OBJECT ORIENTED PROGRAMMING & LINKED LISTS Meta

COMPUTER SCIENCE MENTORS 61A

October 17–October 21, 2022

Recommended Timeline

- Section 1: Object Oriented Programming
  - OOP Mini Lecture: 8 minutes
  - Q1: WWPD (Star Wars): 10 minutes
  - Q2: Build a Bear: 15 minutes
- Section 2: Linked Lists
  - Linked List Mini Lecture: 5 minutes
  - Q1: WWPD: 8 minutes
  - Q2: Skip served two ways: 12 minutes. You can probably choose to only go over one of these if you run out of time.
  - Q3 (optional): Has Cycle: 12 minutes. A nice challenging problem IMO, but only go over it if your students are ready for it.

Yeah, I know. Both OOP and linked lists. On the same worksheet. Unfortunately, this is just how the scheduling for the course worked out. Sorry about that. As you might imagine, this worksheet might be a little longer than usual, so don't worry if you don't get to all the problems. The worksheets are a question bank around which you can structure your section to best meet the needs of your students. The times do not add up to 50 minutes for this reason.

A little note on the above: When I was a JM, I would often feel bad when I didn't get to all (or most) of the problems on the worksheet. I think I saw the worksheet as a thing to be conquered. Since then, I've learned that this is not a good way to look at the world. I've said this on every single meta, and I'll say it again: your goal is not to get through

Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

every problem on the worksheet. It's to help your students. Go at the appropriate pace for you and your students, and you'll be golden :) Anything you don't get to can be extra practice for them on their own time.

Again, I highly recommend that you not spend too much time in mini-lecture. Only go over what your students need you to go over, because the active learning involved in problem solving is far more instructive.

The midterm is next week! Wish your students good luck. Your students are most likely feeling very stressed about the midterm, so give them some study and last minute prep tips. And if you and they decide that they would like to go over something else (e.g. past exam problems), you are of course welcome to do that.

1 Object Oriented Programming

Object oriented programming is a paradigm that organizes relationships among data into **objects** and **classes**. For example, we can write a `Car` class to represent the concept of cars in general:

```
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()
```

To represent an individual car, we can then create a new instance of `Car` by "calling" the class. Doing so will automatically construct a new object of type `Car`, pass it into the `__init__` method (also called the **constructor**), and then return it. Often, the `__init__` method will initialize the **instance attributes** of an object, which represent the state of an individual object. In this case, the `__init__` method initially sets the `gas` instance attribute of each car to 100.

Classes can also have **class attributes**, which are variables shared by all instances of a class. In the above example, `wheels` is shared by all instances of the `Car` class.

Instance methods are special functions that act on the instances of a class. We've already seen the `__init__` method. We can call instance methods by using the dot notation we use for instance attributes:

```
>>> my_car.drive()
```

```
Current gas level: 90
```

In instance methods, `self` is the instance from which the method was called. We don't have to explicitly pass in `self` because, when we call an instance method from an instance, the instance is automatically passed into the first parameter of the method by Python. That is, `my_car.drive()` is exactly equivalent to the following:

```
>>> Car.drive(my_car)
Current gas level: 80
```

Something I like to emphasize with my students is that you can *only* access class and instance attributes using dot notation from an instance. That is, you can never just write `__init__` or `wheels`; you *must* use dot notation to access these attributes. The reason that students are confused by this is that the rules of variable scope in classes are different from those in functions. They often feel like because they are "inside" the class they should be able to access all of these variables without dot notation. I think it's often useful to dispel this notion by emphasizing that the rules are different and that it's essentially the objects and classes that "hold on" to their instance variables. But you should be careful when giving an explanation like this to not confuse your students more.

This overview is not meant to be a first exposure resource for your students, since there are so many ins and outs of OOP. It is likely that you will need to walk through some of the concepts in a more intuitive way than they are presented here.

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.

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.

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.

`__str__` is special method to convert an object to a human-readable string. It may be invoked by directly calling **`str`** on an object. Additionally, whenever we call **`print`**`()` on an object, it will call the `__str__` method of that object and print whatever value the `__str__` call returned.

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

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. 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 ForceWielder():
    force = 25

    def __init__(self, name):
        self.name = name

    def train(self, other):
        other.force += self.force / 5

    def __str__(self):
        return self.name

class Jedi(ForceWielder):
    lightsaber = "blue"

    def __str__(self):
        return "Jedi " + self.name

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

class Sith(ForceWielder):
    lightsaber = "red"
    num_sith = 0

    def __init__(self, name):
        super().__init__(name)
        Sith.num_sith += 1
        if self.num_sith != 2:
            print("Two there should be. No more, no less.")

