OBJECT ORIENTED PROGRAMMING, INHERITANCE, AND REPRESENTATION

COMPUTER SCIENCE MENTORS

October 19 to October 22, 2020

1   Object Oriented Programming

```python
class Car:
    wheels = 4
    def __init__(self):
        self.gas = 100

    def drive(self):
        self.gas -= 10
        print("Current gas level:", self.gas)

my_car = Car()
```

Dot Notation

Dot notation with an instance before the dot automatically supplies the first argument to a method.

```
>>> my_car.drive()
Current gas level: 90
```

We don't have to explicitly pass in a parameter for the `self` argument of the `drive` method as the instance to the left of the dot (the `my_car` object of the `Car` class) is automatically passed into the first parameter of the method by Python. So, what is `self`? By convention, we name the first argument of any method in any class "self" so the `self` you see as the arguments in all the methods will refer to the object that called this method. Note that Python does not enforce this, so you could name the first parameter anything you wanted; but it is best practice to name it `self`.

There is another way of calling a method:

```
>>> Car.drive(my_car)
Current gas level: 80
```

In this case, the thing to the left of the dot is a class itself and not an instance of a class so Python will not automatically use the item on the left as the first argument of the method. Therefore, we have to explicitly pass in an object for `self` which is why we wrote `my_car` in the parentheses as the argument to `self`.

The `__init__` Method

The `__init__` method of a class, which we call the constructor, is a special method that creates a new instance of that class. In our code above, `Car()` makes a new instance of the Car class because Python automatically calls the `__init__` method when it sees a "call" to that class (the class name followed by parentheses that can contain arguments if the `__init__` method takes in arguments). If the `__init__` method takes in only the `self` argument, nothing needs to be passed in to the constructor.

Instance Attributes and Class Attributes

In the example above, the **class attribute** `wheels` is shared by all instances of the Car class; while `gas` is an **instance attribute** that's specific to the instance `my_car`. In this case, `my_car.wheels` and `Car.wheels` both return the value 4. The reason is that the order for looking up an attribute is: instance attributes -> class attributes/methods -> parent class attributes/methods.

Inheritance Overview

Inheritance is the idea that not all the methods or attributes of a class need to be specified in that SPECIFIC class. Instead they can be inherited, like if a class is a subgroup of another class. For example, we can have a `Marker` class and also a `DryEraseMarker` class. In this case, we can use inheritance to convey that a `DryEraseMarker` is a specialized version of a `Marker`. This avoids rewriting large blocks of code and gives us a nice hierarchy to understand how our classes interact with each other.

You include the class you inherit from in the class definition (**class** `SubClass(SuperClass)`). The subclass can inherit any methods, including the constructor from the superclass. You also inherit class attributes of the superclass.

You can call the constructor or any othe method of the superclass with the code `SuperClass.__init__(<whatever parameters are required>)` if you want the same constructor but with some additional information. All methods and class attributes can be overridden in the subclass, by simply creating an attribute or method with the same name.

1. **(H)OOP**

   Given the following code, what will Python output for the following prompts?

   ```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
   ```

   ```python
   >>> alina = Baller('Alina', True)
   >>> kenny = BallHog('Kenny')
   >>> len(Baller.all_players)


   >>> Baller.name


   >>> len(kenny.all_players)



   >>> alina.pass_ball()


   >>> alina.pass_ball(kenny)


   >>> alina.pass_ball(kenny)


   >>> BallHog.pass_ball(kenny, alina)


   >>> kenny.pass_ball(alina)


   >>> kenny.pass_ball(kenny, alina)
   ```

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

```python
class TeamBaller(_____):
    """
    >>> jamie = BallHog('Jamie')
    >>> cheerballer = TeamBaller('Ethan', has_ball=True)
    >>> cheerballer.pass_ball(jamie)
    Yay!
    True
    >>> cheerballer.pass_ball(jamie)
    I don't have the ball
    False
    """
    def pass_ball(_____, _____):
```

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 8 or contains the digit 8.

   The first 30 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 8th, 16th, 18th, 24th, and 28th elements:
   ```
   1 2 3 4 5 6 7 [8] 7 6 5 4 3 2 1 [0] 1 [2] 1 0 −1 −2 −3 [−4] −3
       −2 −1 [0] −1 −2
   ```

   Assume you have a function has_eight(k) that returns True if k contains the digit 8.

   ```
   >>> tracker1 = PingPongTracker()
   >>> tracker2 = PingPongTracker()
   >>> tracker1.next()
   1
   >>> tracker1.next()
   2
   >>> tracker2.next()
   1
   ```

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




       def next(self):
   ```

Representation Overview: `__repr__` and `__str__`

The goal of `__str__` is to convert an object to a human-readable string. The `__str__` function is helpful for printing objects and giving us information that's more readable than `__repr__`. Whenever we call **print**() on an object, it will call the `__str__` method of that object and print whatever value the `__str__` call returned. For example, if we had a `Person` class with a name instance variable, we can create a `__str__` method like this:

```python
def __str__(self):
    return "Hello, my name is " + self.name
```

This `__str__` method gives us readable information: the person's name. Now, when we call print on a person, the following will happen:

```python
>>> p = Person("John Denero")
>>> str(p)
'Hello, my name is John Denero'
>>> print(p)
Hello, my name is John Denero
```

The `__repr__` magic method of objects returns the "official" string representation of an object. You can invoke it directly by calling **repr**(<some **object**>). However, `__repr__` doesn't always return something that is easily readable, that is what `__str__` is for. Rather, `__repr__` ensures that all information about the object is present in the representation. When you ask Python to represent an object in the Python interpreter, it will automatically call **repr** on that object and then print out the string that **repr** returns. If we were to continue our `Person` example from above, let's say that we added a **repr** method:

```python
def __repr__(self):
    return "Name: " + self.name
```

Then we can write the following code:

```python
# Python calls this object's repr function to see what
# to print on the line. Note, Python prints whatever
# result it gets from repr so it removes the quotes
# from the string
>>> p
Name: John Denero

# User is invoking the repr function directly.
# Since the function returns a string, its output
```

```
# has quotes. In the previous line, Python called
# repr and then printed the value. This line works
# like a regular function call: if a function
# returns a string, output that string with quotes.
>>> repr(p)
"Name: John Denero"
```

4. 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 "Noth-
   ing". If an error occurs, write "Error".

```
class Musician:
    popularity = 0
    def __init__(self, instrument):
        self.instrument = instrument
    def perform(self):
        print("a rousing " + self.instrument + " performance")
        self.popularity = self.popularity + 2
    def __repr__(self):
        return self.instrument


class BandLeader(Musician):
    def __init__(self):
        self.band = []
    def recruit(self, musician):
        self.band.append(musician)
    def perform(self, song):
        for m in self.band:
            m.perform()
        Musician.popularity += 1
        print(song)
    def __str__(self):
        return "Here's the band!"
    def __repr__(self):
        band = ""
        for m in self.band:
            band += str(m) + " "
        return band[:-1]


miles = Musician("trumpet")
goodman = Musician("clarinet")
ellington = BandLeader()
```

```
>>> ellington.recruit(goodman)
>>> ellington.perform()


>>> ellington.perform("sing, sing, sing")




>>> goodman.popularity, miles.popularity


>>> ellington.recruit(miles)
>>> ellington.perform("caravan")



>>> ellington.popularity, goodman.popularity, miles.popularity



>>> print(ellington)



>>> ellington
```

5. **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".

```python
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")
```

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

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

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

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

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