A. What the id() function does

The id() function returns a large integer, which is often the machine address of the thing in memory.

\[ \text{id(object)} \]

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

**CPython implementation detail:** This is the address of the object in memory.

```python
>>> x = 5
>>> id(x)
505911024
>>> y=x
>>> id(y)
505911024
>>> y=5
>>> id(y)
505911024

>>> x = []
>>> id(x)
36179920
>>> y = []
>>> id(y)
24173184

>>> x = ("cat", 59)
>>> id(x)
36103872
>>> y = ("cat", 59)
>>> id(y)
36098176
>>> z = x
>>> id(z)
36103872
```

For simple values, there is only one copy in memory. Hence the same id number.

For complex objects, like lists, sets, tuples, dicts, you get a new, unique chunk of memory each time, hence different id numbers.
>>> x = [5, 2, 6, 4]
>>> id(x)
37108912
>>> x.append(9)
>>> id(x)
37108912
>>> del x
>>> id(x)
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    id(x)
NameError: name 'x' is not defined

>>> x = [5, 2, 6, 4]
>>> id(x)
36027408
>>> y = x
>>> id(y)
36027408
>>> del x
>>> id(y)
36027408
>>> y
[5, 2, 6, 4]

Notice that merely putting a new thing inside a container object doesn’t change the container’s id, since you are still pointing at the container with the program variable x.

The reason why this doesn’t break is because Python knows that the list is pointed to by more than one variable so it won’t actually delete it. However, some ancient langs like Pascal achieved the same effect by effectively ignoring the del operator; it didn’t do anything, didn’t delete.
class LinkedList:
    class Node:
        def __init__(self, value):
            self.value = value
            self.next = None
        def __init__(self):
            self.head = None  # empty linked list
        def append(self, something):
            pass
        def prepend(self, something):
            newnode = self.Node(something)
            if self.head == None:
                self.head = newnode
            else:
                newnode.next = self.head
                self.head = newnode
    def print(self):
        runner = self.head
        while runner != None:
            print(runner.value, end="", )
        print()
    def find(self, target):
        currentNode = self.head
        while currentNode != None:
            if currentNode.value == target:
                return currentNode
            break
            currentNode = currentNode.next
        else:
            return None
    def __str__(self):
        res = ""
        runner = self.head
        while runner != None:
            res += str(id(runner)) + ": " + str(runner.value) + ", "
            runner = runner.next
        return res
    # main code
    mylist = LinkedList()
    mylist.prepend("stuff")
    mylist.prepend("apples")
    mylist.prepend("oranges")
    mylist.prepend("bananas")
    print(mylist)
    mylist.print()  # 2 different ways
Here's the code to start with.

class LinkedList:

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

    def __init__(self):
        self.head = None  # empty linked list

    def append(self, something):
        pass

    def prepend(self, something):
        newnode = self.Node(something)
        if self.head == None:
            self.head = newnode
        else:
            newnode.next = self.head
            self.head = newnode

    def print(self):
        runner = self.head
        while runner != None:
            print(runner.value, end=" ",
        print()

    def find(self, target):
        currentNode = self.head
        while currentNode != None:
            if currentNode.value == target:
                return currentNode
            break
            currentNode = currentNode.next
        return None

    def __str__(self):
        rets = ""
        runner = self.head
        while runner != None:
            rets += str(id(runner)) + " : " + str(runner.value) + ", "
            runner = runner.next
        return rets

# main code

mylist = LinkedList()
mylist.prepend("stuff")
mylist.prepend("apples")
mylist.prepend("oranges")
mylist.prepend("bananas")
print(mylist)

mylist.print()  # 2 different ways

print(mylist.find("apples"))
ACE
Day 18

Linked Lists

1. Given the linked list code on the separate page, add line so that we see the machine address of each new node as it is prepended. Mark up the existing code.

2. Write a length method that runs through and counts how many nodes are in the list. Use print() as an example. Put your method below.

   ```python
def length(self):
    counter = 0
    if self.head == None:
        return 0
    else:
        while self.head != None:
            counter += 1
            self.head = self.head.next
        return counter
    ```

3. Given that find() exists and works, write a dunder method that would implement the "in" operator. Do not write the actual comparison code! Use the existing find(). For example, we would be able to code:

   ```python
   if "cat" in mylinkedlist:
       print("Yay!")
   ```

   ```python
def __contains__(self, item):
    if find(item) == None:
        return False
    else:
        return True
    return find(item) != None
   ```
def length(self):
    return _length_helper(self, head)

def _length_helper(self, runner):
    if runner is None:
        return 0
    else:
        return 1 + _length_helper(runner, runner.next)
4. Write the append method which starts at the head, travels to the end node, and attaches a new node containing "something" as its value. Be careful! You must stop BEFORE runner becomes None or else you can't attach anything. You can never do:

```python
runner.next = mynewnode
```

when runner's value is None, because Python will crash. Below is a picture:

def append(self, something):
    newptr = self.Node(something)
    runner = self.head
    while runner.next != None:
        runner = runner.next
    runner.next = newptr
    if self.head == None:
        self.head = newptr
    else:
prev = None
runner = self.head
while runner != None:
    prev = runner
    runner = runner.next
    prev.next = newptr
5. Write a method called `delete(somevalue)` which unlinks the node that has `somevalue` as its value. Assume that we definitely know `somevalue` is in the linked list somewhere. Start out by writing the shell of the method and handling the case where `somevalue` is the first node’s value. If not, then call another method (which we haven’t yet written) named `delete_deep(somevalue)`.

![garbage collected]

6. Now write the method `delete_deep(somevalue)` which will use the runner concept. This time we definitely know that `somevalue` is not the first node, so we must not change `self.head`! Run through the list and when `runner.value equals somevalue`, you’ve found the culprit. You need to copy its next field into the next field of runner’s predecessor. Except you’ve lost the predecessor pointer! How can you fix that? Write `delete_deep(somevalue)`.