CMSC201
Computer Science I for Majors

Lecture 08 – Lists

Last Class We Covered

- Using **`while`** loops
  - Syntax of a **`while`** loop
  - Interactive loops
  - Infinite loops and other problems

- Nested Loops

Any Questions from Last Time?

Today's Objectives

- To learn about lists and what they are used for
  - To be able to create and update lists
  - To learn two different ways to mutate a list
    - **`append()`** and **`remove()`**
- To understand how two-dimensional lists work

- To get more practice with **`while`** loops
  - Sentinel values

Introduction to Lists

Exercise: Average Three Numbers

- Read in three numbers and average them

```
num1 = int(input("Please enter a number: "))
num2 = int(input("Please enter a number: "))
num3 = int(input("Please enter a number: "))
print((num1 + num2 + num3) / 3)
```

- Easy! But what if we want to do 100 numbers? Or 1000 numbers?
- Do we want to make 1000 variables?

Using Lists

- We need an easy way to hold individual data items without needing to make lots of variables
  - Making **`num1`**, **`num2`**, **`...`**, **`num99`**, **`num100`** is time-consuming and impractical

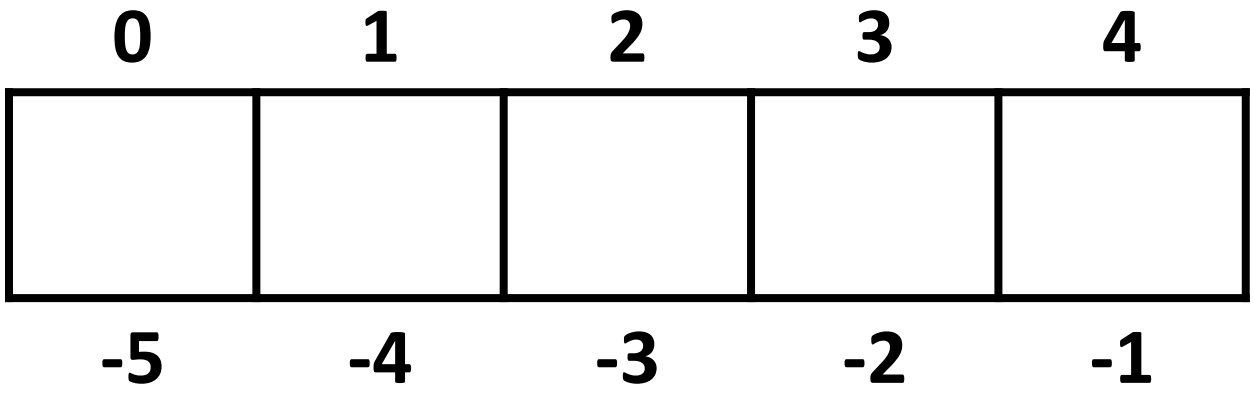
- Instead, we can use a ***list*** to hold our data
  - A list is a ***data structure***: something that holds multiple pieces of data in one structure

Using Lists: Individual Variables

- We need an easy way to refer to each individual variable in our list
- What are some possibilities?
  - Math uses subscripts (x_1, x_2, x_3, etc.)
  - Instructions use numbers ("Step 1: Combine…")
- Programming languages use a different syntax
  - `x[1]`, `x[0]`, `instructions[1]`, `point[i]`

Numbering in Lists

- Lists are numbered the same as strings
  - Index from the left: start at 0
  - Index from the right: start at -1



List Syntax

- Use `[]` to assign initial values (*initialization*)

```
myList = [1, 3, 5]
words  = ["Hello", "to", "you"]
```

- And to refer to individual elements of a list

```
>>> print(words[0])
Hello
>>> myList[0] = 2
```

Properties of a List

- Heterogeneous (multiple data types!)
- Contiguous (all together in memory)
- Ordered (numbered from 0 to n-1)

- Have instant ("random") access to any element
- Elements are added using the **`append`** method
- Are "mutable sequences of arbitrary objects"

List Example: Grocery List

- You are getting ready to head to the grocery store to get some much needed food

