<table>
<thead>
<tr>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FORTRAN 1957</strong></td>
<td>can expand, contract</td>
</tr>
<tr>
<td>fixed-size list</td>
<td>usv. heterogeneous</td>
</tr>
<tr>
<td>usu. homogeneous</td>
<td>pets = []</td>
</tr>
<tr>
<td></td>
<td>delete pets[3]</td>
</tr>
<tr>
<td></td>
<td>dynamic</td>
</tr>
<tr>
<td>![Diagram of an array]</td>
<td>Python list</td>
</tr>
<tr>
<td></td>
<td>static</td>
</tr>
</tbody>
</table>
def _print(pointer):
    if pointer is None:
        return
    else:
        print(pointer.value, end="", sep="")
        print(_print(pointer.next))

def print(self):
    return 
    _print(self)

LinkedList._print(self, head)

output: cat, dog, bird,
Here's the original linked list class:

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="", )
            runner = runner.next
        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):
        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

\[ z = \text{str} (\text{mylist}) \]
```python
def append(self, something):
    newnode = self.Node(something)
    if self.head is None:
        self.head = newnode
    else:
        runner = self.head
        while runner.next != None:
            runner = runner.next
        runner.next = newnode
        set runner to pt to list node
        while runner.next is pointing to an actual node
            advance runner to point to that node
        link in the new node by hooking its ptr
        into runner

None cannot have a "dot" field reference
```
attach "orca"
make new node

now attach to the end

We know we're at end of chain when next field has None in it

When done it will look like
B. Playing with Linked lists

There are many challenges:

1. How do you add a new node to a linked list?
   a.) where, at the end?
   b.) where, at the front?
   c.) where, in the middle somewhere?
   d.) where in order to keep the list sorted?

2. How do you delete a node from a linked list?
   a.) where, at the end?
   b.) where, at the front?
   c.) where, in the middle somewhere?

3. How can you navigate around a linked list?
   a.) can you go backwards?
   b.) can you quickly find the beginning or the end of the list or both?
   c.) is the list circular?

4. How do you copy a linked list?
   a.) shallow copy
   b.) deep copy

5. What about deleting nodes?
   a.) as long as there is one pointer to a chunk of memory in some program variable,
      then the chunk is “alive” and won’t be garbage collected (gc’ed)
   b.) if 2 program variables point to the same chunk, and you delete the chunk, then
      what do they point to? This is the dangling pointer problem.

Patterns:

#1. Run through the entire list with a runner

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

#2. Run through the entire list with a runner and prev (previous) pointer

```python
def xxx(self):
    runner = self.head
    prev = None
    while runner != None:
        # do something with prev and runner
        prev = runner
        runner = runner.next
```
#2a. Run through the entire list with a runner and prev (previous) pointer

```python
def is_sorted(self):
    prev = self.head
    runner = self.head.next  # assume self.head is not None
    
    while runner != None:
        if prev.value < runner.value:
            return False  # list is not in sorted order
        prev = runner
        runner = runner.next
    return True
```

Let's investigate how we could insert_sorted in a doubly linked list.

#1. What does Node have to look like? Two pointers, forward and backward

#2. Where are the None's?? First and last nodes

#3. Prepend to front of list:

(Interpret the code from zybooks...)

```python
ListAppend(list, newNode) {
    if (list->head == 0) { // List empty
        list->head = newNode
        list->tail = newNode
    }
    else {
        list->tail->next = newNode
        newNode->prev = list->tail
        list->tail = newNode
    }
}
```

list->tail->next = list.tail.next

#4. Insert_sorted(), run through list, do not need a prev point like pattern #2a above! Look for where target >= runner.value, and that's where you link it in. Two special cases: what are they?