Object Oriented Programming

Computer Science Mentors 61A

March 17 – March 21, 2024

1 Object Oriented Programming

Inheritance is an important feature of object oriented programming. To create an object that shares its attributes or methods with an existing object, we can have the object inherit these similarities instead of repeating code. In addition to making our code more concise, it allows us to create classes based on other classes, similar to how real-world categories are often divided into smaller subcategories.

For example, say the `HybridCar` class inherits from the `Car` class as a type of car:

```
class HybridCar(Car):
    def __init__(self):
        super().__init__()
        self.battery = 100

    def drive(self):
        super().drive()
        self.battery -= 5
        print("Current battery level:", self.gas)

    def brake(self):
        self.battery += 1

my_hybrid = HybridCar()
```

By default, the child class inherits all of the attributes and methods of its parent class. Consequently, we would be able to call `my_hybrid.drive()` and access `my_hybrid.wheels` from the `HybridCar` instance `my_hybrid`. When dot notation is used on an instance, Python will first check the instance to see if the attribute exists, then the instance's class, and then its parent class, etc. If Python goes all the way up the class tree without finding the attribute, an `AttributeError` is thrown.

Additional or redefined instance and class attributes can be added in a child class, such as `battery`. If we decided that hybrid cars should have 3 wheels, we could assign 3 to a class attribute `wheels` in `HybridCar`. `my_hybrid.wheels` would return 3, but `my_car.wheels` would still return 4. We can also **override** inherited instance methods by redefining them in the child class. If we would like to call the parent class's version of a method, we can use **`super`**`()` to access it.

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback
Provided by CSM 61A contributors under CC BY 4.0/3-clause BSD.

1. What would Python display? Write the result of executing the following code and prompts. If nothing would happen, write "Nothing". If an error occurs, write "Error".

```
class ForceWielder():
    force = 25

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

    def train(self, other):
        other.force += self.force / 5

    def __str__(self):
        return self.name

class Jedi(ForceWielder):
    lightsaber = "blue"

    def __str__(self):
        return "Jedi " + self.name

    def __repr__(self):
        return f"Jedi({repr(self.name)})"

class Sith(ForceWielder):
    lightsaber = "red"
    num_sith = 0

    def __init__(self, name):
        super().__init__(name)
        Sith.num_sith += 1
        if self.num_sith != 2:
            print("Two there should be. No more, no less.")

    def __str__(self):
        return "Darth " + self.name

    def __repr__(self):
        return f"Sith({repr(self.name)})"
```

```
>>> anakin = Jedi("Anakin")
>>> anakin.lightsaber, anakin.force


>>> obiwan = Jedi("Obi-wan")
>>> anakin.master = obiwan
>>> anakin.master


>>> Jedi.master


>>> obiwan.force += anakin.force
>>> obiwan.force, anakin.force


>>> obiwan.train(anakin)
>>> obiwan.force, anakin.force


>>> Jedi.train(obiwan, anakin)
>>> obiwan.force, anakin.force


>>> sidious = Sith("Sidious")


>>> ForceWielder.train(sidious, anakin)
>>> anakin.lightsaber = "red"
>>> anakin.lightsaber, anakin.force


>>> Jedi.lightsaber


>>> print(Sith("Vader"), Sith("Maul").num_sith)


>>> rey = ForceWielder("Rey")
>>> rey


>>> rey.lightsaber
```

2. Write `TeamBaller`, a subclass of `Baller`. An instance of `TeamBaller` cheers on the team every time it passes a ball/states it doesn't have the ball when it doesn't and returns a boolean of whether or not it passed the ball. If the `TeamBaller` did not pass the ball, it'll say it hasn't done so and return False. Assume `Baller` has a defined *pass_ball* method.

```
class Baller:
    all_players = []
    def __init__(self, name, has_ball = False):
        self.name = name
        self.has_ball = has_ball
        Baller.all_players.append(self)

    def pass_ball(self, other_player):
        if self.has_ball:
            self.has_ball = False
            other_player.has_ball = True
            return True
        else:
            return False

class BallHog(Baller):
    def pass_ball(self, other_player):
        return False

class TeamBaller(_________________):
    """
    >>> alyssa = BallHog('Alyssa')
    >>> cheerballer = TeamBaller('Esther', has_ball=True)
    >>> cheerballer.pass_ball(alyssa)
    Yay!
    True
    >>> cheerballer.pass_ball(alyssa)
    I don't have the ball
    False
    """
    def pass_ball(_________________, _____________________):
```

