Object Examples and Linked Lists Intro
Warmup: Objects and Names

class MyClass:
    def __init__(self, x):
        self.x = x

# Plain function, not method
def foo(o, x):
    o.x = x

def bar(o, x):
    o = MyClass(x)

x = MyClass(2)
y = MyClass(3)
foo(x, 4)
bar(y, 5)
print(x.x)
print(y.x)

Question:
What does this code print?
Example: Rectangle

- Build a class with the following properties / fields:
  - Width
  - Height

- And the following methods:
  - `__init__(self, width, height)`
  - `calculateArea(self)`
  - `setHeight(self, height)`
  - `setWidth(self, width)`
  - `getWidth(self)`
  - `getHeight(self)`
Example: Using Rectangle

- Construct 2 rectangles:
  - 8 x 12
  - 4 x 4

- Calculate their areas
Example: Caching Area

- Might not want to recompute area every time
- Add another field (in __init__) called cachedArea
- When calculateArea() called return cachedArea
Example: Circle Object

- Create a “circle” object
  - Needs a “center”
  - Can either have a radius or a diameter (you pick)
- Must support “calculateArea” method
Example: ShapeList

- Create a class ShapeList:
  - One field: underlying array (call this list)
  - __init__(self):
    - Initialize list (to empty list)
  - length(self): calculates the length of the list
  - add(self, shape):
    - Adds a shape to the underlying list
  - sumOfAreas(self):
    - Sum of the areas of all of the shapes
Testing ShapeList

- Create empty ShapeList
- Add a 8 x 12 rectangle
- Add an 4 x 5 CachedRectangle
- Add a circle centered at (1,3) whose radius is 2
- Call sumOfAreas
Example: Array insertion is $O(n)$

\[
    l = \begin{array}{cccccc}
        1 & 2 & 17 & 18 & 19 & -4 \\
    \end{array}
\]

When we do $[i] + l$, we get...
Example: Array insertion is $O(n)$

\[ l = \begin{array}{cccccc}
1 & 2 & 17 & 18 & 19 & -4
\end{array} \]

When we do \[ [i] + l \], we get...

Step (1)
Allocate fresh memory for new array....
Example: Array insertion is $O(n)$

$$l = \begin{array}{cccccc}
1 & 2 & 17 & 18 & 19 & -4
\end{array}$$

When we do $[i] + l$, we get...

Step (2)
Copy array over element-wise

$$\begin{array}{cccccc}
\text{---} & \text{---} & \text{---} & \text{---} & \text{---} & -4
\end{array}$$
Example: Array insertion is $O(n)$

\[ l = \begin{array}{cccccc}
1 & 2 & 17 & 18 & 19 & -4 \\
\end{array} \]

When we do $[i] + l$, we get...

Step (2)
Copy array over element-wise

\[ i \begin{array}{cccccc}
1 & 2 & 17 & 18 & 19 & -4 \\
\end{array} \]
Example: Array insertion is $O(n)$

When we do $[i] + l$, we get...

Takes $O(n)$ time!
Upshot: Performing n insertions takes $O(n^2)$ time

(But: Random-Access is $O(1)$ time!)
We can get $O(1)$ insertion time if we change the structure of the list!
Observation

We can get $O(1)$ insertion time if we change the structure of the list!
Observation

Represent list as a collection of “links”

Each link consists of data + link to next link (or None)
Represent list as a collection of “links”

Each link consists of data + link to next link (or None)
Inserting data to front of list is just creating a new link
We can still “walk over” the list in linear time
We can still “walk over” the list in linear time.
We can still “walk over” the list in linear time
We can still “walk over” the list in linear time
We can still “walk over” the list in linear time
We can still “walk over” the list in linear time
Creating a Link class

We need...
• The data itself
• A reference to “next” node
class Link:
    def __init__(self, data, next):
        self.data = data
        self.next = next

    def getData(self):
        return self.data
    def getNext(self):
        return self.next
class Link:
    def __init__(self, data, next):
        self.data = data
        self.next = next

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

Example Usage:

l1 = Link(12, None)
class Link:
    def __init__(self, data, next):
        self.data = data
        self.next = next

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

Example Usage:

l1 = Link(12, None)
l2 = Link(15, l1)
Challenge: write a **function** (not a method) to check if the list contains a negative num

class Link:
    def __init__(self, data, next):
        self.data = data
        self.next = next

    def getData(self): return self.data
    def getNext(self): return self.next

    def containsNeg(link):
        ... # Your code here
Building a **List**

- Can **encapsulate** first link in a class called **List**

- Then, we can support the following operations:
  
  - **Add**
  
  ```python
class List:
    def __init__(self):
        self.first = None
    def add(self, data):
        self.first = Link(data, self.first)
```}

- **Contains**

- **Remove**
We can see that `add` is $O(1)$ since it simply:
- Creates a new Link
- The constructor for Link runs in constant time
- Sets the first element of the list to that new link

```python
class List:
    def __init__(self):
        self.first = None

    def add(self, data):
        self.first = Link(data, self.first)
```

```python
class List:
    def __init__(self):
        self.first = None

    def add(self, data):
        self.first = Link(data, self.first)
```
Exercise: Implement `contains`

class List:
    def __init__(self):
        self.first = None

    def add(self, data):
        self.first = Link(data, self.first)

Hint: think about using loop to follow links from next...
Exercise: Implement `getIth`

```python
class List:
    def __init__(self):
        self.first = None

    def add(self, data):
        self.first = Link(data, self.first)
```

Question: what is runtime of `getIth`?
Exercise: Implement `remove`

```python
class List:
    def __init__(self):
        self.first = None

    def add(self, data):
        self.first = Link(data, self.first)
```

Brainstorm

• What’s an example application where you would perform frequent insertions

• What’s an example where data is relatively fixed?
Observations

- All data structures are about trade-offs

- Linked lists trade random-access for $O(1)$ insertion

- Can still “go through” (iterate over) lists in linear time

- But random-access is $O(n)$
  - Good for applications that don’t require random-access
  - Many don’t!