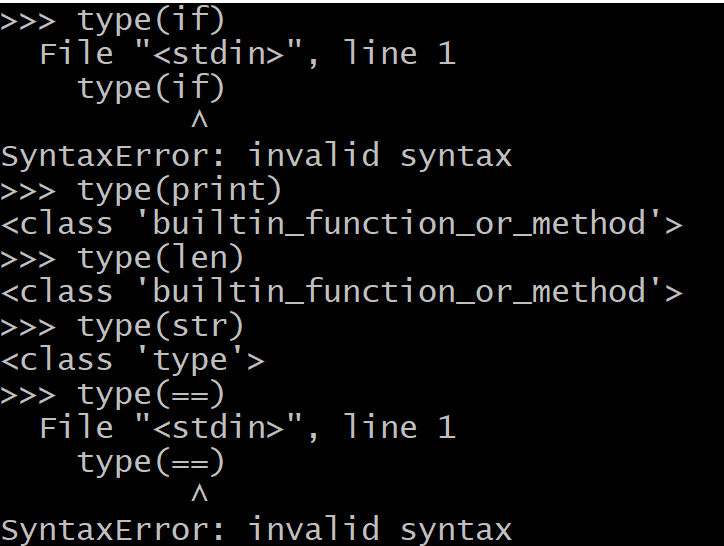
Due Date: Sunday, November 18, 2019, 11:59pm

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, suppose you have the code  
   return [13, 25, 'hello', 'z']  
   Which of the following are objects?
   1. return
   2. 13
   3. 25
   4. 'hello'
   5. 'z'
   6. [13, 25, 'hello', 'z']
   7. ,

-> b, c, d, e, f

1. In Python, do the following keywords or built-in identifiers refer to objects?
   1. if -> NO
   2. print -> NO
   3. len -> NO
   4. str -> YES
   5. == -> NO

1. How can you make a clone of an object?

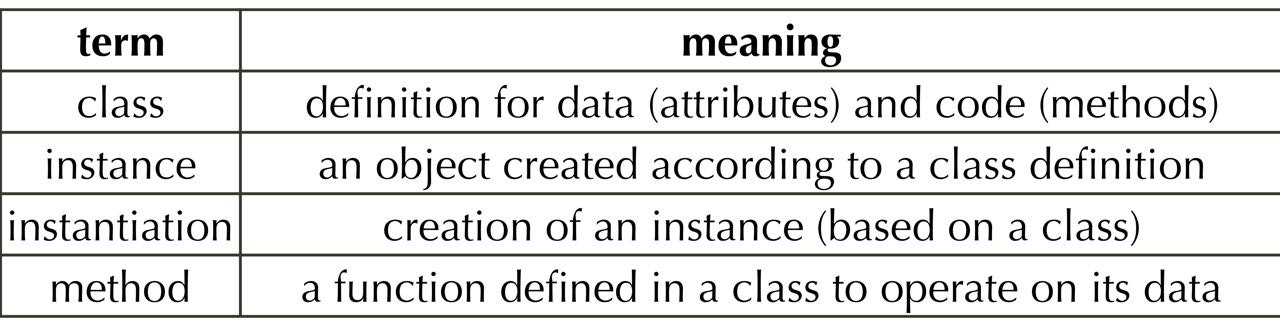
-> by copying (shallow copy / deep copy)

1. What is a **class**?

-> member data and code (member function) that operates on data

How is a class related to an **instance**?

-> object created according to class



1. What is the term for a function call whose name is the name of a class?

-> constructor call

1. How is a **method** different from a function?

-> A method refers to a function which is part of a class. You access it with an instance or object of the class. A function doesn’t have this restriction: it just refers to a standalone function. This means that all methods are functions, but not all functions are methods.

1. 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?

-> method call

1. In Python, suppose you have a 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  
   1. How do you **instantiate** a point with a coordinate of (2, 3) and assign it to the variable p?

