

14.1__OOP(3)

April 17, 2023

1 Introduction to Python for Open Source Geocomputation



- Instructor: Dr. Wei Kang
- Class Location and Time: ENV 336, Mon & Wed 12:30 pm - 1:50 pm

Content:

- Class: parent and child classes
- Inheritance
- Review of OOP

2 Overview of Class

```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy

    def distance(self):
```

```

        return (self.x**2 + self.y**2)**0.5

    def __str__(self):
        return "<" + str(self.x) + ", " + str(self.y) + ">"

    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)

p1 = Point(1,2)
p2 = Point()
p1.translate(1,2)
distance_p2 = p2.distance()
print(p1)
p3 = p1 + p2
print(p3)

```

Class activity: discuss with your group members about the python program above for 5 minutes.
 * What does the program do? * What are the main components? * What are the attributes? *
 What are the methods? * What are the outputs?

```

[1]: class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point dx units to the right and dy units up
        """
        self.x = self.x + dx
        self.y = self.y + dy

    def distance(self):
        return (self.x**2 + self.y**2)**0.5

    def __str__(self):
        return "<" + str(self.x) + ", " + str(self.y) + ">"

    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)

p1 = Point(1,2)
p2 = Point()
p1.translate(1,2)
distance_p2 = p2.distance()
print(distance_p2)
print(p1)
p3 = p1 + p2

```

```
print(p3)
```

```
5.0  
<2,4>  
<5,8>
```

```
[2]: print(p1)
```

```
<2,4>
```

```
[3]: vary = p1.translate(1,2)
```

```
[4]: print(p1)
```

```
<3,6>
```

```
[ ]:
```

2.1 Class and Inheritance in OOP

- Inheritance: How attributes and methods of a **parent class** are passed down to **offspring classes**
- Derivation: The creation of subclasses, which are new classes which retain the data and functionality of the existing class, but permit additional modification and customization
- Hierarchy: Multiple generations of derivation.

2.2 Hierarchies

- parent class (superclass)
- child class (subclass)
 - inherits all attributes and methods of parent class
 - add more info (attributes)
 - add more behavior (methods)
 - override behavior

2.2.1 Inheritance: parent class

```
class Animal:  
    def __init__(self, age):  
        self.age = age  
        self.name = None  
    def __str__(self):  
        return "animal:" + str(self.name)+":"+str(self.age)
```

- defining a class **Animal**
- two attributes: **age**, **name**
 - The **None** keyword is used to define a null value, or no value at all. **None** is not the same as 0, False, or an empty string. **None** is a data type of its own (NoneType) and only **None** can be **None**.

- customize `print()` function with the definition of `__str__`

```
[5]: class Animal:
      def __init__(self, age):
          self.age = age
          self.name = None
      def __str__(self):
          return "animal:" + str(self.name)+":"+str(self.age)
```

```
[6]: a1 = Animal(2)
```

```
[7]: print(a1)
```

```
animal:None:2
```

2.2.2 Inheritance: subclass Cat

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:" + str(self.name)+":"+str(self.age)
```

- defining a class `Cat` which inherits everything from the parent class `Animal`
- `__init__` is not missing, uses the `Animal` version
 - two attributes: `age`, `name`
- add new functionality with `speak()`
- override `__str__` to customize `print()` function to better work with `Cat`

```
[8]: class Cat(Animal):
      def speak(self):
          print("meow")
      def __str__(self):
          return "cat:" + str(self.name)+":"+str(self.age)
```

```
[9]: help(Cat)
```

```
Help on class Cat in module __main__:
```

```
class Cat(Animal)
|   Cat(age)
|
|   Method resolution order:
|       Cat
|       Animal
|       builtins.object
|
|   Methods defined here:
```

```

|   __str__(self)
|       Return str(self).
|
|   speak(self)
|
|   -----
|   Methods inherited from Animal:
|
|   __init__(self, age)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   -----
|   Data descriptors inherited from Animal:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

```

```
[10]: c1 = Cat(1.5)
```

```
[11]: print(c1)
```

```
cat:None:1.5
```

```
[12]: c1.speak()
```

```
meow
```

```
[13]: a1.speak()
```

```

-----
AttributeError                                Traceback (most recent call last)
Cell In[13], line 1
----> 1 a1.speak()

AttributeError: 'Animal' object has no attribute 'speak'

```

2.2.3 Which method to use?

- subclasses can have methods with same name as their superclass
- for an instance of a class, look for a method name in **current** class definition
 - if not found, look for method name up the hierarchy (in parent, then grandparent, and so on)
 - use first method up the hierarchy that you found with that method name

2.2.4 Inheritance: subclass Person

```
class Person(Animal):
    def __init__(self, name, age, friends):
        Animal.__init__(self, age)
        self.name = name
        self.friends = friends

    def speak(self):
        print("Hello!")
    def __str__(self):
        return "person:" + str(self.name)+":"+str(self.age)
```

- defining a class `Person` which inherits everything from the parent class `Animal`
- `__init__` is overridden
 - three attributes
 - call `Animal`'s `__init__` method
- add new functionality with `speak()`
- override `__str__` to customize `print()` function to better work with `Person`

```
[ ]: class Person(Animal):
    def __init__(self, name, age, friends):
        Animal.__init__(self, age)
#         self.age = age
        self.name = name
        self.friends = friends

    def speak(self):
        print("Hello!")
    def __str__(self):
        return "person:" + str(self.name)+":"+str(self.age)+ ":"+str(self.
↪friends)
```

```
[ ]: p1 = Person("Peter",23, ["Hanna", "Wendy"] )
```

```
[ ]: print(p1)
```

```
[ ]: p1.speak()
```

Exercise:

```
class Person(Animal):
    def __init__(self, name, age, friends):
        Animal.__init__(self, age)
        self.name = name
        self.friends = friends

    def speak(self):
        print("Hello!")
```

```
def __str__(self):
    return "person:" + str(self.name)+":"+str(self.age)+ ":"+str(self.friends)
```

Define class `Student` which is a subclass of `Person` above. * Add another attribute `major` and instantiate its value in `__init__`. * Override `speak` method by printing out I have homework * Override `print` function so that it will print `student` instead of `person`

Raise your hand when you are done

```
[ ]: class Student(Person):
    def __init__(self, name, age, friends, major_student):
        Person.__init__(self, name, age, friends)
        self.major = major_student
    def speak(self):
        print("I have homework")
    def talk(self):
        self.speak()
    def __str__(self):
        return "student:" + str(self.name)+":"+str(self.age)+ ":"+str(self.
↵friends) +":" + str(self.major)
```

```
[ ]: s1 = Student("Jane", 19, ["Hanna", "Wendy"], "Geography")
```

```
[ ]: s1.speak()
```

```
[ ]: s1.talk()
```

```
[ ]: print(s1)
```

2.3 OOP

- create your own collections of data
- organize information
- division of work
- access information in a consistent manner
- add layers of complexity
- like functions, classes are a mechanism for decomposition and abstraction in programming

2.3.1 Characteristics of OOP

