CS134:
Lists & Mutability
Announcements & Logistics

• **HW 4** due tonight at 10pm

• **Lab 4** today/tomorrow
  • **Part 1** due Wed/Thur at 10pm
    • We will run some tests and return automated feedback
  • **Part 2** is due next week (but there is no lab next week!)
    • We’ll provide info on students help and TA hours during reading days on Wed (since Friday could be Mountain Day)

---

Do You Have Any Questions?
Last Time

• Learned about **list comprehensions** and accessing **lists of lists**

• Used our knowledge about lists and loops to analyze "interesting" properties of our student data
  
  • Focused on maintaining the state of variables when looping, and how to update state based on conditionals

• Example functions: `characterList`, `yearList`
Today’s Plan

• Learn how to find max/min values in a list (when we can’t use the `min()` and `max()` functions)

• Review old and new list methods that modify the list:
  • `.append()`, `.extend()`, `.insert()`, `.remove()`, `.pop()`, `.sort()`

• Discuss implications of **mutability** in Python in more detail
Exercise: Student Fun Facts!

• Write a function `mostVowels` that can be used to compute the list of students with the most vowels in their first name. (Hint: use `countVowels()` which returns the number of vowels in a string.)

```python
def mostVowels(wordList):
    """Takes a list of strings wordList and returns a list of strings from wordList that contain the most # vowels""
```

• General strategy for finding max in list of lists?
  • Initialize a max value BEFORE the loop to a very small number
  • If you see a value bigger than max while looping, update max
Exercise: Student Fun Facts!

- Write a function `mostVowels` that can be used to compute the list of students with the most vowels in their first name. (Hint: use `countVowels()` which returns the number of vowels in a string.)

```python
def mostVowels(wordList):
    '''Takes a list of strings wordList and returns a list of strings from wordList that contain the most # vowels'''

    maxSoFar = 0 # initialize counter
    result = []
    for word in wordList:
        count = countVowels(word)
        if count > maxSoFar:
            # update: found a better word
            maxSoFar = count
            result = [word]
        elif count == maxSoFar:
            result.append(word)
    return result

# which student(s) has most vowels in their name?
mostVowelNames = mostVowels(firstNames)
mostVowelNames

['Genevieve', 'Maximilian']
```
Exercise: Student Fun Facts!

Write a function `leastVowels` that can be used to compute the list of students with the least vowels in their first name. (Hint: use `countVowels()` again.)

```python
def leastVowels(wordList):
    ''' Takes a list of strings wordList and returns a list of strings in wordList that contain the least number of vowels''
    minSoFar = len(wordList[0]) # initialize counter
    result = []
    for word in wordList:
        count = countVowels(word)
        if count < minSoFar:
            # update: found a better word
            minSoFar = count
            result = [word]
        elif count == minSoFar:
            result.append(word)
    return result

leastVowels(firstNames)

['RJ', 'C.J.', 'M']
```
List Mutability

A quick review of old and new methods that modify a list:

- `append()`, `extend()`,
- `pop()`, `insert()`, `remove()`, `sort()`
Direct Modification: Element Assignment

`myList[index] = item`: though not a method, an assignment to a specific index can modify a list directly (this won’t work using strings!)

**Example.**

`myList[1] = 7`  # assign 7 to index 1 of `myList`

<table>
<thead>
<tr>
<th>myList Before</th>
<th>myList After</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4]</td>
<td>[1, 7, 3, 4]</td>
</tr>
</tbody>
</table>
append()

myList.append(item) : appends item to end of list

Example.

myList.append(5)  # insert 5 at the end of the list

myList Before
[1, 7, 3, 4]

myList After
[1, 7, 3, 4, 5]
**extend()**

`myList.extend([itemList])` : appends all the items in `itemList` to the end of `myList`.

**Example.**

```python
myList.extend([6, 8])  # insert both 6 and 8 at the end of the list
```

<table>
<thead>
<tr>
<th>myList Before</th>
<th>myList After</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 7, 3, 4, 5]</td>
<td>[1, 7, 3, 4, 5, 6, 8]</td>
</tr>
</tbody>
</table>
**pop()**

`myList.pop(index)`: Removes the item at *a given index* (int) and returns it. If no index is given, by default, `pop()` removes and returns the *last item* from the list.