-> Python calls Point.\_\_init\_\_(p, 2, 3) to initialize p

* 1. What are the two **attributes** created by the constructor of this point?

-> self.x

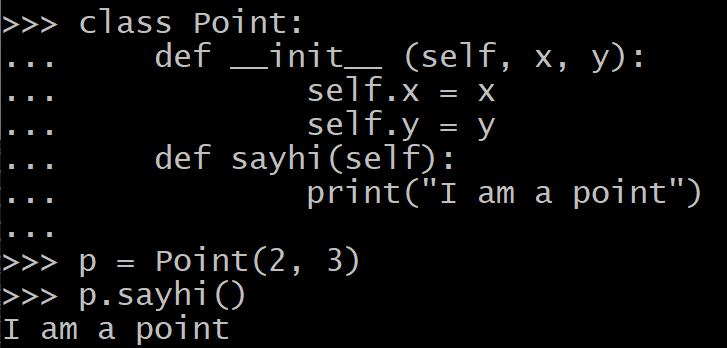
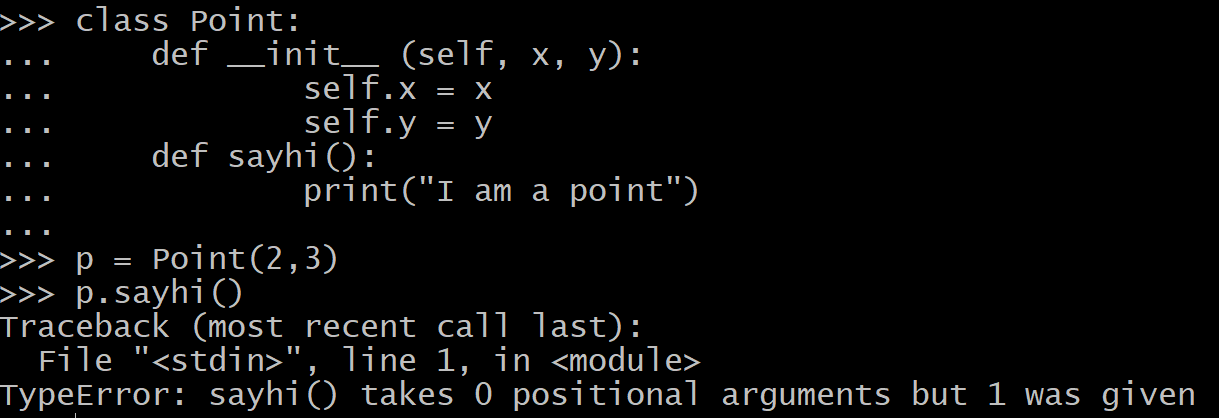
-> self.y

* 1. The move\_by() method defines three parameters (self, dx, dy) but the call takes only two arguments, such as p.move\_by(-2, 7). Why?

->

* 1. Is it ever okay to declare an instance method **without any parameter**, such as   
      def sayhi():  
      print("I am a point")  
     ?  
     -> No, Python requires the first parameter of method to be self

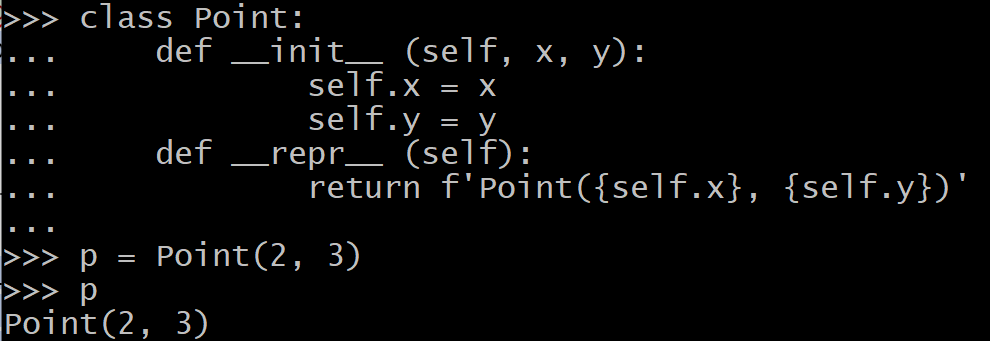
def sayhi(self):  
 print("I am a point")



1. How would you define the \_\_repr\_\_ method for the Point class above?

def \_\_repr\_\_(self):  
 return f'Point({self.x}, {self.y})'

What should it display if p = Point(2, 3) and you type p at the interactive prompt?  
-> Point(2, 3)



1. 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  
   1. What is the effect of line 8?