- In order to organize your trip and to reduce the number of impulse buys, you decide to make a grocery list

Image from flickr.com

List Example: Grocery List

- Inputs:
  - 3 items for grocery list
- Process:
  - Store groceries using list data structure
- Output:
  - Final grocery list

Grocery List Code

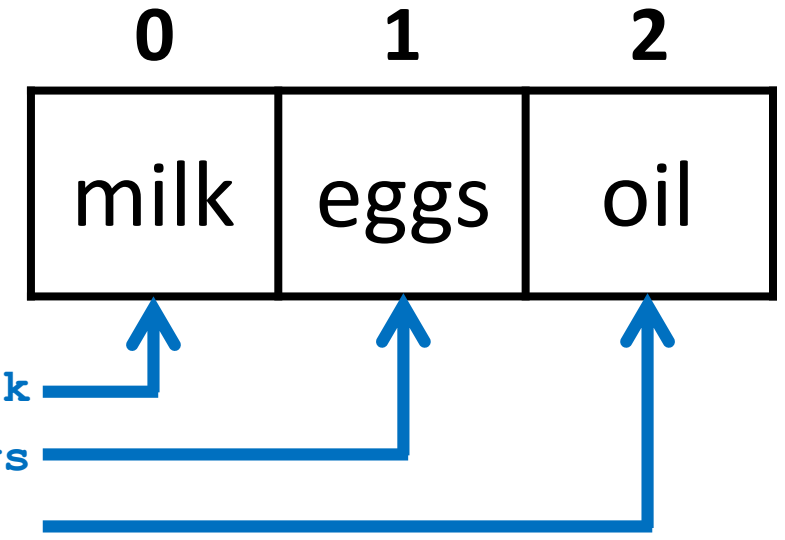
```
def main():
    print("Welcome to the Grocery Manager 1.0")
    grocery_list = [None]*3     # initialize list value and size

    # get grocery items from the user
    grocery_list[0] = input("Please enter your first item:  ")
    grocery_list[1] = input("Please enter your second item: ")
    grocery_list[2] = input("Please enter your third item:  ")

    # print out the items they selected
    print(grocery_list[0])
    print(grocery_list[1])
    print(grocery_list[2])
main()
```

Grocery List Demonstration

- Here's a demonstration of what the code is doing




```
bash-4.1$ python groceries.py
Please enter your first item:  milk
Please enter your second item: eggs
Please enter your third item:  oil
```

```
grocery_list = [None]*3

grocery_list[0] = input("Please enter ...: ")
grocery_list[1] = input("Please enter ...: ")
grocery_list[2] = input("Please enter ...: ")
```

List Example: Grocery List

- What would make this process easier?
- Loops!
  - Instead of asking for each item individually, we could keep adding items to the list until we wanted to stop (or the list was “full”)
- We’ll update our program to use a loop soon
  - For now, let’s talk about **`while`** loops a bit more

Mutating Lists

Mutating Lists

- Remember that lists are defined as "mutable sequences of arbitrary objects"
  - "Mutable" just means we can change them

- So far, the only thing we've changed has been the content of the list
  - But we can also change a list's size, by adding and removing elements

Two List Functions

- There are two functions we'll cover today that can add and remove things to our lists
  - There are more, but we'll cover them later

```
.append()
.remove()
```

List Function: `append()`

- The **`append()`** function lets us add items to the end of a list, increasing its size

  `listName.append(itemToAppend)`

- Useful for creating a list from flexible input
  - Allows the list to expand as the user needs
  - No longer need to initialize lists to **`[None]*num`**
    - Can instead start with an empty list **`[]`**

Example of `append()`

- We can use **`append()`** to create a list of numbers (continuing until the user enters 0)

