OOP Review Worksheet Solutions

Computer Science Mentors

November 24, 2020

1 Build A Bear

1. Let's slowly build a Bear from start to finish using OOP!

   First, let's build a Bear class for our basic bear. Bear instances should have an attribute `name` that holds the name of the bear. The Bear class should have an attribute `bears`, a list that stores the name of each bear.

```
>>> oski = Bear('Oski')
>>> oski.name
'Oski'
>>> Bear.bears
['Oski']
>>> winnie = Bear('Winnie')
>>> Bear.bears
['Oski', 'Winnie']
```

```
class Bear:
    bears = []
    def __init__(self, name):
        self.name = name
        Bear.bears.append(self.name)
```

   Note that just doing `bears.append(self.name)` will result in an error! There is no `bears` variable in the `__init__` function frame.

2. Next, let's build an Organ class to put in our bear. Organ instances should have an attribute `name` that holds the name of the organ. The Organ class should contain a dictionary `organs` that holds (organ, bear name) key-value pairs, mapping each organ object to the name of its bear.
   Note that this maps **objects** to strings! We may need to change the representation of this object for our doc tests to be correct.
   The Organ class should also have a method `discard(self)` that removes the organ from Organ.organs.

```
>>> oski_liver = Organ('liver', oski)
>>> winnie_stomach = Organ('stomach', winnie)
>>> winnie_liver = Organ('liver', winnie)
>>> Organ.organs
{liver: 'Oski', stomach: 'Winnie', liver: 'Winnie'}
>>> winnie_liver.discard()
>>> Organ.organs
{liver: 'Oski', stomach: 'Winnie'}
```

```
class Organ:
    organs = {}
    def __init__(self, name, bear):
        self.name = name
        Organ.organs[self] = bear.name
    def discard(self):
        Organ.organs.pop(self)
    def __repr__(self):
        return self.name
```

   Note if we used `Organ.organs[self.name]` instead of `Organ.organs[self]`, it would be impossible to store 2 Organs with the same name.
   Without the `__repr__`, an Organ returns `<__main__.Organ object>` instead of its name in `Organ.organs`.
   Organs do not inherit from Bear, nor should they. Inheritance is used in **is a** relationships, not **has a**.

3. Now, let's design a Heart class that inherits from the Organ class. When a heart is created, if its bear does not have a heart, it creates a `heart` attribute for that bear. If a bear already has a heart, the old `heart` is discarded and replaced with the new one. Hint: you can use **`hasattr`** to check if a bear has a heart attribute.

```
>>> hasattr(oski, 'heart')
False
>>> oski_heart = Heart('small heart', oski)
>>> oski.heart
small heart
>>> Organ.organs
{liver: 'Oski', stomach: 'Winnie', small heart: 'Oski'}
>>> new_heart = Heart('big heart', oski)
>>> oski.heart
big heart
>>> Organ.organs
{liver: 'Oski', stomach: 'Winnie', big heart: 'Oski'}
```

```
class Heart(Organ):
    def __init__(self, name, bear):
        if hasattr(bear, 'heart'):
            bear.heart.discard()
        bear.heart = self
        Organ.__init__(self, name, bear)
```

Since Hearts are Organs, we can use Organ's `discard` method to remove an old heart easily, without breaking any abstraction barriers.
We also can use `Organ.__init__` instead of repeating code.

4. Finally, let's design a Brain class that inherits from the Organ class. Bears want to rise up against the Bourgeoise, and a Brain gives them the power to do so.
   The appearance of a brain gives the Bear class an attribute `rebels`, a **set** containing the names of all rebelling bears.
   A bear with a brain is added to `rebels` by default.
   It also gives **all** bears, even ones without brains, a `rebel` method that, when called, adds the caller to `rebels`.

```
>>> big_brain = Brain('big brain', oski)
>>> Bear.rebels
{'Oski'}
>>> smokey = Bear('Smokey')
>>> smokey.rebel()
>>> Bear.rebels
{'Smokey', 'Oski'}
>>> small_brain = Brain('brain', winnie)
>>> Bear.rebels
{'Smokey', 'Oski', 'Winnie'}
>>> Organ.organs
{liver: 'Oski', stomach: 'Winnie', big heart: 'Oski', big
   brain: 'Oski', brain: 'Winnie'}
```

```
class Brain(Organ):
    def __init__(self, name, bear):
        if hasattr(Bear, 'rebels'):
            Bear.rebels.add(bear.name)
        else:
            Bear.rebels = set([bear.name])
        Bear.rebel = lambda self: Bear.rebels.add(self.name)
        Organ.__init__(self, name, bear)
```

