OOP AND ORDERS OF GROWTH

COMPUTER SCIENCE MENTORS 61A

March 5 to March 7, 2018

Object Oriented Programming

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
```

```
>>> ajay = Baller('Ajay', True)
>>> surya = BallHog('Surya')
>>> len(Baller.all_players)

>>> Baller.name

>>> len(surya.all_players)


>>> ajay.pass_ball()

>>> ajay.pass_ball(surya)

>>> ajay.pass_ball(surya)

>>> BallHog.pass_ball(surya, ajay)

>>> surya.pass_ball(ajay)

>>> surya.pass_ball(surya, ajay)
```

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

```
class TeamBaller(________________):
    """
    >>> cheerballer = TeamBaller('Thomas', has_ball=True)
    >>> cheerballer.pass_ball(surya)
    Yay!
    True
    >>> cheerballer.pass_ball(surya)
    I don't have the ball
    False
    """
    def pass_ball(________________, ________________):
```

3. Last week you used nonlocal to implement pingpong; now, let's use OOP!

   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)
```

Orders of Growth

1. In big-Θ notation, what is the runtime for `foo`?
   (a)
```
def foo(n):
    for i in range(n):
        print('hello')
```

   (b) What's the runtime of `foo` if we change `range(n)`:

      i. To `range(n / 2)`?

      ii. To `range(10)`?

      iii. To `range(10000000)`?

2. What is the order of growth in time for the following functions? Use big-Θ notation.
   (a)
```
def strange_add(n):
    if n == 0:
        return 1
    else:
        return strange_add(n - 1) + strange_add(n - 1)
```

   (b)
```
def stranger_add(n):
    if n < 3:
        return n
    elif n % 3 ==  0:
        return stranger_add(n - 1) + stranger_add(n - 2) +
           stranger_add(n - 3)
    else:
        return n
```

(c)
```
def waffle(n):
    i = 0
    total = 0
    while i < n:
        for j in range(50 * n):
            total += 1
        i += 1
    return total
```

(d)
```
def belgian_waffle(n):
    i = 0
    total = 0
    while i < n:
        for j in range(n ** 2):
            total += 1
        i += 1
    return total
```

(e)
```
def pancake(n):
    if n == 0 or n == 1:
        return n
    # Flip will always perform three operations and return
       -n.
    return flip(n) + pancake(n - 1) + pancake(n - 2)
```

(f)
```
def toast(n):
    i = 0
    j = 0
    stack = 0
    while i < n:
        stack += pancake(n)
        i += 1
    while j < n:
        stack += 1
        j += 1
    return stack
```

3. Consider the following functions:

```
def hailstone(n):
    print(n)
    if n < 2:
        return
    if n % 2 == 0:
        hailstone(n // 2)
    else:
        hailstone((n * 3) + 1)

def fib(n):
   if n < 2:
      return n
   return fib(n - 1) + fib(n - 2)

def foo(n, f):
    return n + f(500)
```

In big-Θ notation, describe the runtime for the following with respect to the input n:

(a) `foo(10, hailstone)`

(b) `foo(3000, fib)`

4. **Orders of Growth and Trees:** Assume we are using the ADT tree implementation introduced in discussion. Consider the following function:

```
def word_finder(t, p, word):
    if label(t) == word:
        p -= 1
        if p == 0:
            return True
    for branch in branches(t):
        if word_finder(branch, p, word):
            return True
    return False
```

(a) What does this function do?

(b) If a tree has n total nodes, what is the worst case runtime in big-Θ notation?