-> declares m to be an alias for move\_by

* 1. What type of construct is the @property on line 9?

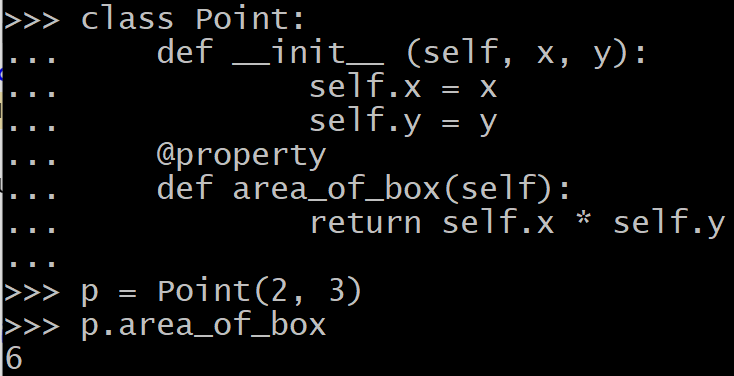
-> decorators

* 1. With line 9, how should you invoke the code for area\_of\_box on a Point object p?

-> p.area\_of\_box

* 1. What is the purpose of applying the @property decorator here?

-> To just name derived attributes without the calling () syntax



1. Suppose you have  
   >>> p = Point(2, 3)  
   >>> q = Point(4, 5)  
   >>> p.z = 7  
   1. What happens when you try to readf the value of q.z?

-> Error

* 1. if you set q.z = 10, what happens to the value of p.z?

-> Won’t change, still = 7

* 1. Is it okay if you do Point.count = 0 next? If so, what kind of attribute is it called?

-> class arrtibute

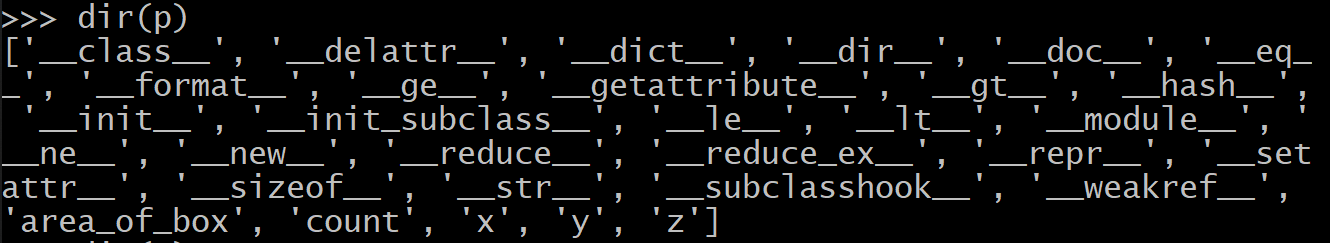
* 1. Assuming the assignment Point.count = 0 is allowed, what is the value of p.count?

-> 0

Is it defined?