    def __str__(self):
        return "Darth " + self.name

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

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

```
("blue", 25)
```

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

```
Jedi("Obi-wan")
```

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

```
AttributeError
```

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

```
(50, 25)
```

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

```
(50, 35.0)
```

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

```
(50, 45.0)
```

```
>>> sidious = Sith("Sidious")
```

```
Two there should be. No more, no less.
```

```
>>> ForceWielder.train(sidious, anakin)
>>> anakin.lightsaber = "red"
>>> anakin.lightsaber, anakin.force
```

```
("red", 50.0)

>>> Jedi.lightsaber

"blue"

>>> print(Sith("Vader"), Sith("Maul").num_sith)

Two there should be. No more, no less.
Darth Vader 3

>>> rey = ForceWielder("Rey")
>>> rey

<__main__.ForceWielder object>

>>> rey.lightsaber

AttributeError
```

In my opinion, going through an example like this is far more helpful for students than a mini-lecture. Try to foresee some questions and confusions might have and how you might address them, for example:

- Why, in the `__init__` method of `Sith` can we use `self.num_sith` instead of `Sith.num_sith`? And why can't we write `self.num_sith += 1`?
- Why does evaluating `rey` give us `<__main__.ForceWielder object>`, but this is not the case when we evaluated `anakin.master`?
- What's going on with `ForceWielder.train(sidious, anakin)`?
- Can we write `Jedi.train(sidious, rey)`, even though neither `rey` nor `sidious` are Jedi?

These are also questions you could bring up if students don't ask them.

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)
```

```python
    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.

2 Linked Lists

Linked lists are a recursive data structure for representing sequences. They consist of a series of "links," each of which has two attributes: `first` and `rest`. The `first` attribute contains the value of the link (which can be an integer, string, list, even another linked list!). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just an empty linked list.

For example, `Link(1, Link(2, Link(3)))` is a linked list representation of the sequence $1, 2, 3$.

Like trees, linked lists naturally lend themselves to recursive problem solving. Consider the following example, in which we double every value in linked list. We mutate the current link and then recursively double the rest.

```
def double_values(link):
     if link is not Link.empty:
        link.first *= 2 # we mutate the value inside of the link
        double_val(link.rest) # we mutate the values in the rest
                              # of the linked list
    # if the link is empty then do nothing
```

Teaching Tips

- Try to draw box and pointer diagrams.
- Make clear that the pointer *points* to a linked list if we have nested linked lists.
- Try to experiment with going over various ways to mutate and create linked lists.
- We have a great visualizer on `https://code.cs61a.org/` where you can call draw(lst) to visualize a list!
- Try using PythonTutor as well!

For each of the following problems, assume linked lists are defined as follows:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
```

```
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

1. What will Python output? Draw box-and-pointer diagrams along the way.

```
>>> a = Link(1, Link(2, Link(3)))
```

```
+---+---+   +---+---+   +---+---+
| 1 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

```
>>> a.first
```

1

```
>>> a.first = 5
```

```
+---+---+   +---+---+   +---+---+
| 5 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

```
>>> a.first
```

5

```
>>> a.rest.first
```

2

```
>>> a.rest.rest.rest.rest.first
```

Error: tuple object has no attribute rest (Link.empty has no rest)

```
>>> a.rest.rest.rest = a
```

```
   +---+---+  +---+---+  +---+---+
+->| 5 | --|->| 2 | --|->| 3 | --|--+
|  +---+---+  +---+---+  +---+---+  |
|                                   |
+-----------------------------------+
```

```
>>> a.rest.rest.rest.rest.first
```

```
2
```

```
>>> repr(Link(1, Link(2, Link(3, Link.empty))))
```

```
"Link(1, Link(2, Link(3)))"
```

```
>>> Link(1, Link(2, Link(3, Link.empty)))
```

```
Link(1, Link(2, Link(3)))
```

```
>>> str(Link(1, Link(2, Link(3))))
```

```
'<1 2 3>'
```

```
>>> print(Link(Link(1), Link(2, Link(3))))
```

```
<<1> 2 3>
```

Teaching Tips

- For assignment statements, Python will not print anything but still have them draw out what the linked list will look like
- Note that we are doing mutation here, so we are actually altering the object that was created in the first assignment.
  - Some students may have minimal exposure to mutating objects so try to emphasize this and make it obvious through diagrams.
- For the error, walk-through how to keep track of which rest corresponds to which

object in the box and pointer diagram. **Make sure they understand why calling rest a fourth time will give us an error (look back at the class definition)**

- Abstraction:
  * our last .rest is set to Link.empty
  * Link.empty is not a Link objects — they do not have a .rest attribute
- Actual implementation:
  * our last .rest is set to Link.empty
  * Link.empty is not a Link objects — they do not have a .rest attribute

• Reassigning the last .rest to point back at the front always trips students up.
  - Make it clear that a is a pointer that points to the linked list. So we are trying to assign the last rest of a to point at what a points to, which is the beginning of the list. **To test their understanding ask what would be different if we instead had**:
    * a.rest.rest.rest = a.rest
  - a way to explain the assignment for this problem is to emphasize the "evaluation" of the RHS and the LHS
  - what is the value of a (a pointer). Really emphasize the implications of pointers here.
  - where are we putting a into? (the box that represents a.rest.rest.rest)
  - same for a.rest.rest.rest = a.rest. what is the value of a.rest? (still a pointer!)
  - Mention that this creates a cycle in the list