Be careful not to reset `Bear.rebels` upon the creation of a second brain.

2 Clone Ant

In this problem, we will be implementing a new ant into Ants vs. Some Bees. The bees are winning too often, so deep in the research labs of CSM 61A, we've developed a new ant. called a Clone Ant. The more clone ants that exist, the more powerful each clone ant becomes! Specifically, the damage of each clone ant is equal to the total number of clone ants that exist in the world when the clone ant is created **including** the clone ant being created.

1. For this part, just implement up through the `__init__` method and include any class attributes that might be helpful. Assume each CloneAnt starts off with 2 health. **Note:** In the project, this would probably inherit from the Ant class, but let's make our own self-contained version!

CSM 61A Fall 2020: Jason Chang and Jade Singh, with
Catherine Cang, Alina Trinh, Ethan Chiu, Jamie Ip, Kenneth Chi, and
Aidan Tong, Aman Shah, Bridget Cheng, Cindy Lin, Ethan Yeh, Ivan Penev, Ivy Li, James Fletcher, Jemmy Zhou, Jennifer Huang, Jessica Yu, Joshua Baum, Kevin Moy, Kunal Agarwal, Lauren Meier, Matthew Guo, Mihira Patel, Nikhita Anumukonda, Owen Gong, Richard Roggenkemper, Sean Sananikone, Uma Unni

```
class CloneAnt:
"""
>>> ant1 = CloneAnt()
>>> ant2 = CloneAnt()
>>> print(ant1.damage, ant2.damage)
1 2
>>> ant1.health
2
>>> ant2.take_damage(1)
>>> ant2.health
1
>>> ant2.attack(ant1)
goodbye...
>>> ant3 = CloneAnt()
>>> print(ant3)
CloneAnt with 2 damage and 2 health
"""
    _____________________________________________________________

    def __init__(self):

        _____________________________________________________

        _____________________________________________________

        _____________________________________________________

    def attack(self, other):

        _____________________________________________________

    def take_damage(self, damage):

        _____________________________________________________

        _____________________________________________________

        _____________________________________________________


     # continued on next page
```

```
    def die(_____________):

        ____________________________________________________

        ____________________________________________________

    def __str__(self):

        ____________________________________________________
```

```
class CloneAnt:
    num_clones = 0

    def __init__(self):
        CloneAnt.num_clones += 1 # what if we replaced
           this with self.num_clones += 1?
        self.damage = CloneAnt.num_clones
        self.health = 2

    def attack(self, other):
        other.take_damage(self.damage)

    def take_damage(self, damage):
        self.health -= damage
        if self.health <= 0:
            self.die()

    def die(self):
        CloneAnt.num_clones -= 1
        print("goodbye...")

    def __str__(self):
        return "CloneAnt with {} damage and {} health".
           format(self.damage, self.health)
```

2. Nice! Now we can construct CloneAnts. Next, let's implement it so that our CloneAnt can interact with the world. Implement the `attack`, `take_damage`, and `die`methods accordingly in the code block from the previous part such that the doctests pass. Assume `other` has a `take_damage` method that you want to call in `attack`.

   Quick dot notation check! What's another way to write `clone1.die()`, where `clone`

is an instance of a `CloneAnt`?

```
CloneAnt.die(clone1)
```

3. Debugging has been super annoying for this problem - to check my CloneAnt instance's health and damage, I've had to print out `clone.health` and `clone.damage` each time! I sure wish I could just `print(clone)` instead...

