Inheritance and Interfaces 7

Computer Science 61A

October 16, 2014

1 Inheritance

Today, we explore another powerful tool that comes with object-oriented programming — inheritance.

Suppose we want to write `Dog` and `Cat` classes. Here's our first attempt:

```
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says woof!"

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says meow!"
```

Notice that both the `Dog` and `Cat` classes have a `name`, `owner`, `eat` method, and `talk` method. That's a lot of effort for so much repeated code!

This is where **inheritance** comes in. In Python, a class can **inherit** the instance variables and methods of a another class without having to type them all out again. For example:

```
class Foo(object):
    # This is the superclass

class Bar(Foo):
    # This is the subclass
```

`Bar` inherits from `Foo`. We call `Foo` the **superclass** (the class that is being inherited) and `Bar` the **subclass** (the class that does the inheriting).

Notice that `Foo` also inherits from class, the `object` class. In Python, `object` is the top-level superclass — everything inherits from it, whether directly or through other superclasses. `object` provides basic functionality that is needed for other classes to work with Python.

1.1 When should we use inheritance?

One common use of inheritance is to represent a hierachcal relationship between two or more classes — one class is a more specific version of the other class. For example, dogs are a specific type of pet, and a pet is a specific type of animal.

Using inheritance, here is a second attempt at representing `Dog`s.

```
class Pet(object):
    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print('...')

class Dog(Pet):
    def __init__(self, name, owner, color):
        Pet.__init__(self, name, owner)
        self.color = color
    def talk(self):
        print('woof!')
```

Notice that, by using inheritance, we did not have to redefine `self.name`, `self.owner`, or the `eat` method. We did, however, redefine the `talk` method in the `Dog` class. In this case, we want `Dog`s to `talk` differently, so we **override** the method.

The line `Pet.__init__(self, name, owner)` in the `Dog` class is necessary for inheriting the instance attributes `self.is_alive`, `self.name`, and `self.owner`. Without this line, `Dog` will never inherit those instance attributes. Notice that when we call `Pet.__init__`, we need to pass in `self`, since `Pet` is a class, not an instance.

1.2 Questions

1. Implement the `Cat` class by inheriting from the `Pet` class. Make sure to use superclass methods wherever possible. In addition, add a `lose_life` method to the `Cat` class.

```
class Cat(Pet):
    def __init__(self, name, owner, lives=9):




    def talk(self):
        """A cat says meow! when asked to talk."""



    def lose_life(self):
        """A cat can only lose a life if they have at least
        one life. When lives reach zero, the 'is_alive'
        variable becomes False.
        """
```

2. Assume these commands are entered in order. What would Python output?

```
>>> class Foo(object):
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)
>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val
>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a
```

```
>>> b.a
```

```
>>> f.garply()
```

```
>>> b.garply()
```

```
>>> b.a = 9
>>> b.garply()
```

```
>>> f.baz = lambda val: val * val
>>> f.garply()
```

1.3 Extra Questions

1. More Cats!

```
class NoisyCat(Cat):
    """A class that behaves just like a Cat, but always
    repeats things twice.
    """
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?



    def talk(self):
        """A NoisyCat will always repeat what he/she said
        twice.
        """
```

2 Interfaces

In computer science, an **interface** is a shared set of attributes, along with a specification of the attributes' behavior. For example, an interface for vehicles might consist of the following methods:

- `def drive(self)`: Drives the vehicle if it has stopped.
- `def stop(self)`: Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a `Car` class and a `Train` can both implement the interface described above, but the `Car` probably has a different mechanism for `drive` than the `Train`.

The power of interfaces is that other programs don't have to know *how* each data type implements the interface — only that they *have* implemented the interface. The following `travel` function can work with both `Car`s and `Train`s:

```
def travel(vehicle):
    while not at_destination():
        vehicle.drive()
    vehicle.stop()
```

2.1 Interfaces in Python

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- `def __add__(self, other)`: Allows objects to do `self + other`.
- `def __sub__(self, other)`: Allows objects to do `self - other`.
- `def __mul__(self, other)`: Allows objects to do `self * other`.

In addiiton, there is also an interface for sequences:

- `def __len__(self)`: Allows objects to do `len(self)`.
- `def __getitem__(self, index)`: Allows objects to do `self[i]`.

2.2 Questions

Let's implement a `Vector` class that support basic operations on vectors. These include adding and subtracting vectors of the same length, multiplying a vector with a scalar, and taking the dot product of two vectors. The results of these operations are shown in the table below:

Operation	Result
`-Vector([1, 2, 3])`	`Vector([-1, -2, -3])`
`Vector([1, 2, 3]) + Vector([4, 5, 6])`	`Vector([5, 7, 9])`
`Vector([4, 5, 6]) - Vector([1, 2, 3])`	`Vector([3, 3, 3])`
`Vector([1, 2, 3]) * Vector([1, 2, 3])`	`14`
`Vector([1, 2, 3]) * 4`	`Vector([4, 8, 12])`
`10 * Vector([1, 2, 3])`	`Vector([10, 20, 30])`
`len(Vector([1, 2, 3]))`	`3`
`Vector([1, 2, 3])[1]`	`2`

We begin with an implmentation of the `Vector` class:

```
class Vector:
    def __init__(self, vector):
        self.vector = vector

    def __neg__ (self)        : "*** YOUR CODE HERE ***"
    def __add__ (self, other): "*** YOUR CODE HERE ***"
    def __sub__ (self, other): return self.__add__(-other)
    def __mul__ (self, other): "*** YOUR CODE HERE ***"
    def __rmul__(self, other): return self.__mul__(other)
    def __len__(self)         : return len(self.vector)
    def __getitem__(self, n) : return self.vector[n]
```

1. Implement `__neg__`, which returns a new `Vector` that is the negation of the current vector, `self`. Try using list comprehensions.

```
def __neg__(self):
    return Vector(_____________________________________________
       )
```

2. Implement `__add__`, which takes in two vectors of the same length and returns a new vector which is their sum. Try using list comprehensions.

```
def __add__(self, other):
    assert type(other) == Vector, "Invalid operation!"
    assert len(self) == len(other), "Invalid dimensions!"
```

3. Implement `__mul__`, which takes in a value, and performs a scalar product if the value is a number, or a vector product if the value is another vector.

```
def __mul__(self, other):
    if type(other) == int or type(other) == float:
        "*** YOUR CODE HERE ***"







    elif type(other) == Vector:
        "*** YOUR CODE HERE ***"
```

2.3 Extra Questions

1. Now that we have a definition of a vector and its basic operations using type dispatching, we can write more complex expressions using Python's operator syntax.

$$\text{Length}(\boldsymbol{v}) = ||\boldsymbol{v}|| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}}$$
$$\text{Norm}(\boldsymbol{v}) = \frac{\boldsymbol{v}}{||\boldsymbol{v}||}$$
$$\text{Proj}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{v}\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\boldsymbol{v} \cdot \boldsymbol{v}}$$

Now write these vector functions using the Python operators we've just defined. Notice how much cleaner this is compared to using function calls.

```
from math import sqrt
def vector_length(v):

    return ________________________________________

def normalize(v):

    return ________________________________________

def proj(u, v):

    return ________________________________________
```