2. Write a function `skip`, which takes in a `Link` and skips every other element in the linked list.

(a) First, implement `skip` non-mutatively. That is, return a new linked list with every other element skipped, and do not modify the original linked list.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Unchanged
    """
    if ________________________________________:

        ________________________________________

    elif ______________________________________:

        ________________________________________

    ____________________________________________
```

```
    if lst is Link.empty:
        return Link.empty
     elif lst.rest is Link.empty:
        return Link(lst.first)
    return Link(lst.first, skip(lst.rest.rest))
```

Base cases:

- When the linked list is empty, we want to return a new Link.empty.
- If there is only one element in the linked list (aka the next element is empty), we want to return a new linked list with that single element.

Recursive case:
All other longer linked lists can be reduced down to either a single element or empty linked list depending on whether it has odd or even length. Therefore, we want to keep the first element, and recurse on the element after the next (skipping the immediate next element with `lst.rest.rest`). To build a new linked list, we can add new links to the end of the linked list by calling skip recursively inside the `rest` argument of the `Link` constructor.

Teaching Tips

- Walk through what we want to do by looking at an example box-and-pointer diagram first.
- Make sure they understand, in English, what we are trying to do.
- If students are struggling, have them think about what we can change (pointers), since we can't make new Link objects
  - Specifically, compare the pointers in the original list to the ones in the output list.
  - Think about how you could modify the original pointers.

(b) Now, implement `skip` mutatively. That is, mutate the original list so that every other element is skipped. Do not call the `Link` constructor, and do not return anything.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> skip(a)
    >>> a
    Link(1, Link(3))
    """
```

```
def skip(lst): # Recursively
    if lst is Link.empty or lst.rest is Link.empty:
        return
    lst.rest = lst.rest.rest
    skip(lst.rest)

def skip(lst): # Iteratively
    while lst is not Link.empty and lst.rest is not
       Link.empty:
        lst.rest = lst.rest.rest
        lst = lst.rest
```

Because this problem is mutative, we should never be creating a new list - we should never have `Link(x)`, or the creation of a new Link instance, anywhere in our code! Instead, we'll be reassigning `lst.rest`.

In order to skip a node, we can assign `lst.rest = lst.rest.rest`. If we have lst assigned to a link list that looks like the following:

```
1 -> 2 -> 3 -> 4 -> 5
```

Setting `lst.rest = lst.rest.rest` will take the arrow that points form 1 to 2 and change it to point from 1 to 3. We can see this by evaluating `lst.rest.rest`. `lst.rest` is the arrow that comes from 1, and `lst.rest.rest` is the link with 3.

Once we've created the following list:

```
1 -> 3 -> 4 -> 5
```

we just need to call skip on the rest of the list. If we call skip on the list that starts at 3, we'll skip over the link with 4 and set the pointer from 3 to point to the link with 5. This is the behavior that we want! Therefore, our recursive call is `skip(lst.rest)`, since `lst.rest` is now the link that contains 3.

The purpose of having two parts of this problem is to illustrate the difference between mutative and non-mutative solutions for problems. You should make this clear in your presentation of this. **Teaching Tips**

- Make sure they understand when we are mutating and when we are creating a new linked list
- Draw box-and-pointer diagrams!
- Look for "patterns" or repeated work while you work with your box-and-pointer diagram that you can abstract away with your recursive call.
- Sometimes it is easier to write the recursive call before doing the base cases
- I usually write the recursive call and then see what could "break"
  - If we access lst.first at any point, we have to make sure that lst exists
  - If we access lst.rest.rest at any point we have to make sure that lst.rest exists
  - What errors would we get if we didn't ensure these conditions?

3. **(Optional)** Write `has_cycle` which takes in a `Link` and returns `True` if and only if there is a cycle in the `Link`. Note that the cycle may start at any node and be of any length. Try writing a solution that keeps track of all the links we've seen. Then try to write a solution that doesn't store those witnessed links (consider using two pointers!).

```
def has_cycle(s):
    """
    >>> has_cycle(Link.empty)
    False
    >>> a = Link(1, Link(2, Link(3)))
    >>> has_cycle(a)
    False
    >>> a.rest.rest.rest = a
    >>> has_cycle(a)
    True
    """


    seen = []
    while s:
        if s in seen:
            return True
        seen.append(s)
        s = s.rest
    return False

    # Challenge solution

    if s is Link.empty:
        return False
    slow, fast = s, s.rest
    while fast is not Link.empty:
        if fast.rest is Link.empty:
            return False
        elif fast is slow or fast.rest is slow:
            return True
        slow, fast = slow.rest, fast.rest.rest
    return False
```

Teaching Tips

- Go through multiple examples of Linked List with cycles alongside examples of Linked Lists without cycles.
- Ask your students what patterns they see for lists that have cycles
- It might take some time for students to come up with the fast and slow pointers solution. A common analogy used is the hare and tortoise analogy for this problem.
- If the slow pointer catches up to the fast pointer, we know a cycle must have occured because the slow pointer should never pass the fast pointer in a non-cycle list.