-> Yes

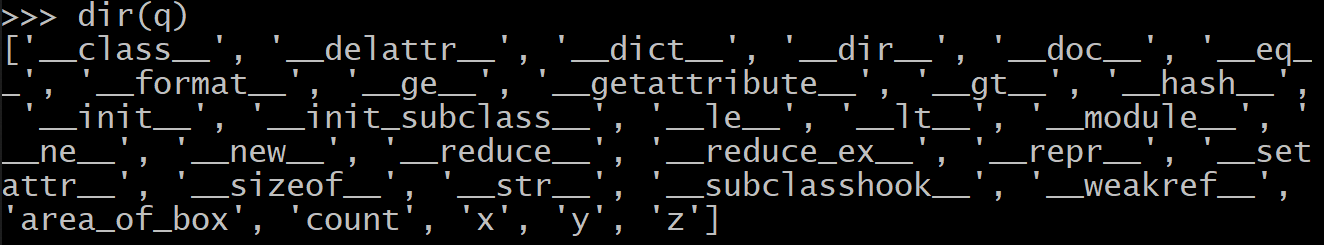
* 1. What is the value of dir(p)?

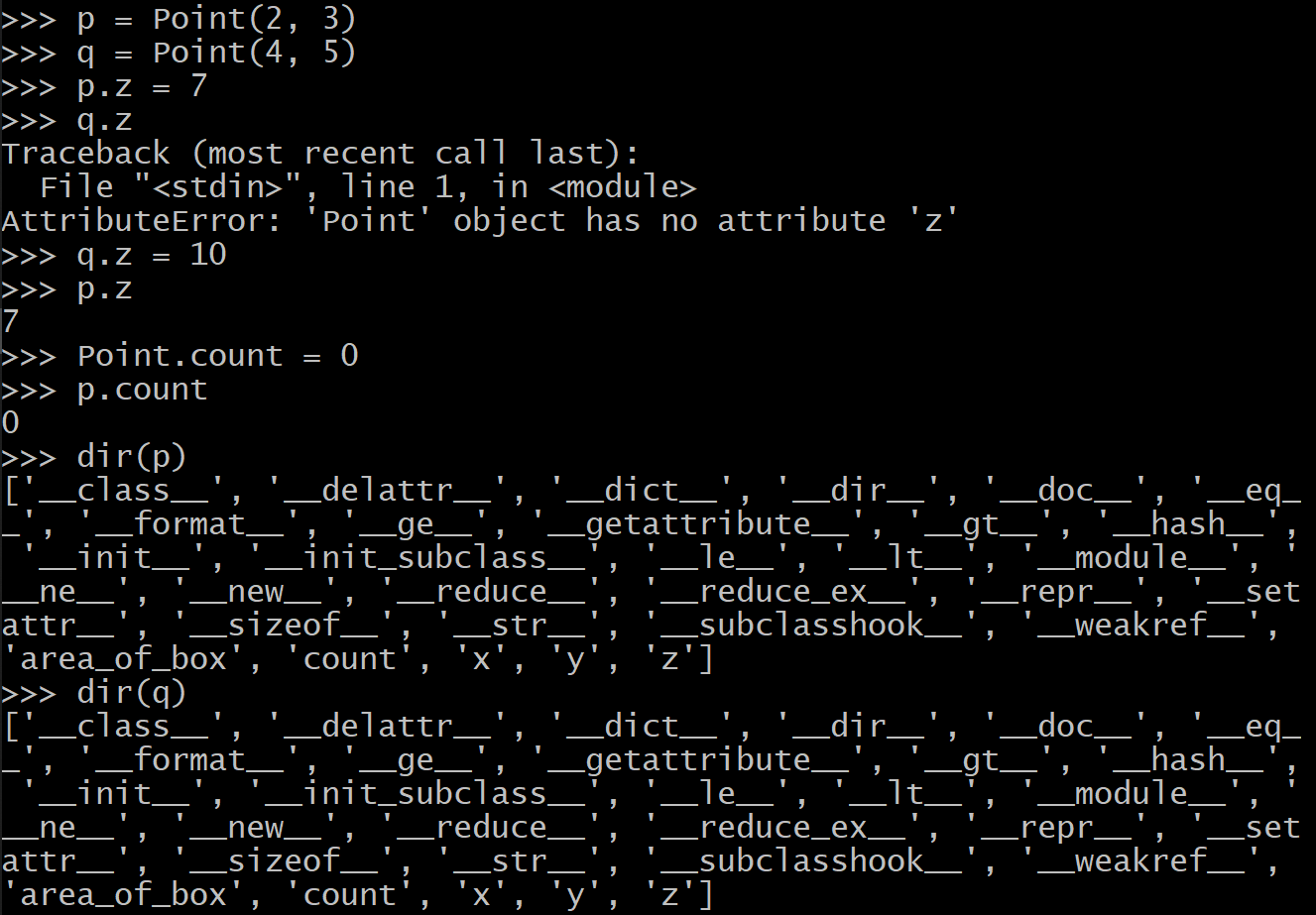


Does it include 'count' as a key in this dict?

-> Yes

What about dir(q)?





1. If you want your **constructor** to increment a **class attribute** to count the number of instances created so far,
   1. How should you initialize the **class attribute** count = 0?

-> at the beginning of the class

* 1. How should you **increment** the class attribute count **in the constructor**? As self.count += 1 or as Point.count += 1? Why?

-> as Point.count += 1  
-> since count is the class attribute

1. Why is it better to define setter/getter methods than allowing user code to modify the attributes directly? 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  
   ?  
   -> “setter” is better
2. In DateTime class shown on slide #35, the attributes are named with an underscore in front, such as \_year, \_month, \_day, etc. What is the reason for this?  
   -> to protect attribute from outside access
3. 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   
   1. What is the purpose of self.\_\_dict\_\_['\_'+field\_name] = field\_value?

-> 把外面的function搬到class裡面，把不同欄位的屬性存取的動作，集中到check\_and\_set裡

If field\_name is 'year', self is p, and field\_value is 2010, then what attribute of p gets assigned the value 2010?

-> \_year

* 1. 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?  
     -> Because you can compute the key (attribute name) !

1. 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) -> getter method  
   and   
   dt.month = 5 -> setter method  
   ?
2. 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, b, d

* 1. 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

* 1. 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, cls

* 1. 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

-> Correct

* 1. 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.

-> Correct

1. 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.
   1. define it as an **instance method**

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

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

@classmethod

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

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

