Inheritance, Representation, and Efficiency

Computer Science Mentors 61A

March 20, 2023 - March 24, 2023

1 Inheritance

Inheritance is an important feature of object oriented programming. To create an object that shares its attributes or methods with an existing object, we can have the object inherit these similarities instead of repeating code. In addition to making our code more concise, it allows us to create classes based on other classes, similar to how real-world categories are often divided into smaller subcategories.

For example, say the `HybridCar` class inherits from the `Car` class as a type of car:

```python
class HybridCar(Car):
    def __init__(self):
        super().__init__()
        self.battery = 100

    def drive(self):
        super().drive()
        self.battery -= 5
        print("Current battery level:", self.gas)

    def brake(self):
        self.battery += 1

my_hybrid = HybridCar()
```

By default, the child class inherits all of the attributes and methods of its parent class. Consequently, we would be able to call `my_hybrid.drive()` and access `my_hybrid.wheels` from the `HybridCar` instance `my_hybrid`. When dot notation is used on an instance, Python will first check the instance to see if the attribute exists, then the instance's class, and then its parent class, etc. If Python goes all the way up the class tree without finding the attribute, an `AttributeError` is thrown.

Additional or redefined instance and class attributes can be added in a child class, such as `battery`. If we decided that hybrid cars should have 3 wheels, we could assign 3 to a class attribute `wheels` in `HybridCar`. `my_hybrid.wheels` would return 3, but `my_car.wheels` would still return 4. We can also **override** inherited instance methods by redefining them in the child class. If we would like to call the parent class's version of a method, we can use **super**() to access it.

1. **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".

   Hint: You may find it helpful to make an environment diagram tracking your objects and *each* new instance of an object (whether it's assigned to a variable or not!).

```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 see you"
        print(call)

andre = Chicken("cluck")
gunter = Penguin("noot")
```

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

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

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

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

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

2. What would Python display? The questions continue on the next page.

```python
class Food:
    def __init__(self, name, spoiled = False):
        self.name = name
        self.num_days = 0
        self.spoiled = spoiled

    def can_eat(self):
        self.num_days += 1
        if self.num_days >= 3:
            self.spoiled = True
            print("Oh no! Your food is spoiled!")
        return not self.spoiled

    def mix_food(self, other_food):
        self.num_days = self.num_days + other_food.num_days
        self.name += " " + other_food.name
        self.spoiled = self.spoiled and other_food.spoiled

class Salad(Food):
    def __init__(self, ingredients):
        super().__init__("salad", False)
        self.ingredients = ingredients

    def add_ingredients(self, ingredient):
        self.ingredients.append(ingredient)
        print(ingredient.name + " has been added")

    def mix_ingredients(self):
        for ingredient in self.ingredients:
            self.mix_food(ingredient)
        print("Your salad has been mixed.")

lettuce = Food("lettuce")
tomatoes = Food("tomatoes")
chicken = Food("chicken")
ingredients = [lettuce, tomatoes]
my_salad = Salad(ingredients)
```

```
>>> lettuce.can_eat()
```

```
>>> my_salad.can_eat()
```

```
>>> my_salad.mix_ingredients()


>>> my_salad.name
```

__**str**__ is special method that converts an object to a string meant to be readable by humans. It may be invoked by directly calling **str** on an object. Additionally, calling **print**() on an object will call the __str__ method of that object and print whatever value the __str__ call returns.

The __**repr**__ method also returns a string representation of an object. However, the representation created by **repr** is meant to be read by the Python interpreter, not by humans. When we evaluate some object in the Python interpreter, it will automatically call **repr** on that object and then print out the string that **repr** returns. It should contain all information about the object.

For example, if we had a Person class with a name instance variable, we can create a __repr__ and __str__ method like so:

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

def __repr__(self):
    return f"Person({repr(self.name)})"

>>> nobel_laureate = Person("Carolyn Bertozzi")
>>> str(nobel_laureate)
'Hello, my name is Carolyn Bertozzi'

>>> print(nobel_laureate)
Hello, my name is Carolyn Bertozzi

>>> repr(nobel_laureate)
'Person("Carolyn Bertozzi")'

>>> nobel_laureate
Person("Carolyn Bertozzi")