3. Let's use OOP to help us implement our good friend, the ping-pong sequence!

   As a reminder, the ping-pong sequence counts up starting from 1 and is always either counting up or counting down.

   At element `k`, the direction switches if `k` is a multiple of 7 or contains the digit 7.

   The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 7th, 14th, 17th, 21st, 27th, and 28th elements:

   ```
   1 2 3 4 5 6 [7] 6 5 4 3 2 1 [0] 1 2 [3] 2 1 0 [-1] 0 1 2 3 4
   [5] [4] 5 6
   ```

   Assume you have a function `has_seven(k)` that returns True if k contains the digit 7.

   ```
   >>> tracker1 = PingPongTracker()
   >>> tracker2 = PingPongTracker()
   >>> tracker1.next()
   1
   >>> tracker1.next()
   2
   >>> tracker2.next()
   1
   ```

   ```
   class PingPongTracker:
       def __init__(self):




       def next(self):
   ```

4. Implement the classes so the following code runs.

```
"""
>>> p = Plant()
>>> p.height
1
>>> p.materials
[]
>>> p.absorb()
>>> p.materials
[|Sugar|]
>>> Sugar.sugars_created
1
>>> p.leaf.sugars_used
0
>>> p.grow()
>>> p.materials
[]
>>> p.height
2
>>> p.leaf.sugars_used
1
"""

class Plant:
    def __init__(self):
        """A Plant has a Leaf, a list of sugars created so far,
        and an initial height of 1.
        """
        self.leaf = Leaf(self)
        self.materials = ____________
        self.height = ____________

    def absorb(self):
        """Calls the Leaf to create sugar."""









    def grow(self):
        """A Plant consumes all of its sugars to grow, each of which
        increases its height by 1.
        """
```

```
class Leaf:
    def __init__(self, plant): # plant is a Plant instance
        """A Leaf is initially alive, and keeps track of how many
        sugars it has created.
        """



    def absorb(self):
        """If this Leaf is alive, a Sugar is added to the plant's
        list of sugars.
        """
        if self.alive:


    def __repr__(self):
        return '|Leaf|'

class Sugar:
    sugars_created = 0

    def __init__(self, leaf, plant):



    def activate(self):
        """A sugar is used."""



    def __repr__(self):
        return '|Sugar|'
```

5. [Exam Level] Implement the `Poll` class and the `tally` function, which takes a choice `c` and returns a list describing the number of votes for `c`. This list contains pairs, each with a name and the number of times `vote` was called on that `choice` at the `Poll` with that name. Pairs can be in any order. Assume all `Poll` instances have distinct names. Hint: the dictionary `get(key, default)` method (MT 2 guide, page 1 top-right) returns the value for a `key` if it appears in the dictionary and `default` otherwise. [Adapted from CS61A Fa18 Midterm 2 Q5(a)]

```
class Poll:
    s = []

    def __init__(self, n):

        self.name = _____________________________________________________

        self.votes = {}

        ____________________________________________________________________

    def vote(self, choice):

        self._____________________________ = _____________________________________


def tally(c):
    """Tally all votes for a choice c as a list of (poll name, vote count)
       pairs.
    >>> a, b, c = Poll('A'), Poll('B'), Poll('C')
    >>> c.vote('dog')
    >>> a.vote('dog')
    >>> a.vote('cat')
    >>> b.vote('cat')
    >>> a.vote('dog')
    >>> tally('dog')
    [('A', 2), ('C', 1)]
    >>> tally('cat')
    [('A', 1), ('B', 1)]
    """

    return_____________________________________________________________________________
```