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)
```

Solution: 2

```
>>> Baller.name
```

Solution: Error

```
>>> len(surya.all_players)
```

Solution: 2

```
>>> ajay.pass_ball()
```

Solution: Error

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

Solution: True

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

Solution: False

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

Solution: False

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

Solution: False

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

Solution: Error

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

Solution:

```
class TeamBaller(Baller):
    """
    >>> 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(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
```

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.

Solution:

```
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
```

Notice how the OOP approach is insanely similar to the non local function. Instead of using `nonlocal`, we use `self.varName` and the code becomes exactly the same. We just store the data in a slightly different way. This implies that OOP and functions are pretty similar, and it turns out you can even write your own OOP framework using just functions and `nonlocal`!

In addition, there are a lot of python specific features that can be written using functions or using classes. If you are interested, check out the powerful python feature decorators, and note how we can write them both as functions and as classes!

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"))
```

Solution: cluck
coo

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

Solution: Error

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

Solution: "Don't stop me now!"

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

Solution: cluck
Ice to meet you

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

Solution: noot

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')
```

   Solution: $\Theta(n)$. This is simple loop that will run n times.

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

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

      Solution: $\Theta(n)$. The loop runs $n/2$ times, but we ignore constant factors.

      ii. To `range(10)`?

      Solution: $\Theta(1)$. No matter the size of n, we will run the loop the same number of times.

      iii. To `range(10000000)`?

      Solution: $\Theta(1)$. No matter the size of n, we will run the loop the same number of times.

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)
```

   Solution: $\Theta(2^n)$. To see this, try drawing out the call tree. Each level will create two new calls to `strange_add`, and there are n levels. Therefore, 2^n calls.

   (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
```

Solution: $\Theta(n)$ is n is a multiple of 3, otherwise $\Theta(1)$.
The case where n is not a multiple of 3 is fairly obvious – we step into the else clause and immediately return.
If n is a multiple of 3, then neither $n-1$ nor $n-2$ are multiples of 3 so those calls will take constant time. Therefore, we just run `stranger_add`, decrementing the argument by 3 each time.

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

Solution: $\Theta(n^2)$. Ignore the constant term in $50 * n$, and it because just two for loops.

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

Solution: $\Theta(n^3)$. Inner loop runs n^2 times, and the outer loop runs n times. To get the total, multiply those together.

(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)
```

Solution: $\Theta(2^n)$. Flip will run in constant time. Therefore, this call tree looks very similar to fib! (which is 2^n)

(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
```

Solution: $\Theta(n2^n)$. There are two loops: the first runs n times for 2^n calls each time (due to pancake), for a total of $n2^n$. The second loop runs n times. When calculating orders of growth however, we focus on the dominating term – in this case, $n2^n$.

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)`

Solution: $\Theta(1)$. $f(500)$ is independent of the size of the input n.

(b) `foo(3000, fib)`

Solution: $\Theta(1)$. See above.

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?

Solution: This function take a Tree `t`, an integer `p`, and a string `word` in as input.

Then, `word_finder` returns True if any paths from the root towards the leaves have at least `p` occurrences of the word and False otherwise.

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

Solution: $\Theta(n)$. At worst, we must visit every node of the tree.