>>> [nobel_laureate]
[Person("Carolyn Bertozzi")]
```

(In an **f-string**, which is a string with an f in front of it, the expressions in curly braces are evaluated and their values [converted into strings] are inserted into the f-string, allowing us to customize the f-string based on what the expressions evaluate to.)

__str__, __repr__, and __init__ are a just a few examples of double-underscored "magic" methods that implement all sorts of special built-in and syntactical features of Python.

1. **Musician** 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 Musician:
    popularity = 0
    def __init__(self, instrument):
        self.instrument = instrument
    def perform(self):
        print("a stellar " + 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
```

2. Let's slowly build a Bear from start to finish using OOP!

(a) First, let's build a `Bear` class for our basic bear. Bear instances should have an attribute `name` that holds the name of the bear and an attribute `organs`, an initially empty list of the bear's organs. The Bear class should have an attribute `bears`, a list that stores the name of each bear.

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

(b) Next, let's build an `Organ` class to put in our bear. `Organ` instances should have an attribute `name` that holds the name of the organ and an attribute `bear` that holds the bear it belongs to. The `Organ` class should also have an instance method `discard(self)` that removes the organ from `Organ.organ_count` and the bear's organs list.

The `Organ` class should contain a dictionary `organ_count` that maps the name of each bear to the number of organs it has.

Hint: We may need to change the representation of this object for our doc tests to be correct.

```python
class Organ:
    """
    >>> oski, winnie = Bear('Oski'), Bear('Winnie')
    >>> oski_liver = Organ('liver', oski)
    >>> Organ.organ_counts
    {'Oski': 1}
    >>> winnie_stomach = Organ('stomach', winnie)
    >>> winnie_liver = Organ('liver', winnie)
    >>> winnie.organs
    [stomach, liver]
    >>> winnie_liver.discard()
    >>> Organ.organ_counts
    {'Oski': 1, 'Winnie': 1}
    >>> winnie.organs
    [stomach]
    """
```

(c) Now, let's design a `Heart` class that inherits from the `Organ` class. When a heart is created, if its bear does not already have a heart, it creates a `heart` attribute for that bear. If a bear already has a heart, the old `heart` is discarded and replaced with the new one. The bear's organs list and `Organ.organ_count` should be updated appropriately.

Hint: you can use **hasattr** to check if a bear has a heart attribute.

```
class Heart(Organ):
    """
    >>> oski, winnie = Bear('Oski'), Bear('Winnie')
    >>> hasattr(oski, 'heart')
    False
    >>> oski_heart = Heart('small heart', oski)
    >>> oski.heart
    small heart
    >>> oski.organs
    [small heart]
    >>> new_heart = Heart('big heart', oski)
    >>> oski.heart
    big heart
    >>> oski.organs
    [big heart]
    >>> Organ.organ_counts["Oski"]
    1
    """
```

3   Efficiency

An order of growth (OOG) characterizes the runtime **efficiency** of a program as its input becomes extremely large. Since we care about rate of growth, we ignore constant coefficients and exclusively consider the fastest growing term. For example, on very large inputs, $2n^2 + 3n - 20$ behaves the same as n^2. Common runtimes, in increasing order of time, are: constant, logarithmic, linear, quadratic, and exponential.

Examples:

Constant time means that no matter the size of the input, the runtime of your program is consistent. In the function `f` below, no matter what you pass in for `n`, the runtime is the same.

```python
def f(n):
    return 1 + 2
```

A common example of a linear OOG involves a single for/while loop. In the example below, as `n` gets larger, the amount of time to run the function grows proportionally.

```python
def f(n):
    while n > 0:
        print(n)
        n -= 1
```

We can modify this while loop to get an example of logarithmic OOG. Suppose that, instead of subtracting 1 each time, we halve the size of `n`. For `n = 1000`, the program would take 10 iterations to terminate (since $2^{10} = 1024$). The runtime is propotional to $\log(n)$.

```python
def f(n):
    while n > 0:
        print(n)
        n //= 2
```

An example of a quadratic runtime involves nested for loops. For every one of the `n` iterations of the outer loop, there is `n` work done in the inner loop. This means that the runtime is proportional to n^2.

```python
def f(n):
    for i in range(n):
        for j in range(n):
            print(i*j)
```

1. What is the order of growth in time for the following functions? Use big-Θ notation.

   (a) ```
   def belgian_waffle(n):
       i = 0
       total = 0
       while i < n:
           for j in range(50 * n ** 2):
               total += 1
           i += 1
       return total
   ```

   (b) ```
   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 - 1)
   ```

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

2. Consider the following functions:

```python
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(n, hailstone)`

(b) `foo(n, fib)`