   Implement the `__str__` method in the code from part (a) so that it returns `"CloneAnt with x damage and y health"` Where x and y are its respective damage and health values.

4. Uh oh, in our experimentation, we accidentally created a Mutant Clone Ant that, when attacked, loses attack damage rather than health! Mutant Clone Ants still are classified as Clone Ants, and have damage equal to the total amount of CloneAnts upon instantiation.. It should die when it has no more attack damage. Fill out the class below. Think about what methods we need to define, and which ones we don't.

```
class MutantCloneAnt(_____):
    """
    >>> ant = CloneAnt()
    >>> mutant = MutantCloneAnt()
    >>> print(ant.damage, mutant.damage)
    1 2
    >>> ant.attack(mutant)
    >>> mutant.damage
    1
    >>> mutant.health
    2
    >>> mutant2 = MutantCloneAnt() # We count total alive
       CloneAnts, including other Mutant ants
    >>> mutant2.damage
    3
    """

class MutantCloneAnt(CloneAnt):
    def take_damage(self, damage):
        self.attack -= damage
        if self.attack <= 0:
            self.die()
```

3 Ballot Rigging

1. **a. Ballot Rigger** In the 2020 CSM election, Jade's campaign advisors have decided to push back against the election fraud by rigging ballots themselves. Let's see how many ballots they were able to rig!

   Fill in the `Ballot` Class such that each ballot's `vote` attribute corresponds to the name of the person they vote for.

   In the ballot, bubble 'a' corresponds to 'Jade', 'b' corresponds to 'Jason', 'c' corresponds to 'Richard', and 'd' corresponds to 'Other'. Fill out the blanks above to categorize each voter's choice.

   Hint: It may be useful to let `choices` be a dictionary of bubble choices to names.

```
class Ballot:
    choices = _____________________________
    def __init__(self, name, bubble):
        self.name = name
        self.vote = _____________________________

california = [Ballot('kenny', 'c'), Ballot('alina', 'd'),
   Ballot('jamie', 'b')]
white_house = [Ballot('jade', 'a'), Ballot('jason', 'a'),
   Ballot('catherine', 'a')]
postman = []
postman.extend(white_house)
postman.append(california)
```

```
class Ballot:
    choices = {'a': 'Jade','b':'Jason', 'c':'Richard', 'd':'
       Other'}
    def __init__(self, name, bubble):
        self.name = name
        self.vote = Ballot.choices[bubble]
california = [Ballot('kenny', 'c'), Ballot('alina', 'd'),
   Ballot('jamie', 'b')]
white_house = [Ballot('jade', 'a'), Ballot('jason', 'a'),
   Ballot('catherine', 'a')]
postman = []
postman.extend(white_house)
postman.append(california)
```

b. Changing Ballots The Ballot Rigger takes all ballots and rigs them, changing votes for Jason and Other to Jade. Implement this in the lines of code above.

```
class BallotRigger:
    def rig(self, ballot):
        ____________________________________
            ____________________________________

bad_guy = ____________________
for ballot in california:
    bad_guy.____________________
for ballot in white_house:
    bad_guy.____________________
```

```
class BallotRigger:
    def rig(self, ballot):
        if ballot.vote == 'Jason' or ballot.vote == 'Other':
            ballot.vote = 'Jade'
bad_guy = BallotRigger()
for ballot in california:
    bad_guy.rig(ballot)
for ballot in white_house:
    bad_guy.rig(ballot)
```

c. Tallying Votes After the Ballot Rigger has rigged the ballots, return the number of votes for each candidate, implement the `count_votes` method, which adds votes to the count dictionary. *Hint:* Our ballot list can contain both Ballots and lists of Ballots! How can we deal with nested lists of ballots?

```
choices = {'Jade': 0 ,'Jason': 0, 'Richard': 0, 'Other': 0}
def count_votes(ballots):
    for b in ballots:
        if isinstance(_________,_________):
            ________________
        else:
            _____________ += 1

ballot_counter = postman[:]
count_ballots(ballot_counter)
```

```
choices = {'Jade': 0 ,'Jason': 0, 'Richard': 0, 'Other': 0}
def count_votes(ballots):
    for b in ballots:
        if isinstance(b, list):
            count_votes(b)
        else:
            choices[b.vote] += 1

ballot_counter = postman[:]
count_votes(ballot_counter)
```