**Example.**

```python
val = myList.pop(3)  # returns val = 4
```

**myList Before**

```
[1, 7, 3, 4, 5, 6, 8]
```

**myList After**

```
[1, 7, 3, 5, 6, 8]
```
pop() 

`myList.pop(index)`: Removes the item at a given index (int) and returns it. If no index is given, by default, `pop()` removes and returns the last item from the list.

Example.

```python
val = myList.pop()  
val = 8
```

**myList Before**

```
[1, 7, 3, 5, 6, 8]
```

**myList After**

```
[1, 7, 3, 5, 6]
```
**insert()**

`myList.insert(index, item)` : inserts item at index (int) in `myList`, all items to the right of index shift over to make room

**Example.**

`myList.insert(0,11)`  # insert 11 at index 0

<table>
<thead>
<tr>
<th>myList Before</th>
<th>myList After</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 7, 3, 5, 6]</td>
<td>[11, 1, 7, 3, 5, 6]</td>
</tr>
</tbody>
</table>
myList.insert(index, item) : inserts item at index (int) in myList, all items to the right of index shift over to make room

Example.

myList.insert(10,12)  # insert 12 at index 10

myList Before
[11, 1, 7, 3, 5, 6]

myList After
[11, 1, 7, 3, 5, 6, 12]
remove()

**myList.remove(item)**: removes first occurrence of **item** from **myList**, all items to the right of removed item shift to the left by one (Unlike pop(), item is not returned!)

**Example.**

```python
myList.remove(12)  # remove 12 from myList
```

<table>
<thead>
<tr>
<th>myList Before</th>
<th>myList After</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11, 1, 7, 3, 5, 6, 12]</td>
<td>[11, 1, 7, 3, 5, 6]</td>
</tr>
</tbody>
</table>

**DO NOT USE remove() IN LAB 4!!!!!**
DO NOT USE .remove() IN LAB 04!
**sort()**

`myList.sort()` : sorts the list *in place* in ascending order

**Example.**

`myList.sort()`  # sort by mutating myList

**myList Before**  

[11, 1, 7, 3, 5, 6]

**myList After**  

[1, 3, 5, 6, 7, 11]
Identity and Value
Value vs Identity

- Python is an **object oriented language**: everything is an object!
- An **object's identity** never changes once it has been created; think of it as the object's **address** in memory.
  - The `id()` function returns an integer representing an object's identity (or address).
- An **object's value** is the value assigned to the object when it is created.

```python
>>> num = 5
>>> id(num)
4486937008
```
Value vs Identity

• An **object’s identity** never changes once it has been created; think of it as the object’s **address** in memory

• On the other hand, an **object’s value** can change
  
  • Objects whose values can change are called **mutable**; objects whose values cannot change are called **immutable**

```python
>>> num = 5
>>> id(num)
4486937008
```

Variable names like `num` point to memory addresses of stored value
Comparing Value vs Identity

- The `==` operator compares the value of an object (i.e., are the contents of the objects the same?)
- The `is` operator compares the identity of two objects (i.e., do they have the same memory address?)
  - `var1 is var2` is equivalent to `id(var1) == id(var2)`

```python
>>> num = 5
>>> id(num)
4486937008
```

Variable names like `num` point to memory addresses of stored value
Mutability in Python

**Strings, Ints, Floats are Immutable**

- Once you create them, their value **cannot** be changed!
- All functions and methods that manipulate these objects return a new **object** and **do not modify** the original object

**Lists are Mutable**

- List values **can** be changed
- We just reviewed how we can mutate/change what’s in a list using methods; these methods **modify** original list
- If we use sequence operators on lists, these functions and operations return a new **list** and **do not modify** the original list
Ints, Floats are Immutable

```python
>>> num = 5
>>> id(num)
4486937008

>>> num = num + 1
>>> id(num)
```

Has the identity of `num` changed?

