INHERITANCE, REPRESENTATION, AND EFFICIENCY Meta

COMPUTER SCIENCE MENTORS 61A

March 20, 2023 - March 24, 2023

Recommended Timeline

- Inheritance
  - Flying the cOOP: 8 min
  - Food: 8 min
- String and Representation
  - String and Representation minilecture: 5 min
  - Musician: 12 min
  - Build-a-Bear: 15 min
- Efficiency
  - Medley: 10 min
  - Hailstone (again!): 8 min

This worksheet includes a very large number of problems because we essentially intend it to be a problem bank for any review activities you decide to do in section. You, of course, are free to ignore any and all of the problems we provide you and simply do exam problems, which is honestly what I would probably do if I were teaching a section this week. It's never too early to start preparing!
Also, note that the inheritance overview at the start of the worksheet is redundant with last week's. We included this as a refresher!

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:

```
class HybridCar(Car):
    def __init__(self):
        super().__init__()
```

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

```
        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.

NOTE: AS OF THE SPRING 2023 ITERATION OF THIS COURSE, IT IS DEFINITELY POSSIBLE THAT THIS IS YOUR STUDENTS' FIRST INTERACTION WITH INHERITANCE. We included this section as, honestly, there's not much you can do with OOP without the concepts of inheritance and representation, so as such, we included some baseline examples of such in the problems following this overview.

Again, you probably want to go over this differently than the reference material presented here. I like to draw out a class tree on the board and emphasize that there should be an "is-a" relationship between child class and parent class. For example, a hybrid car "is a" car. The reasoning behind this "is-a" rule of thumb is that objects of the child class should generally have all the same properties as objects of the parent class. It's also often instructive to give some examples that do not work in a class hierarchy. A wagon is not a car. A vehicle is not a car (but a car is a vehicle). A car is not a garage (although a car is contained in a garage).

Variable look-up can be rather confusing for students. If you draw the class hierarchy as a tree on the board, you can demonstrate the process of successively looking up from instance to class and then from child class to parent class until you find the attribute or error out. I tell my students that you can only look up the class hierarchy, not down 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!).

```
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"))
```

cluck
coo

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

Error

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

”Don’t stop me now!”

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

cluck
Ice to see you

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

noot

Explanations

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

The `Bird` object is created and immediately passed in as the parameter for `Bird`. Even though we don't assign it to a variable, the object still exists and has all the features of a `Bird` object.

This might be difficult to conceptualize to students, so stop to make sure students understand how this works, since this same create-and-use method will be used often in midterm / exam level environment diagram questions.

- `andre.speak()`

Python expects two parameters but in this case we are only assigning `self`.

It might be good to emphasize to students that this is how regular functions work as well. We have to pass in the correct amount of values.

- `gunter.fly()`

Note that the `Penguin` class will use the constructor for the `Bird` class, which sets `gunter.can_fly` for the particular instance.

Step slowly through this part to emphasize how the `Penguin` class variable differs from the `gunter` instance variable for `can_fly`.

- `andre.speak(gunter)`

This question is really similar to the first one, but instead of `Bird("coo")` we use the `gunter` object instead.

- `Bird.speak(gunter)`

`Bird.speak` looks within the `Bird` class to find the speak method.

Some students may think that `Bird` is passed in for `self`. This is not the case because `Bird` is not an instance; it's a class.

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

```
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
```

```python
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)
```

See visualizations for solutions: `https://docs.google.com/presentation/d/1t1yE9DuT8a2ij_QszLOxzUu6-unN46PY1SA_Q48fLz4/edit?usp=sharing`

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

True

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

True

```
>>> my_salad.mix_ingredients()
```

Your salad has been mixed.

```
>>> my_salad.name
```

"salad lettuce tomatoes"

Teaching Tips

- Note that the use of the **`super`** keyword is going to call the `__init__` method of the salad class which is Food. This means that Food will pass on all of its attributes to salad.
- Also note `self.mix_food(ingredient)` is calling the `mix_food` method of the `Salad` class which refers back to the `Food` class due to Inheritance.

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 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()
```

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` doesn't 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 `ellington`'s value, we return the representation of ellington given by `__repr__()`. So, we call `BandLeader`'s `__repr__()` method.