d. Final Count Fill in the final tallies of votes for each candidate. *Hint:* It might help to draw box and pointer diagrams for `postman` abd `ballot_counter`.

```
choices = {'Jade': __ ,'Jason': __, 'Richard': __, 'Other': __
   }
```

```
choices = {'Jade': 5 ,'Jason': 0, 'Richard': 1, 'Other': 0}
```

4 Among Us

In this problem, we are going to be recreating (part of) Among Us!

1. First, let's implement the Crewmate class so that it satisfies the doctests below.

```
class Crewmate:
    """
    >>> pink = Crewmate("Pink")
    >>> lime = Crewmate("Lime")
    >>> pink.add_tasks(["clear asteroids", "fix wiring", "
       swipe card"])
    >>> print(pink)
    Pink crewmate has 3 tasks left.
    >>> lime.add_tasks(["fuel engines", "chart course", "prime
        shields"])
    >>> lime.do_task()
    >>> [lime.do_task() for _ in range(3)]
    AssertionError: Lime crewmate has no more tasks left!
    >>> lime.tasks
    ["chart course", "prime shields"]
    >>> pink.alive
    True
    >>> print(Crewmate.crew_size)
    2
    >>> another_pink = Crewmate("Pink")
    Cannot have two crewmates of the same color!
    >>> Crewmate.total_tasks
    3
    """
    crew_size = 0
    crew_colors = []
    total_tasks = 0

    def __init__(self, color):
        if _____________________________:
            _____________________________
            _____________________________
            _____________________________
            Crewmate.crew_size += ____________________
            Crewmate.crew_colors._____________(_____________)
        else:
            print("Cannot have two crewmates of the same color
```

```
                !")

    def add_tasks(self, tasks):
        _____________________________
        _____________________________

    def do_task(self):
        assert _____________________
        _____________________________
        _____________________________

    def __str__(self):
        return "{} crewmate has {} tasks left.".format(
          ____________________, ____________________)

class Crewmate:
    crew_size = 0
    crew_colors = []
    total_tasks = 0

    def __init__(self, color):
        if color not in Crewmate.crew_colors:
            self.color = color
            self.tasks = []
            self.alive = True
            Crewmate.crew_size += 1
            Crewmate.crew_colors.append(color)
        else:
            print("Cannot have two crewmates of the same color
               !")

    def add_tasks(self, tasks):
        self.tasks.extend(tasks)
        Crewmate.total_tasks += len(tasks)

    def do_task(self):
        assert len(self.tasks) > 0, "{} crewmate has no more
           tasks left!".format(self.color)
        self.tasks.pop(0)
        Crewmate.total_tasks -= 1

    def __str__(self):
```

```
        return "{} crewmate has {} tasks left.".format(self.
            color, len(self.tasks))
```

