Object Oriented Programming, Inheritance, and Representation Guide

Computer Science Mentors

October 19 to October 22, 2020

Recommended Timeline for Non-NPE Sections

- OOP mini lecture - 5 minutes
- (H)OOP - 5 minutes
- Inheritance mini lecture - 5 minutes
- Team Baller - 7 minutes
- Pingpong - 8 minutes
- Representation mini lecture - 6 minutes
- Musician - 10 minutes

Recommended Timeline for NPE Sections

- Section 1:
  - OOP mini lecture - 6 minutes
  - (H)OOP - 6 minutes
  - Inheritance mini lecture - 6 minutes
  - Team Baller - 8 minutes
  - Pingpong - 10 minutes
- Section 2:
  - OOP and Inheritance review + Representation mini lecture - 2 minutes
  - Musician - 10 minutes
  - Flying the cOOP - 8 minutes

1 Object Oriented Programming

```
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 dont 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** thats 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?

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

   2

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

   Error

```
>>> len(kenny.all_players)
```

   2

```
>>> alina.pass_ball()
```

Error

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

True

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

False

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

False

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

False

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

Error

Teaching Tips

- Create a separate box for each class and instance, adding variables to them as you go line by line through the code
- Emphasize the distinction between class and instance attributes. (Class attributes are features shared by the entire class)
- Emphasize the ability of `BallHog` to override the `pass_ball` function in `Baller`
- It's important to note the difference including a class or instance before the dot in dot notation. Which method do we call in each scenario?

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

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

Teaching Tips

- Remember that `pass_ball` that we're defining should have the same functionality as in the Baller class, but with slight modification.
- Ask students what should happen if the ball doesn't get passed (this should hint to them to somehow use the boolean retyrn type of `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):
```

```
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_eight(self.index) or self.index % 8 == 0:
            self.add = not self.add
        self.index += 1
        return self.current
```

Teaching Tips

- Describe instance independence when going through doctests- why does `tracker2` not update when `tracker1` updates?
- Ask students what variables a tracker needs to use and how they should be initialized in `__init__`
- If you have time, compare and contrast the nonlocal vs. object oriented approach
  - Emphasize how nonlocal and OOP both save the "state" of something, but note the pros/cons of both
- Make sure students don't confuse this **`next`** with the built-in iterator function **`next`**

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:

```
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:

```
>>> 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:

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

Then we can write the following code:

```
# 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"
```

Teaching Tips

- `__repr__` and `__str__` error if they do not return strings
- Writing the correct quotation marks for **`str`** and **`repr`** results is required on midterms and can be confusing for students
  - `string` leaves quotation marks
    - `"test"= 'test'`
  - **`print`** removes quotation marks
    - **`print`**`("test")= test`
  - **`repr`** adds quotation marks
    - **`repr`**`("test")= "'test'"`
  - **`str`** leaves quotation marks
    - **`str`**`("test")= 'test'`
  - **`repr`** and **`str`** both add quotation marks to numbers or booleans
    - **`repr`**`(3)= '3'`

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 "Nothing". 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()
```

Some Quick Refreshers

Defining attributes: Instance attributes are defined with the `self.attr_name` notation (usually in `__init__` but could be elsewhere like in this problem). Class attributes are defined outside of methods in the body of the class definition, like the variable `popularity` in the class `Musician`.

Accessing attributes: Instance attributes are referred to using `self.attr_name` Class attributes can be referred to using `classname.attr_name` or `self.attr_name` (Note: using the latter will only work if there are no instance attributes bound with the name `attr_name`).

Before running any of the code below, `miles` and `goodman` are set to the musicians created as a result of calling the `__init__` constructor method in `Musician`. `ellington` uses `BandLeader's __init__` method, since `BandLeader` is the subclass and has `__init__` defined.

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

Error

`ellington.recruit(goodman)` adds `goodman` to the end of `ellington's` instance attribute, `band`. Then, `ellington` checks its class (`BandLeader`) for the `perform()` method. But this `perform()` is expecting an argument, so this errors.

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

a rousing clarinet performance
sing, sing, sing

Using the same `perform()` method, now providing the correct number of arguments. First, going through the band list, `goodman` calls its `perform()` method, which is defined in `Musician`. Here, we print `"a rousing" + goodman`'s instrument `+ " performance"`, and then `goodman`'s `self.popularity = self.popularity + 2` happens. The `self.popularity` on the right of the equal sign is `Musician.popularity` because `goodman` doesn't have its own instance attribute named `popularity` yet; then it becomes `self.popularity = 0 + 2`, and this creates the instance attribute `popularity` for `goodman`. Then `Musician.popularity`, the class attribute, in incremented by 1.

```
>>> goodman.popularity, miles.popularity
```

(2, 1)

First, we try to get the value of `goodman.popularity`. In our environment diagram, we see that `goodman` has the instance variable `popularity` already defined. Therefore, we get that value, 2, back. Then, we try to access `miles.popularity`. In this case, `miles` doesnt have a `popularity` instance variable defined, so we default to the class variable. There, we see it defined as 1, so we get that value. Finally, since commas in Python define a tuple, we return the two values as `(2, 1)`.

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

a rousing clarinet performance
a rousing trumpet performance
caravan

First, we call `ellington.recruit(miles)`. This appends `miles` to `ellington`'s instance variable, `band`. After that, we call `ellington.perform("caravan")`. Similar to the previous call on perform, we will loop through all of the values in `ellington.band`, calling their perform methods in order. This causes the first two lines to be printed. Next, we increment `Musician.popularity` (the class variable of `Musician` called `popularity`). Lastly, we print the `song` variable that was passed in, completing the last line.

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

(2, 4, 3)

```
>>> print(ellington)
```

Here's the band!

`print``()` expects the string representation of `ellington`, which is given by calling the `__str__()` method of `ellington`. `ellington` checks to see if `BandLeader` has a `__str__()` method, which it does. So, **`print`**`(ellington)` then becomes **`print`**`("Here's the band!")`.

```
>>> ellington
```

clarinet trumpet

When prompting for `ellingtons` value, we return the representation of ellington given by `__repr__()`. So, we call `BandLeaders` `__repr__()` method.

Teaching Tips

- Draw a very thorough OOP diagram for tracking class and instance attributes.
  - Create a separate box for each class and instance, adding variables to them as you go line by line through the code
  - Emphasize to students that drawing the diagram carefully, before even starting to answer questions, is very important
- Use this problem to help students understand repr vs str

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".

```
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"))
```
cluck
coo

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

```
>>> gunter.fly()
```
"Don't stop me now!"

```
>>> andre.speak(gunter)
```
cluck
Ice to meet you

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

Teaching Tips

- Consider giving a mini lecture on inheritance before doing this question
- It may be helpful to use a live environment diagram of the execution
- Emphasize the order of variable lookup
  - When looking for a variable, go from instance to class to parent until failure
- Emphasize the different ways to pass in `self` for any function, either using dot notation or passing it in as a parameter
  - For `Bird.speak(gunter)`, `Bird` is not passed in for `self`, since `Bird` is a class, not an instance
- For `andre.speak(Bird("coo"))`, an "unnamed" `Bird` object is created and passed in as a parameter
  - Make sure students understand how this works- it can help to draw a temporary box for this `Bird` object in your OOP diagram and erase it afterwards