MUTATION AND OOP 5

COMPUTER SCIENCE 61A

February 23, 2017

1 List Mutation

Let's imagine you order a mushroom and cheese pizza from Domino's, and that they represent your order as a list:

```
>>> pizza1 = ['cheese', 'mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, Domino's would have to build an entirely new list to add onions:

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all Domino's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i` (does not replace element but adds a new one)
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `pop(i)` removes and returns the element at index `i`

List methods are called via *dot notation*, as in:

```
>>> sharks = ['joe thornton', 'patrick marleau']
>>> sharks.append('logan couture')
>>> sharks.pop(1)
'patrick marleau'
>>> sharks
['joe thornton', 'logan couture']
```

1.1 Questions

1. Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*. It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2 #compares each value


>>> lst1 is lst2 #compares references


>>> lst2 = lst1
>>> lst2 is lst1


>>> lst1.append(4)
>>> lst1


>>> lst2


>>> lst2[1] = 42
```

```
>>> lst2


>>> lst1 = lst1 + [5]
>>> lst1 == lst2


>>> lst1


>>> lst2


>>> lst2 is lst1
```

2. Write a function that removes all instances of an element from a list.

```
def remove_all(el, lst):
    """
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
```

3. Write a function `square_elements` which takes a `lst` and replaces each element with the square of that element. *Mutate* `lst` *rather than returning a new list*.

```
def square_elements(lst):
    """
    >>> lst = [1, 2, 3]
    >>> square_elements(lst)
    >>> lst
    [1, 4, 9]
    """
```

2 Object Oriented Programming

In a previous lecture, you were introduced to the programming paradigm known as Object-Oriented Programming (OOP). OOP allows us to treat data as objects - like we do in real life.

For example, consider the **class** `CS61A_Student`. Each of you as individuals are an **instance** of this class. So, a student `Mitas` would be an instance of the class `CS61A_Student`.

Details that all CS61A students have, such as `name`, `year`, and `major`, are called **instance attributes**. Every student has these attributes, but their values differ from student to student. An attribute that is shared among all instances of `CS61A_Student` is known as a **class attribute**. An example would be the `instructors` attribute; the instructor for 61A, Professor DeNero, is the same for every student in CS61A.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are said to be **methods**. In this case, these actions would be bound methods of `CS61A_Student` objects.

Here is a recap of what we discussed above:

- **class:** a template for creating objects
- **instance:** a single object created from a class
- **instance attribute:** a property of an object, specific to an instance
- **class attribute:** a property of an object, shared by all instances of the same class
- **method:** an action (function) that all instances of a class may perform

2.1 Questions

1. Below we have defined the classes `Instructor`, `Student`, and `TeachingAssistant`, implementing some of what was described above. Remember that we pass the `self` argument implicitly to instance methods when using dot-notation.

```
class Instructor:
    degree = "PhD (Magic)" # this is a class attribute
    def __init__(self, name):
        self.name = name # this is an instance attribute

    def lecture(self, topic):
        print("Today we're learning about " + topic)

dumbledore = Instructor("Dumbledore")
class Student:
    instructor = dumbledore

    def __init__(self, name, ta):
        self.name = name
        self.understanding = 0
        ta.add_student(self)

    def attend_lecture(self, topic):
        Student.instructor.lecture(topic)
        print(Student.instructor.name + " is awesome!")
        self.understanding += 1

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)

class TeachingAssistant:
    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1
```

What will the following lines output?

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")


>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")


>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))


>>> harry.understanding


>>> snape.students["Hermione"].understanding


>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
# Equivalent to harry.attend_lecture("transfiguration")
```

3 Inheritance

Let's explore another powerful object-oriented programming tool: **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):
        print(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):
        print(self.name + " says meow!")
```

Notice that the only difference between both the `Dog` and `Cat` classes are the `talk` method as well as the `color` and `lives` attributes. That's a lot of 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 base class

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

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

Notice that `Foo` also inherits from the `object` class. In Python, `object` is the top-level base class that provides basic functionality; everything inherits from it, even when you don't specify a class to inherit from. One common use of inheritance is to represent a hierarchical relationship between two or more classes where one class *is a* more specific version of the other class. For example, a dog *is a* pet.

```
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(self.name)

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

By making `Dog` a subclass of `Pet`, we did not have to redefine `self.name`, `self.owner`, or `eat`. However, since we want `Dog` to `talk` differently, we did redefine, or **override**, the `talk` method.

The line `Pet.__init__(self, name, owner)` in the `Dog` class is necessary for inheriting the instance attributes and methods from `Pet`. Notice that when we call `Pet.__init__`, we need to pass in `self` as a regular argument (that is, inside the parentheses, rather than by dot-notation) since `Pet` is a class, not an instance.

3.1 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 reaches zero, 'is_alive'
        becomes False.
        """
```

2. More cats! Fill in the methods for `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot, printing twice whatever a `Cat` says.

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



    def talk(self):
        """Repeat what a Cat says twice."""
```

3. What would Python print? (Summer 2013 Final)

```
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4
```

```
>>> x, y = A(), B()
>>> x.f()


>>> B.f()


>>> x.g(x, 1)


>>> y.g(x, 2)
```

4. Implement the `Yolo` class so that the following interpreter session works as expected. (Summer 2013 Final)

```
>>> x = Yolo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.motto = 5
>>> x.g(5)
10
```