Object-Oriented Programming and Inheritance

Introduction

Object-oriented programming (OOP) is a style of programming that allows you to think of code in terms of "objects." Here's an example of a `Car` class:

```
class Car(object):
    num_wheels = 4

    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):
        if self.wheels > 0:
            self.wheels -= 1
```

Here's some terminology:

- **class**: a blueprint for how to build a certain type of object. The `Car` class (shown above) describes the behavior and data that all `Car` objects have.
- **instance**: a particular occurrence of a class. In Python, we create instances of a class like this:

  ```
  >>> my_car = Car('red')
  ```

  `my_car` is an instance of the `Car` class.

- **attribute** or **field**: a variable that belongs to the class. Think of an attribute as a quality of the object: cars have *wheels* and *color*, so we have given our `Car` class `self.wheels` and `self.color` attributes. We can access attributes using **dot notation**:

  ```
  >>> my_car.color
  'red'
  >>> my_car.wheels
  4
  ```

- **method**: Methods are just like normal functions, except that they are tied to an instance or a class. Think of a method as a "verb" of the class: cars can *drive* and also *pop their tires*, so we have given our `Car` class the methods `drive` and `pop_tire`. We call methods using **dot notation**:

  ```
  >>> my_car = Car('red')
  >>> my_car.drive()
  'red car goes vroom!'
  ```

- **constructor**: As with data abstraction, constructors describe how to build an instance of the class. Most classes have a constructor. In Python, the constructor of the class defined as `__init__`. For example, here is the `Car` class's constructor:

  ```
  def __init__(self, color):
      self.wheels = Car.num_wheels
      self.color = color
  ```

  The constructor takes in one argument, `color`. As you can see, the constructor also creates the `self.wheels` and `self.color` attributes.

- `self`: in Python, `self` is the first parameter for many methods (in this class, we will only use methods whose first parameter is `self`). When a method is called, `self` is bound to an instance of the class. For example:

  ```
  >>> my_car = Car('red')
  >>> car.drive()
  ```

  Notice that the `drive` method takes in `self` as an argument, but it looks like we didn't pass one in! This is because the dot notation *implicitly* passes in `car` as `self` for us.

Types of variables

When dealing with OOP, there are three types of variables you should be aware of:

- **local variable**: These are just like the variables you see in normal functions — once the function or method is done being called, this variable is no longer able to be accessed. For example, the `color` variable in the `__init__` method is a local variable (not the `self.color` variable).
- **instance attribute**: Unlike local variables, instance attributes will still be accessible after method calls have finished. Each instance of a class keeps its own version of the instance attribute — for example, we might have two `Car` objects, where one's `self.color` is red, and the other's `self.color` is blue.

  ```
  >>> car1 = Car('red')
  >>> car2 = Car('blue')
  >>> car1.color
  'red'
  >>> car2.color
  'blue'
  >>> car1.color = 'yellow'
  >>> car1.color
  'yellow'
  >>> car2.color
  'blue'
  ```

- **class attribute**: As with instance attributes, class attributes also persist across method calls. However, unlike instance attributes, all instances of a class will share the same class attributes. For example, `num_wheels` is a class attribute of the `Car` class.

  ```
  >>> car1 = Car('red')
  >>> car2 = Car('blue')
  >>> car1.num_wheels
  4
  ```

```
>>> car2.num_wheels
4
>>> Car.num_wheels = 2
>>> car1.num_wheels
2
>>> car2.num_wheels
2
```

Notice that we can access class attributes by saying `<class name>.<attribute>`, such as `Car.num_wheels`, or by saying `<instance>.<attribute>`, such as `car1.num_wheels`.

Question 1

Predict the result of evaluating the following calls in the interpreter. Then try them out yourself!

```
>>> class Account(object):
...     interest = 0.02
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
...     def deposit(self, amount):
...         self.balance = self.balance + amount
...         print("Yes!")
...
>>> a = Account("Billy")
>>> a.account_holder
______

>>> a.holder
______

>>> Account.holder
______

>>> Account.interest
______

>>> a.interest
______

>>> Account.interest = 0.03
>>> a.interest
______

>>> a.deposit(1000)
______

>>> a.balance
______

>>> a.interest = 9001
>>> Account.interest
______
```

Question 2

Modify the following `Person` class to add a `repeat` method, which repeats the last thing said. See the docstring tests for an example of its use.