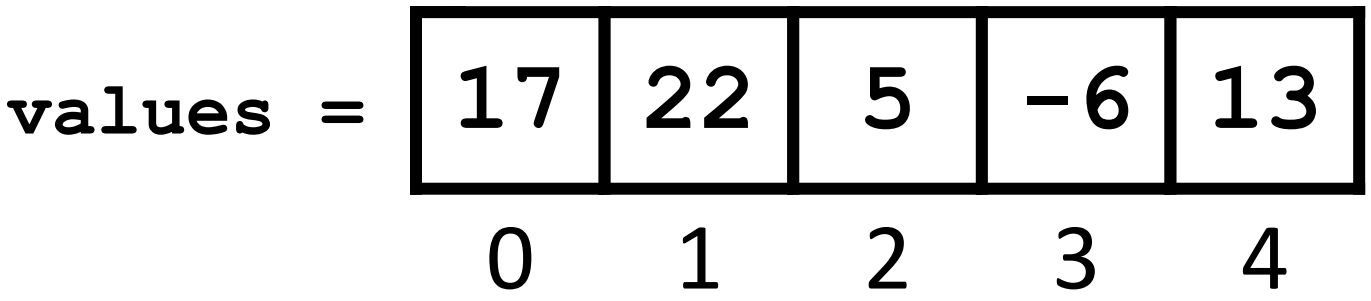
```
values = []    # initialize the list to be empty
userVal = 1    # give loop variable an initial value

while userVal != 0:
    userVal = int(input("Enter a number, 0 to stop: "))
    if userVal != 0:          # only append if it's valid
        values.append(userVal) # add value to the list
```

Example of `append()`

- We can use **`append()`** to create a list of numbers (continuing until the user enters 0)

```
while userVal != 0:
    userVal = int(input("Enter a number, 0 to stop: "))
    if userVal != 0:          # only append if it's valid
        values.append(userVal) # add value to the list
```



List Function: `remove()`

- The `remove()` function lets us remove an item from the list – specifically, it finds and removes the first instance of a given value

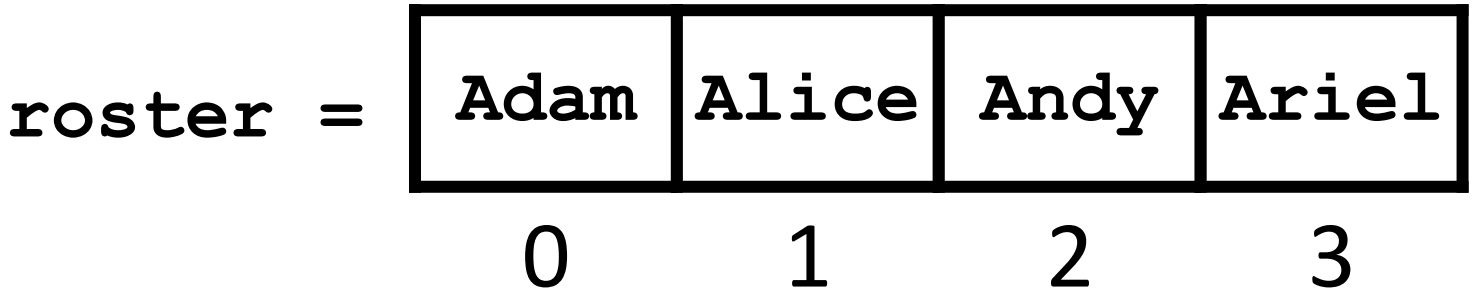
```
listName.remove( valueToRemove )
```

- Useful for deleting things we don't need
  - For example, removing students who have dropped the class from the class roster
  - Instead of the list having "empty" elements

Example of `remove()`

- We can use `remove()` to remove students who have dropped the class from the roster

