CS 61A
Spring 2023

OOP, Inheritance, String Representation

Discussion 8: March 15, 2023

Inheritance

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which the similar classes **inherit**. For example, we can write a class called **Pet** and define **Dog** as a **subclass** of **Pet**:

```
class Pet:

    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 talk(self):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet (We use **is a** to describe this sort of relationship in OOP languages, and not to refer to the Python `is` operator).

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk()` method from the `Pet` class, but passing the `Dog` instance as the `self`.

This is a little bit of a simplification, and if you're interested you can read more in the Python documentation on `super`.

Q1: That's inheritance, init?

Let's say we want to create a class `Monarch` that inherits from another class, `Butterfly`. We've partially written an `__init__` method for `Monarch`. For each of the following options, state whether it would correctly complete the method so that every instance of `Monarch` has all of the instance attributes of a `Butterfly` instance. You may assume that a monarch butterfly has the default value of 2 wings.

```
class Butterfly():
    def __init__(self, wings=2):
        self.wings = wings

class Monarch(Butterfly):
    def __init__(self):
        ________________________________________
        self.colors = ['orange', 'black', 'white']
```

```
super.__init__()
```

```
super().__init__()
```

```
Butterfly.__init__()
```

```
Butterfly.__init__(self)
```

Some butterflies like the Owl Butterfly have adaptations that allow them to mimic other animals with their wing patterns. Let's write a class for these `MimicButterflies`. In addition to all of the instance variables of a regular `Butterfly` instance, these should also have an instance variable `mimic_animal` describing the name of the animal they mimic. Fill in the blanks in the lines below to create this class.

```
class MimicButterfly(______________):
    def __init__(self, mimic_animal):
        _______________.__init__()
        ______________ = mimic_animal
```

What expression completes the first blank?

What expression completes the second blank?

What expression completes the third blank?

Q2: Shapes

Fill out the skeleton below for a set of classes used to describe geometric shapes. Each class has an `area` and a `perimeter` method, but the implementation of those methods is slightly different. Please override the base `Shape` class's methods where necessary so that we can accurately calculate the perimeters and areas of our shapes with ease.

```
import math
pi = math.pi

class Shape:
    """All geometric shapes will inherit from this Shape class."""
    def __init__(self, name):
        self.name = name

    def area(self):
        """Returns the area of a shape"""
        print("Override this method in ", type(self))

    def perimeter(self):
        """Returns the perimeter of a shape"""
        print("Override this function in ", type(self))

class Circle(Shape):
    """A circle is characterized by its radii"""
    def __init__(self, name, radius):
        "*** YOUR CODE HERE ***"









    def perimeter(self):
        """Returns the perimeter of a circle  (2r)"""
        "*** YOUR CODE HERE ***"









    def area(self):
        """Returns the area of a circle  (r^2)"""
        "*** YOUR CODE HERE ***"









class RegPolygon(Shape):
    """A regular polygon is defined as a shape whose angles and side lengths are all the
    same.
    This means the perimeter is easy to calculate. The area can also be done, but it's
    more inconvenient."""
    def __init__(self, name, num_sides, side_length):
        "*** YOUR CODE HERE ***"
```

Q3: Cat

Below is a skeleton for the `Cat` class, which inherits from the `Pet` class we saw in the Inheritance introduction. To complete the implementation, override the `__init__` and `talk` methods and add a new `lose_life` method.

Hint: You can call the `__init__` method of `Pet` (the superclass of `Cat`) to set a cat's `name` and `owner`.

Hint: The `__init__` method is not a real constructor, and can be called like any other method.

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        "*** YOUR CODE HERE ***"

    def talk(self):
        """Print out a cat's greeting.

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """
        "*** YOUR CODE HERE ***"

    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches zero,
        is_alive becomes False. If this is called after lives has
        reached zero, print 'This cat has no more lives to lose.'
        """
        "*** YOUR CODE HERE ***"

    def revive(self):
        """Revives a cat from the dead. The cat should now have
        9 lives and is_alive should be true. Can only be called
        on a cat that is dead. If the cat isn't dead, print
        'This cat still has lives to lose.'
        """
        if not self.is_alive:
            ---------------------------
        else:
            ---------------------------
```

Q4: NoisyCat

More cats! Fill in this implementation of a class called `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot: in fact, it talks twice as much as a regular `Cat`! If you'd like to test your code, feel free to copy over your solution to the `Cat` class above.

```
class __________ # Fill me in!
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        "*** YOUR CODE HERE ***"









    def talk(self):
        """Talks twice as much as a regular cat.
        >>> NoisyCat('Magic', 'James').talk()
        Magic says meow!
        Magic says meow!
        """
        "*** YOUR CODE HERE ***"




# You can use more space on the back if you want
```

Representation: Repr, Str

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes.

`str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development.

When we define a class in Python, `__str__` and `__repr__` are both built-in methods for the class.

We can call those methods using the global built-in functions `str(obj)` or `repr(obj)` instead of dot notation, `obj.__repr__()` or `obj.__str__()`.

In addition, the `print()` function calls the `__str__` method of the object and displays the returned string **with the quotations removed**, while simply calling the object in interactive mode in the interpreter calls the `_repr__` method and displays the returned string **with the quotations removed**.

Here are some examples:

```
class Rational:

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return f'{self.numerator}/{self.denominator}'

    def __repr__(self):
        return f'Rational({self.numerator},{self.denominator})'

>>> a = Rational(1, 2)
>>> str(a)
'1/2'
>>> repr(a)
'Rational(1,2)'
>>> print(a)
1/2
>>> a
Rational(1,2)
```

```
>>> s = "hello" # Python String objects also have __repr__ and __str__ methods!
>>> repr(s)
"'hello'"
>>> s
'hello' # displays the repr string with the outer layer of quotations removed
>>> print(repr(s))
'hello' # printing the repr string removes the outer layer of quotations
>>> str(s)
'hello'
>>> print(s)
hello # displays the str string with quotations removed
```

Q5: WWPD: Repr-esentation

Note: This is not the typical way `repr` is used, nor is this way of writing `repr` recommended, this problem is mainly just to make sure you understand how `repr` and `str` work.

```
class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret
```

Given the above class definitions, what will the following lines output?

```
>>> A('one')
```

```
>>> print(A('one'))
```

```
>>> repr(A('two'))
```

```
>>> b = B()
```

```
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
```

Q6: Cat Representation

Now let's implement the `__str__` and `__repr__` methods for the `Cat` class from earlier so that they exhibit the following behavior:

```
>>> cat = Cat("Felix", "Kevin")
>>> cat
Felix, 9 lives
>>> cat.lose_life()
>>> cat
Felix, 8 lives
>>> print(cat)
Felix
```

```
# (The rest of the Cat class is omitted here, but assume all methods from the Cat class
    above are implemented)
    def __repr__(self):
        "*** YOUR CODE HERE ***"




    def __str__(self):
        "*** YOUR CODE HERE ***"




# You can use more space on the back if you want
```