OOP

CS 61A Group Mentoring

July 19, 2017

1 OOP

1. **(H)OOP**
   Given the following code, what will Python output for the following prompts?

```
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
```

```
>>> alex = Baller('Alex', True)
>>> mitas = BallHog('Mitas')
>>> len(Baller.all_players)

>>> Baller.name

>>> len(mitas.all_players)
```

```
>>> alex.pass_ball()

>>> alex.pass_ball(mitas)

>>> alex.pass_ball(mitas)

>>> BallHog.pass_ball(mitas, alex)

>>> mitas.pass_ball(alex)

>>> mitas.pass_ball(mitas, alex)
```

2. Write `TeamBaller`, a subclass of `Baller`. An instance of `TeamBaller` cheers on the team every time it passes a ball.

```
class TeamBaller(_______________):
    """
    >>> mitas = BallHog('Mitas')
    >>> cheerballer = TeamBaller('Chris', has_ball=True)
    >>> cheerballer.pass_ball(mitas)
    Yay!
    True
    >>> cheerballer.pass_ball(mitas)
    I don't have the ball
    False
    """
    def pass_ball(_______________, ________________):
```

3. Lets 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):
        self.current = 0
        self.index = 1
        self.add = True
    def next(self):
```

4. **Flying the cOOP** What would Python display? Write the result of executing the code and the prompts below. If a function is returned, write ”Function”. If nothing is returned, write ”Nothing”. If an error occurs, write ”Error”.

```
class Bird:
    def __init__(self, call):
        self.call = call
        self.can_fly = True
    def fly(self):
        if self.can_fly:
            return "Don't stop me now!"
        else:
            return "Ground control to Major Tom..."
    def speak(self):
        print(self.call)

class Chicken(Bird):
    def speak(self, other):
        Bird.speak(self)
        other.speak()

class Penguin(Bird):
    can_fly = False
    def speak(self):
        call = "Ice to meet you"
        print(call)

andre = Chicken("cluck")
gunter = Penguin("noot")
```

```
>>> andre.speak(Bird("coo"))
```

```
>>> andre.speak()
```

```
>>> gunter.fly()
```

```
>>> andre.speak(gunter)
```

```
>>> Bird.speak(gunter)
```

2 Mutable Trees

Now that we know how to create objects using Python's class system, we have a new way of implementing some of the ADTs we saw earlier in the course. This allows us to reassign attributes of that object any time we want!

Here's an example of implementing trees using a class.

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

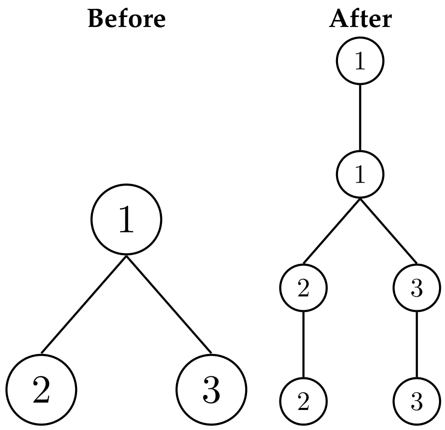
Here's how we might use this class:

```
>>> t = Tree(1, [Tree(2)])
>>> t.label
1
>>> t.label = 2
>>> t.label
2
>>> t.branches = t.branches + [Tree(3)]
>>> [b.label for b in t.branches]
[2, 3]
>>> t.branches[1].is_leaf()
True
```

5. Implement `tree_sum` which takes in a Tree object and replaces the root value with the sum of all the values in the tree. `tree_sum` should also return the new root value.

```
def tree_sum(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> tree_sum(t)
    10
    >>> t.label
    10
    >>> t.branches[0].label
    5
    >>> t.branches[1].label
    4
    """
```

6. DoubleTree hired you to architect one of their hotel expansions! As you might expect, their floor plan can be modeled as a tree and the expansion plan requires doubling each node (the patented double tree floor plan). Here's what some sample expansions look like:



Fill in the implementation for `double_tree`.

```
def double_tree(t):
    """
    Given a tree, mutate it such that each entry appears
    twice.
    >>> t = Tree(1)
    >>> double_tree(t)
    >>> t.label
    1
    >>> t.branches[0].label
    1
    """
```