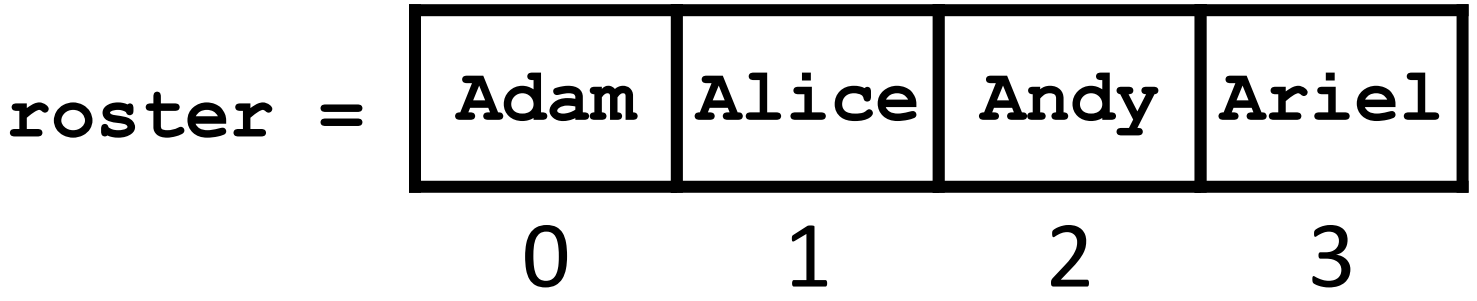
```
roster = ["Adam", "Alice", "Andy", "Ariel"]
```



Example of `remove()`

- We can use `remove()` to remove students who have dropped the class from the roster

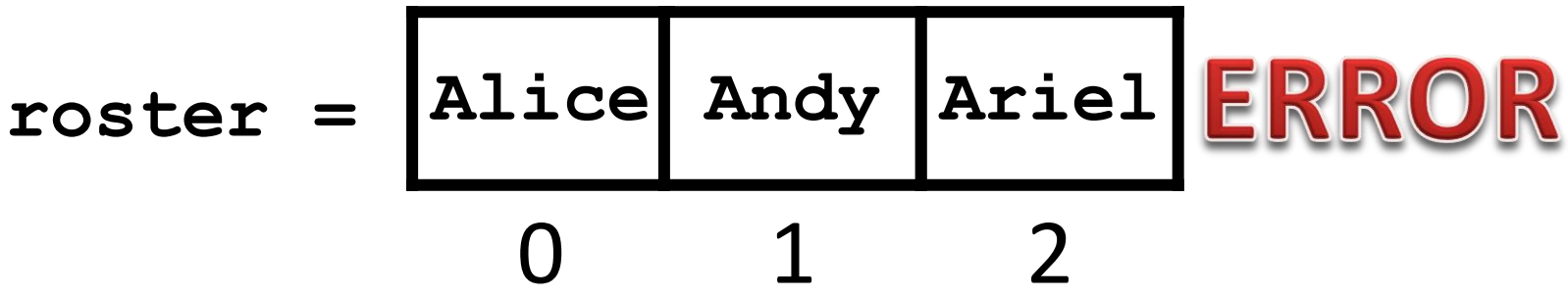
```
roster = ["Adam", "Alice", "Andy", "Ariel"]
roster.remove("Adam")    # Adam has dropped the class
```



Example of `remove()`

- We can use `remove()` to remove students who have dropped the class from the roster

```
roster = ["Adam", "Alice", "Andy", "Ariel"]
roster.remove("Adam")    # Adam has dropped the class
roster.remove("Bob")     # Bob is not in the roster
```



Sentinel Values and `while` Loops

When to Use `while` Loops

- `while` loops are very helpful when you:
  - Want to get input from the user that meets certain specific conditions
    - Positive number
    - A non-empty string
  - Want to keep getting input until some "end"
    - User inputs a value that means they're finished
    - Reached the end of some input (a file, etc.)

what we're covering now

Sentinel Values

- ***Sentinel values*** "guard" the end of your input
- They are used:
  - When you don't know the number of entries
  - In **`while`** loops to control data entry
  - To let the user indicate an "end" to the data
- Common sentinel values include:
  - **`STOP`**, **`-1`**, **`0`**, **`QUIT`**, and **`EXIT`**

Image from publicdomainpictures.net

Sentinel Loop Example

- Here's an example, where we ask the user to enter student names:

```
students = []
name = input("Please enter a student, or 'QUIT' to stop: ")


while name != "QUIT":
    students.append(name)
    name = input("Please enter a student, or 'QUIT' to stop: ")
```