Teaching Tips

- For the error, it's important to make sure your students realize that BandLeader overrides Musician's `perform` function, and therefore a function call without the correct number of parameters in the new function will not work.

- Clarify to students the difference between `__str__` and `__repr__`, especially that **`print`** implicitly calls `__str__` and `__repr__`.
  - Another nuance to this which may confuse students is the `__repr__` method in BandLeader, which calls **`str`** on all Musicians in the band. Since the Musician class does not have a defined `__str__` method, it defaults to the defined `__repr__` method.
- The main challenge with this problem is distinctly identifying the class and instance variables and modifying both separately.
  - In particular, every Musician begins with a class variable `popularity`. However, after the first call to `perform`, a new instance variable `self.popularity` is created, which begins with the value `Musician.popularity` + 2.
  - After this first call to a Musician's `perform`, successive calls will increment the respective instance variable by 2.
  - Calling a BandLeader's `perform` function will increment the class variable `Musician.popularity`, which will raise the starting popularities of any new Musicians after their initial performances.
  - Ensure that students understand the interplay between the `popularity` class and instance variable.

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']
    """

    bears = []
    def __init__(self, name):
        self.name = name
        self.organs = []
        Bear.bears.append(self.name)
```

   Note that just doing `bears.append(self.name)` will result in an error! There is no `bears` variable in the `__init__` function frame.

(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]
    """


    organ_counts = {}

    def __init__(self, name, bear):
        self.name = name
        self.bear = bear
        if bear.name in Organ.organ_counts:
            Organ.organ_counts[bear.name] += 1
        else:
            Organ.organ_counts[bear.name] = 1
        bear.organs.append(self)


    def discard(self):
        Organ.organ_counts[self.bear.name] -= 1
        self.bear.organs.remove(self)

    def __repr__(self):
        return self.name
```

Without the `__repr__`, an Organ returns <`__main__.Organ` **`object`**> instead of its name in `Organ.organs`.

Organs do not inherit from Bear, nor should they. Inheritance is used in **is a** relationships, not **has a**.

(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
    """

    def __init__(self, name, bear):
        if hasattr(bear, 'heart'):
            bear.heart.discard()
        bear.heart = self
        Organ.__init__(self, name, bear)
```

Since Hearts are Organs, we can use Organ's `discard` method to remove an old heart easily, without breaking any abstraction barriers. We also can use `Organ.__init__` instead of repeating code.

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.

```
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.

```
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 $\log(\mathtt{n})$.

```
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 $\mathtt{n}^2$.

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

Teaching Tips

- Try visualizing how orders of growth works for each orders of growth (Tables work especially well for quadratic).
    - You can walk through each of the examples above and tally up the amount of times the program runs.
    - For example, in the second f(n), we would jot down that for f(6):
        * Before while: n = 6. Runs: 0.
        * During first iteration: n = 5. Runs: 1.
        * During second iteration: n = 4. Runs: 2.
        * During third iteration: n = 3. Runs: 3.
        * During fourth iteration: n = 2. Runs: 4.
        * During fifth iteration: n = 1. Runs: 5.
        * During sixth iteration: n = 0. Runs: 6.
    - So, from that visualization, we can see that this program seems to run in linear time.

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
   ```

   $\Theta(n^3)$. Inner loop runs n^2 times, and the outer loop runs n times. To get the total, multiply those together.

   (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)
   ```

   $\Theta(2^n)$. Flip will run in constant time so the recursive calls are what end up contributing to the total runtime.

   The runtime can be calculated by the equation $f(n) = f(n-1) + f(n-1) = 2f(n-1)$ and $f(1) = 1$ which together gives us that $f(n) = 2 * 2 * 2 * \cdots * 2 * f(1)$. Rewritten: $f(n) = 2^n$

   (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
   ```

   $\Theta(n2^n)$. There are two loops: the first runs n times for 2^n calls each time (due to pancake), for a total of $n2^n$. The second loop runs n times. When calculating orders of growth however, we focus on the dominating term – in this case, $n2^n$.

2. 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)`

$\Theta(1)$. $f(n)$ is independent of the size of the input n.

(b) `foo(n, fib)`

$\Theta(1)$. See above.