2. Now let's create the Impostor! The Impostor is a type of Crewmate so that he/she blends in, but they have certain extra powers. In addition to implementing the Impostor class, you'll also have to modify some parts of the Crewmate class to meet all the doctest requirements.

```
class Impostor(Crewmate):
    """
    >>> red = Impostor("Red")
    >>> yellow = Crewmate("Yellow")
    >>> purple = Crewmate("Purple")
    >>> cyan = Crewmate("Cyan")
    >>> cyan.add_tasks(["submit scan", "divert power", "unlock
        manifolds"])
    >>> red.add_tasks(["download data", "empty chute"])
    >>> print(Impostor.total_tasks)
    3
    >>> red.enable_sabotages()
    <generator ...>
    >>> red.kill(cyan)
    >>> print(cyan.alive)
    False
    >>> print(Crewmate.crew_size)
    3
    >>> red.sabotage()
    Comms sabotaged!
    >>> red.sabotage()
    Reactor meltdown!
    >>> red.sabotage()
    Oxygen depleted!
    >>> red.sabotage()
    Fix lights!
    >>> red.sabotage()
    No more sabotages left.
    """
    sabotages = ["Comms sabotaged!", "Reactor meltdown!", "
       Oxygen depleted!", "Fix lights!"]

    def __init__(self, color):
        _____________________________
```

```
        self.s = _____________________

    def kill(self, crewmate):
        _____________________________
        _____________________________

    def sabotage(self):
        try:
            next(self.s)
        _____________________________:
            print(_____________________________)

    def enable_sabotages(self):
        _____________________________

class Crewmate:
    . . .
    def add_tasks(self, tasks):
        self.tasks.extend(tasks)
        if _____________________________:
            Crewmate.total_tasks += len(tasks)

    def do_task(self):
        assert _____________________________
        if _____________________________:
            self.tasks.pop(0)
            Crewmate.total_tasks -= 1
    . . .

class Impostor(Crewmate):
    sabotages = ["Comms sabotaged!", "Reactor meltdown!", "
       Oxygen depleted!", "Fix lights!"]

    def __init__(self, color):
        super().__init__(color)
        self.s = self.enable_sabotages()

    def kill(self, crewmate):
        crewmate.alive = False
        Crewmate.crew_size -= 1

    def sabotage(self):
```

```
            try:
                next(self.s)
            except StopIteration:
                print("No more sabotages left.")

        def enable_sabotages(self):
            yield from Impostor.sabotages

class Crewmate:
    . . .
    def add_tasks(self, tasks):
        self.tasks.extend(tasks)
        if not isinstance(self, Impostor):
            Crewmate.total_tasks += len(tasks)

    def do_task(self):
        assert len(self.tasks) > 0, "{} crewmate has no more
           tasks left!".format(self.color)
        if not isinstance(self, Impostor):
            self.tasks.pop(0)
            Crewmate.total_tasks -= 1
    . . .
```

3. Finally, let's add a meeting functionality. At meetings, either a crewmate or the Impostor will be voted off. If a crewmate is voted off, the crew gets sad but must still continue their work. If the Impostor gets voted off, the crewmates win! Remember to modify the Crewmate class a bit to support meetings in addition to implementing the Meeting class.

```
class Meeting:
    """
    >>> brown = Crewmate("Brown")
    >>> black = Crewmate("Black")
    >>> green = Crewmate("Green")
    >>> white = Impostor("White")
    >>> m = Meeting("White")
    >>> brown.vote(m, green)
    >>> black.vote(m, green)
    >>> m.eject()
    Boo hoo
    >>> n = Meeting("White")
    >>> brown.vote(n, white)
```

```
        >>> black.vote(n, white)
        >>> white.vote(n, black)
        >>> n.eject()
        Woo hoo
        """
        def __init__(self, impostor):
            self.impostor = impostor
            self.votes = [[c, 0] for c in ____________________]

        def eject(self):
            sus = ___________(self.votes, key =
              ____________________)[0]
            if ___________________________:
                ___________________________ -= 1
                ___________________________.remove(sus)
                print("Boo hoo")
            else:
                print("Woo hoo")

    class Crewmate:
        . . .
        def vote(self, meeting, member):
            i = ___________________________.index(
              ___________________________)
            meeting.votes[i][1] += ___________________________
        . . .

    class Meeting:
        def __init__(self, impostor):
            self.impostor = impostor
            self.votes = [[c, 0] for c in Crewmate.crew_colors]

        def eject(self):
            sus = max(self.votes, key = lambda v : v[1])[0]
            if sus != self.impostor:
                Crewmate.crew_size -= 1
                Crewmate.crew_colors.remove(sus)
                print("Boo hoo")
            else:
                print("Woo hoo")

    class Crewmate:
```

```
. . .
def vote(self, meeting, member):
    i = Crewmate.crew_colors.index(member.color)
    meeting.votes[i][1] += 1
. . .
```