Hint: you will have to modify other methods as well, not just the `repeat` method.

```
class Person(object):
    """Person class. Docstring tests follow:

    >>> steven = Person("Steven")
    >>> steven.repeat()       # starts at whatever value you'd like
    'I squirreled it away before it could catch on fire.'
    >>> steven.say("Hello")
    'Hello'
    >>> steven.repeat()
    'Hello'
    >>> steven.greet()
    'Hello, my name is Steven'
    >>> steven.repeat()
    'Hello, my name is Steven'
    >>> steven.ask("preserve abstraction barriers")
    'Would you please preserve abstraction barriers'
    >>> steven.repeat()
    'Would you please preserve abstraction barriers'

    """
    # Class definitons begin here:

    def __init__(self, name):
        self.name = name

    def say(self, stuff):
        return stuff

    def ask(self, stuff):
        return self.say("Would you please " + stuff)

    def greet(self):
        return self.say("Hello, my name is " + self.name)

    def repeat(self):
        "*** YOUR CODE HERE ***"
```

Inheritance

Question 3

Predict the result of evaluating the following calls in the interpreter. Then try them out yourself!

```
>>> class Account(object):
...     interest = 0.02
...     def __init__(self, account_holder):
...         self.balance = 0
...         self.holder = account_holder
...     def deposit(self, amount):
...         self.balance = self.balance + amount
...         print("Yes!")
...
>>> class CheckingAccount(Account):
...     def __init__(self, account_holder):
...         Account.__init__(self, account_holder)
...     def deposit(self, amount):
...         Account.deposit(self, amount)
...         print("Have a nice day!")
...
>>> a = Account("Billy")
>>> a.balance
______

>>> c = CheckingAccount("Eric")
>>> c.balance
______

>>> a.deposit(30)
______

>>> c.deposit(30)
______

>>> c.interest
______
```

Question 4

Suppose now that we wanted to define a class called `DoubleTalker` to represent people who always say things twice:

```
>>> steven = DoubleTalker("Steven")
>>> steven.say("hello")
"hello hello"
>>> steven.say("the sky is falling")
"the sky is falling the sky is falling"
```

Consider the following three definitions for `DoubleTalker` that inherit from the `Person` class:

```
class DoubleTalker(Person):
```

```
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return Person.say(self, stuff) + " " + self.repeat()

class DoubleTalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return stuff + " " + stuff

class DoubleTalker(Person):
    def __init__(self, name):
        Person.__init__(self, name)
    def say(self, stuff):
        return Person.say(self, stuff + " " + stuff)
```

Determine which of these definitions work as intended. Also determine for which of the methods the three versions would respond differently. (Don't forget about the `repeat` method!)

Question 5

Modify the `Account` class so that it has a new attribute, `transactions`, that is a list keeping track of any transactions performed. Add a `report()` method that will print the transactions list in a suitable printout (Hint: use the print format utility). See the docstring tests for examples.

```
class Account(object):
    """A bank account that allows deposits and withdrawals.

    >>> eric_account = Account('Eric')
    >>> eric_account.deposit(10000)   # depositing my paycheck for the
week
    10000
    >>> eric_account.transactions
    [('deposit', 10000)]
    >>> eric_account.withdraw(100)      # buying dinner
    999900
    >>> eric_account.transactions
    [('deposit', 10000), ('withdraw', 100)]
    >>> eric_account.report()

    Transaction Report for Eric

    Type         Amount
    ================================
    deposit       10000
    withdraw        100
    --------------------------------
    Balance        9900

    """

    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
```

```
        self.holder = account_holder

    def deposit(self, amount):
        """Increase the account balance by amount and return the
        new balance.
        """
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Decrease the account balance by amount and return the
        new balance.
        """
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```