Attempts to change an immutable object create a new object.
Ints, Floats are Immutable

>>> num = 5
>>> id(num)
4486937008

>>> num = num + 1
>>> id(num)
4486937040

Identity of ints cannot be changed, num assumes a new identity

Attempts to change an immutable object create a new object
Strings are Immutable

>>> word = "Williams"
>>> college = word
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True

Attempts to change an immutable object create a new object

Even though word and college have the same identity and value, if we update one of them, it just assumes a new identity!

Variable names point to memory addresses of stored value
Strings are Immutable

```python
>>> word = "Williams"
>>> college = word
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True

>>> word = "Amherst"
>>> print(id(word), id(college))
4518871920 4518582576

>>> word is college
False
```

Attempts to change an immutable object create a new object
Strings are Immutable

>>> word = "Williams"
>>> college = "Williams"
>>> word == college
True

>>> print(id(word), id(college))
4518582576 4518582576

>>> word is college
True

Even though we created word and college separately, they still point to the same memory address. This is a (confusing) optimization in Python.

Immutable objects that are == also share an identity
String Methods/Operations Return New Strings

• String methods like `.lower()`, `.upper()` return a new string
• Sequence operations, like slicing `[:]`, return new sequences

```python
>>> name = "sally"
>>> id(name)
4574657776
```
String Methods/Operations Return New Strings

- String methods like `.lower()`, `.upper()` return a new string
- Sequence operations, like slicing `[:]`, return new sequences

```python
>>> name = "sally"
>>> id(name)
4574657776
>>> name = name[1:4]
>>> id(name)
4574684720
```
Sequence Operations Return New Sequences

• The following operations, that can be performed on both lists and strings, and always return a new list/string
  • [: :) ] slicing operator: returns a new sliced sequence
  • assignment of a new sequence to a variable
    • names = 'Iris and Jeannie'
    • myList = [1, 2, 3]
  • concatenation (+) always creates a new sequence
Lists are Mutable

>>> myList = [1, 2, 3]
>>> id(myList)
4418551104

>>> myList.append(4)
>>> id(myList)
4418551104

Note: Value changes, identity stays the same

More on this next time!

Value of list objects can change, keeping identity the same
The end!
Lab 4
Lab 4 Goals

• In Lab 4 you will implement several voting algorithms and helpful functions for manipulating election data

• Lab 4 will give you experience with:
  • Lists of strings
  • Lists of lists of strings
  • Loops
  • Using string and list methods
  • File reading

• Pay close attention to expected input (lists of strings, list of lists of strings, etc.) and expected output
Ballot Data

- Ballot data is represented in various text files
- Each line represents a single voter's ranked choices

```python
# different types of coffee
filename = "csv/coffee.csv"
with open(filename) as coffeeTypes:
    allCoffee = []
    for coffee in coffeeTypes:
        allCoffee.append(coffee.strip().split(<?,
print(allCoffee)
```

```
[['kona', 'dickason', 'ambrosia', 'wonderbar', 'house'],
['kona', 'house', 'ambrosia', 'wonderbar', 'dickason'],
['kona', 'ambrosia', 'dickason', 'wonderbar', 'house'],
['kona', 'ambrosia', 'wonderbar', 'dickason', 'house'],
['house', 'kona', 'dickason', 'wonderbar', 'ambrosia'],
['kona', 'house', 'dickason', 'ambrosia', 'wonderbar'],
['kona', 'house', 'dickason', 'ambrosia', 'wonderbar'],
['dickason', 'ambrosia', 'wonderbar', 'kona', 'house'],
['house', 'kona', 'ambrosia', 'dickason', 'wonderbar'],
['ambrosia', 'house', 'wonderbar', 'kona', 'dickason'],
['wonderbar', 'ambrosia', 'kona', 'house', 'dickason'],
['house', 'wonderbar', 'kona', 'ambrosia', 'dickason']
```
You’ll use string and list methods to process the data and implement several different voting algorithms.
Remember
mostVowels(..) and
leastVowels(..)
from lecture!
The end!