OOP, INHERITANCE, AND ITERATORS & GENERATORS

CS 61A

July 21, 2021

1 OOP

```
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.

1. What would Python display? Write the result of executing the following code and prompts. If nothing would happen, write "Nothing". If an error occurs, write "Error".

```
class Jedi:
    lightsaber = "blue"
    force = 25
    def __init__(self, name):
        self.name = name
    def train(self, other):
        other.force += self.force / 5
    def __repr__(self):
        return "Jedi " + self.name
```

```
>>> anakin = Jedi("Anakin")
>>> anakin.lightsaber, anakin.force
```

```
>>> anakin.lightsaber = "red"
>>> anakin.lightsaber
```

```
>>> Jedi.lightsaber
```

```
>>> obiwan = Jedi("Obi-wan")
>>> anakin.master = obiwan
>>> anakin.master
```

```
>>> Jedi.master
```

```
>>> obiwan.force += anakin.force
>>> obiwan.force, anakin.force
```

```
>>> obiwan.train(anakin)
>>> obiwan.force, anakin.force
```

```
>>> Jedi.train(obiwan, anakin)
>>> obiwan.force, anakin.force
```

2. Let's build a Bear using OOP!

   Bear instances should have an attribute `name` that holds the name of the bear. The Bear class should have an attribute `bears`, a list that stores the name of each bear.

```
>>> oski = Bear('Oski')
>>> oski.name
'Oski'
>>> Bear.bears
['Oski']
>>> winnie = Bear('Winnie')
>>> Bear.bears
['Oski', 'Winnie']
```

```
class Bear:
```

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):
```

2 Inheritance

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
```

```
>>> catherine = Baller('Catherine', True)
>>> albert = BallHog('Albert')
>>> len(Baller.all_players)
```

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

```
>>> len(albert.all_players)
```

```
>>> catherine.pass_ball()


>>> catherine.pass_ball(albert)


>>> catherine.pass_ball(albert)


>>> BallHog.pass_ball(albert, catherine)


>>> albert.pass_ball(catherine)


>>> albert.pass_ball(albert, catherine)
```

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

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

3 Iterators & Generators

An **iterable** is any container that can be processed sequentially. Think of an iterable as anything you can loop over, such as lists or strings. You can see this in `for` loops, which sequentially loop through each element of a sequence. The anatomy of the for loop can be described as:

```
for some_var in iterable:
    <do something with some_var>
```

An **iterator** remembers where it is during its iteration. Though an iterator is an iterable, the reverse is not necessarily true. Think of an iterable as a book whereas an iterator is a bookmark.

Generators, which are a specific type of **iterators**, are created using the traditional function definition syntax in Python (`def`) with the body of the function containing one or more `yield` statements. When a generator (a function that has `yield` in the body) is called, it returns a generator object. When we call the generator object, we evaluate the body of the function until we have yielded a value. The `yield` statement pauses the function, yields the value, saves the local state so that evaluation can be resumed right where it left off. `yield` operates similarly to a return statement.

1. Write a generator that will take in two iterators and compares the first element of each iterator and yields the smaller of the two values.

```
def interleave(iter1, iter2):
    """
    >>> gen = interleave(iter([1, 3, 5, 7, 9]),
                         iter([2, 4, 6, 8, 10]))
    >>> for elem in gen:
    ...     print(elem)
    ...
    1
    2
    3
    4
    5
    6
    7
    8
    9
    """
```

2. (a) Implement `n_apply`, which takes in 3 inputs `f`, `n`, `x`, and outputs the result of applying `f`, a function, `n` times to `x`. For example, for `n = 3`, output the result of `f(f(f(x)))`.

```
def n_apply(f, n, x):
    """
    >>> n_apply(lambda x: x + 1, 3, 2)
    5
    """

    for ______________________________:

        x = ____________________________

    return ___________________________
```

(b) Now implement `list_gen`, which takes in some list of integers `lst` and a `function f`. For the element at index `i` of `lst`, `list_gen` should apply `f` to the element `i` times and yield this value `lst[i]` times. You may use `n_apply` from the previous part.

```
def list_gen(lst, f):
    """
    >>> a = list_gen([1, 2, 3], lambda x: x + 1)
    >>> list(a)
    [1, 3, 3, 5, 5, 5]
    """

    for ______________________________:

        yield from [______________________________________________]
```

3. Define `filter_gen`, a generator that takes in iterable `s` and one-argument function `f` and yields every value from `s` for which `f` returns a truthy value.

```
def filter_gen(s, f):
    """
    >>> list(filter_gen([1, 2, 3, 4, 5],
                                    lambda x: x % 2 == 0))
    [2, 4]
    >>> list(filter_gen((1, 2, 3, 4, 5), lambda x: x < 3))
    [1, 2]
    """
```

4. Write a generator function `in_order` that takes in a possibly nested list of integers `lst` and yields its integer elements in ascending order as a single non-nested list. You may find the built-in **`sorted`** function useful, which takes in a list of *integers* and returns a sorted list.

```
def in_order(lst):
    """
    >>> l1 = [[3, 4, 2], [1, 7, 4]]
    >>> list(in_order(l1))
    [1, 2, 3, 4, 4, 7]
    >>> l2 = [2, [3], [1, [8], 4]]
    >>> list(in_order(l2))
    [1, 2, 3, 4, 8]
    """
    order = []

    for ___________________________:

        if ________________________________:

            ________________________________________
        else:

            ________________________________________

    ________________________________________
```