Inheritance, Representation, and Efficiency

Computer Science Mentors 61A

March 20, 2022 - March 24, 2022

1 Inheritance

Inheritance is an important feature of object oriented programs. In addition to making our code more concise, it allows us to create classes based on other classes in a similar way to how real-world categories are often divided into smaller subcategories.

For example, the `HybridCar` class may inherit from the `Car` class:

```
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. So from the `HybridCar` instance `my_hybrid`, we can call `my_hybrid.drive()` and access `my_hybrid.wheels`, for example. 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.

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

Additional or redefined instance and class attributes can be added in a child class. 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".

```
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
```

2 String and Representation

__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:

```
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”.

```
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
```

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.

```
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

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

(a)
```python
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)
```python
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)
```python
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.
```python
def flip(n):
    return -n

def pancake(n):
    if n < 1:
        return 1
    return flip(n) + pancake(n - 1) + pancake(n - 1)
```

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(n, hailstone)`

(b) `foo(n, fib)`