COMPUTER SCIENCE MENTORS 61A

March 17 – March 21, 2024

Recommended Timeline

- Inheritance Mini Lecture: 5 mins
- Q1: Star Wars WWPD: 10 mins (long and pretty comprehensive)
- Q2 TeamBaller: 8 mins - Easy difficulty
- Q3 Pingpong: 10 mins - Medium difficulty
- Q4 Photosynthesis: 15 mins [Optional] - Medium difficulty but very long.
- Q5 Midterm Elections: 10 mins [Exam Level] - Hard. The use of dictionaries combined with list comprehension might make things a bit tricky, be prepared to explain to students what's happening.

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:

```python
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.

NOTE: AS OF THE SPRING 2023 ITERATION OF THIS COURSE, IT IS DEFINITELY POSSIBLE THAT THIS IS YOUR STUDENTS' FIRST INTERACTION WITH INHERITANCE. We included this section as, honestly, there's not much you can do with OOP without the concepts of inheritance and representation, so as such, we included some baseline examples of such in the problems following this overview.

Again, you probably want to go over this differently than the reference material presented here. I like to draw out a class tree on the board and emphasize that there should be an "is-a" relationship between child class and parent class. For example, a hybrid car "is a" car. The reasoning behind this "is-a" rule of thumb is that objects of the child class should generally have all the same properties as objects of the parent class. It's also often instructive to give some examples that do not work in a class hierarchy. A wagon is not a car. A vehicle is not a car (but a car is a vehicle). A car is not a garage (although a car is contained in a garage).

Variable look-up can be rather confusing for students. If you draw the class hierarchy as a tree on the board, you can demonstrate the process of successively looking up from instance to class and then from child class to parent class until you find the attribute or error out. I tell my students that you can only look up the class hierarchy, not down it.

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".

```python
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
```

("blue", 25)

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

Jedi("Obi-wan")

```
>>> Jedi.master
```

AttributeError

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

(50, 25)

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

(50, 35.0)

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

(50, 45.0)

```
>>> sidious = Sith("Sidious")
```

Two there should be. No more, no less.

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

("red", 50.0)

```
>>> Jedi.lightsaber
```

```
"blue"

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


Two there should be. No more, no less.
Darth Vader 3

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


<__main__.ForceWielder object>

>>> rey.lightsaber


AttributeError
```

To address a point brought up in previous semesters: You may realize that as both a ForceWielder and Jedi, anakin has no attribute named master at first. Note to students that it is possible in our OOP objects for us to declare object attributes on the fly. Also communicate to students that while you can declare object attributes on the fly, that does not mean that such attributes persist to the class the object exists in as a whole.

In my opinion, going through an example like this is far more helpful for students than a mini-lecture. Try to foresee some questions and confusions might have and how you might address them, for example:

- Why, in the __init__ method of Sith can we use self.num_sith instead of Sith.num_sith? And why can't we write self.num_sith += 1?

- Why does evaluating rey give us <__main__.ForceWielder object>, but this is not the case when we evaluated anakin.master?

- What's going on with ForceWielder.train(sidious, anakin)?

- Can we write Jedi.train(sidious, rey), even though neither rey nor sidious are Jedi?

These are also questions you could bring up if students don't ask them.

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.

```python
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(Baller):
    """
    >>> 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(self, other):
        did_pass = Baller.pass_ball(self, other)
        if did_pass:
            print('Yay!')
        else:
            print("I don't have the ball")
        return did_pass
```

Teaching Tips

- This is mean to be an introduction to how object interact with other objects, overwriting methods, and inheritance. As of the Spring 2024 iteration of this course, students are fairly new to the concept of inheritance, and as such, may need a little hand-holding through this problem. Let this serve as a gentle introduction and if they get confused on nuances, clarify.

- Let students think about what methods are being inherited, along with the attributes that are inherited in the init method.

- Remember that although the pass_ball method will be overridden in the TeamBaller method, it can call the inherited class's method.

- Remember that the pass_ball method of the Baller class will return whether the ball was passed or not, which decides whether the TeamBaller instance should cheer or not.

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
```

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




    def next(self):
```

```python
class PingPongTracker:
    def __init__(self):
        self.current = 0
        self.index = 1
        self.add = True

    def next(self):
        if self.add:
            self.current += 1
        else:
            self.current -= 1
        if has_seven(self.index) or self.index % 7 == 0:
            self.add = not self.add
        self.index += 1
        return self.current
```

- Emphasize the fact that the important part of such sequence problems are *keeping track of state at a given time step*. With OOP, this state is inherently saved as object attributes.

- Make sure the difference between `self.current` and `self.index` is clear: index always increases by 1 each step, while current is the actual pingpong sequence number we want.

- Remember that the **index** denotes the the progress we've made along the pingpong sequence-current is just the current number of our sequence we happen to be on.

- Students may have seen a version of pingpong that uses -1 and 1 as a direction variable instead of a boolean "add" variable. Make sure to clarify how the add variable operates here, and how it differs from the -1/1 version.

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 = 1

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

    def grow(self):
```

```python
        """A Plant consumes all of its sugars to grow, each of which
        increases its height by 1.
        """
        for sugar in self.materials:
            sugar.activate()
            self.height += 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 used.
        """
        self.alive = True
        self.sugars_used = 0
        self.plant = plant

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


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


class Sugar:
    sugars_created = 0

    def __init__(self, leaf, plant):
        self.leaf = leaf
        self.plant = plant
        Sugar.sugars_created += 1

    def activate(self):
        """A sugar is used."""
        self.leaf.sugars_used += 1
        self.plant.materials.remove(self)

    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_____
```

```
class Poll:
    s = []
    def __init__(self, n):
        self.name = n
        self.votes = {}
        Poll.s.append(self)
    def vote(self, choice):
        self.votes[choice] = self.votes.get(choice, 0) + 1

def tally(c):
    return [(p.name, p.votes[c]) for p in Poll.s if c in p.votes]
```