@staticmethod

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

* 1. Is there any difference in how you would call the leap method?

-> No

* 1. Which of the three methods would be the preferred way and why?  
     -> staticmethod

-> no name pollution, does not pass cls or self unnecessarily

## 2. Programming

1. (Difficulty: ★★☆☆☆) Define a class for 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 + ...

Your Polynomial constructor would take variable-length arguments for the coefficients from the 0th 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 \* 32 + 7 \* 33 + 34  = 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 function. 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 special symbol, Python lets you say  
>>> f(3)  
324   
which is a more concise way than saying f.evaluate(3).

1. (Difficulty: ★★★☆☆) Define a class for Temperature. The requirements are
   1. The constructor should take two arguments (degree, unit):
   2. degree is an int or float
      1. The constructor needs to check if degree is an int or float; if not, raise a TypeError.
   3. unit defaults to 'C' for Celsius, but it can be 'F' for Fahrenheit
      1. The constructor needs to check if the unit is an allowed character; if not, raise a ValueError. Actually, lower case 'c' and 'f' are also accepted, but they should be converted to the upper case.
   4. The \_\_repr\_\_() method should return a string that, when printed, is a constructor call that yields the same value as the object.
   5. Define an instance method named get\_temp(). It should return a tuple (*degree*, *unit*). It takes one optional argument for the unit, which should be either 'C' (default) or 'F', in the same way as the constructor.
   6. Define a **property** named degree. You should define two methods
      1. get\_degree(), which returns the value of the \_degree attribute
      2. set\_degree(), which assigns the parameter value to the \_degree attribute
      3. use degree = @property(...) to make degree a property
   7. 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  
(50, 'C')  
>>> c.get\_temp('F')  
(122.0, 'C')  
>>> 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  
'68.000 F'

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

(之後會再更新, will be update soon)