Sentinel Loop Example

- Here's an example, where we ask the user to enter student names:

initialize the loop variable with user input

```
students = []
name = input("Please enter a student, or 'QUIT' to stop: ")
```

check for the termination condition

```
while name != "QUIT":
    students.append(name)
    name = input("Please enter a student, or 'QUIT' to stop: ")
```

get a new value for the loop variable

Sentinel Loop Example

- Here's an example, where we ask the user to enter student names:

make sure to tell the user how to stop entering data

```
students = []
name = input("Please enter a student, or 'QUIT' to stop: ")

while name != "QUIT":
    students.append(name)
    name = input("Please enter a student, or 'QUIT' to stop: ")
```

make sure to save the value before asking for the next one

Priming Reads

- This loop example uses a ***priming read***
  - We "prime" the loop by reading in information before the loop runs the first time

- We duplicate the line of code asking for input
  - Once before the loop
  - Once inside the loop

Loop Without a Priming Read

- It's also possible to do a sentinel loop without a priming read
- Instead of duplicating the input, we must duplicate something else… what?
  - The conditional that is checking the sentinel
- We must also declare the sentinel variable and a starting value before the loop begins

Example Without a Priming Read

```
students = []
name = ""       # can be set to anything other than 'QUIT'

while name != "QUIT":
    name = input("Please enter a student, or 'QUIT' to stop: ")

    # check if it's a sentinel before appending to list
    if name != "QUIT":
        students.append(name)
```

- If we don't check, we will add a student called "QUIT" to the list before exiting the loop

Sentinel Loop Style Choice

- You can use either sentinel loop style

- Priming read
  - Requires duplication of input
  - Fewer lines of code overall
- Non-priming read
  - Requires duplication of conditional checking
  - All of the data input happens inside the loop

Time for…

LIVECODING!!!

Livecoding: Updated Grocery List

- Let's update our grocery list program to be as long as the user wants, using a while loop and a sentinel value of "STOP"
  - Print out the grocery list (item by item) at the end
- You will need to use:
  - At least one while loop (a sentinel loop)
  - Conditionals
  - A single list

Other List Operations

Previously Seen Operations

- Many of the operations we saw on strings are possible with lists
- Which of the following works with lists?
  - Concatenation (**+**)
  - Repetition (*****)
  - Indexing
  - Slicing
  - **`.lower()`** and **`.upper()`**
  - **`len()`**

Concatenation

- Concatenation does work on lists!
  - But it has the same limit as string concatenation
  - You can only concatenate lists with lists

- So this works:

```
bookList + supplyList
```

- But this doesn't:

```
animalList + "horse"
```

Repetition

- Repetition does work on lists!

```
>>> animalList = ["dog", "cat", "ferret"]
>>> print(animalList * 3)
```

- What will this print out?

```
['dog', 'cat', 'ferret', 'dog', 'cat',
'ferret', 'dog', 'cat', 'ferret']
```

- The list gets “added” together multiple times, so the order of the elements stays the same

Indexing

- Indexing does work on lists!
- In the exact same way it does for strings

- Can use negative or positive indexing

```
studentNames[16]
courseTitles[-4]
```

Slicing

- Slicing does work on lists!
- In the exact same way it does for strings

- Slicing goes "up to but not including" the end of the slice

```
>>> lst = [17, "A", -22, True, "Hello"]
>>> print( lst[1:3] )
['A', -22]
```

`.lower()` and `.upper()`

- These operations do not work on lists!
  - They don't make sense for a list
- In the same way, `.append()` and `.remove()` don't work on strings

- If you try, you get an error about attributes:

```
AttributeError: 'str' object has no
attribute 'remove'
```

`.len()`

- Calling **`len()`** does work on lists!
- In the exact same way it does for strings

- Returns the length of the list
  - In other words, the number of elements

Two-Dimensional Lists

Two-Dimensional Lists

- We've looked at lists as being one-dimensional
  - But lists can also be two- (or three- or four- or five-, etc.) dimensional!
- Lists can hold any type (int, string, float, etc.)
  - This means they can also hold another list

Two-Dimensional Lists: A Grid

- It may help to think of 2D lists as a grid

```
twoD = [ [1,2,3], [4,5,6], [7,8,9] ]
```

1	2	3
4	5	6
7	8	9

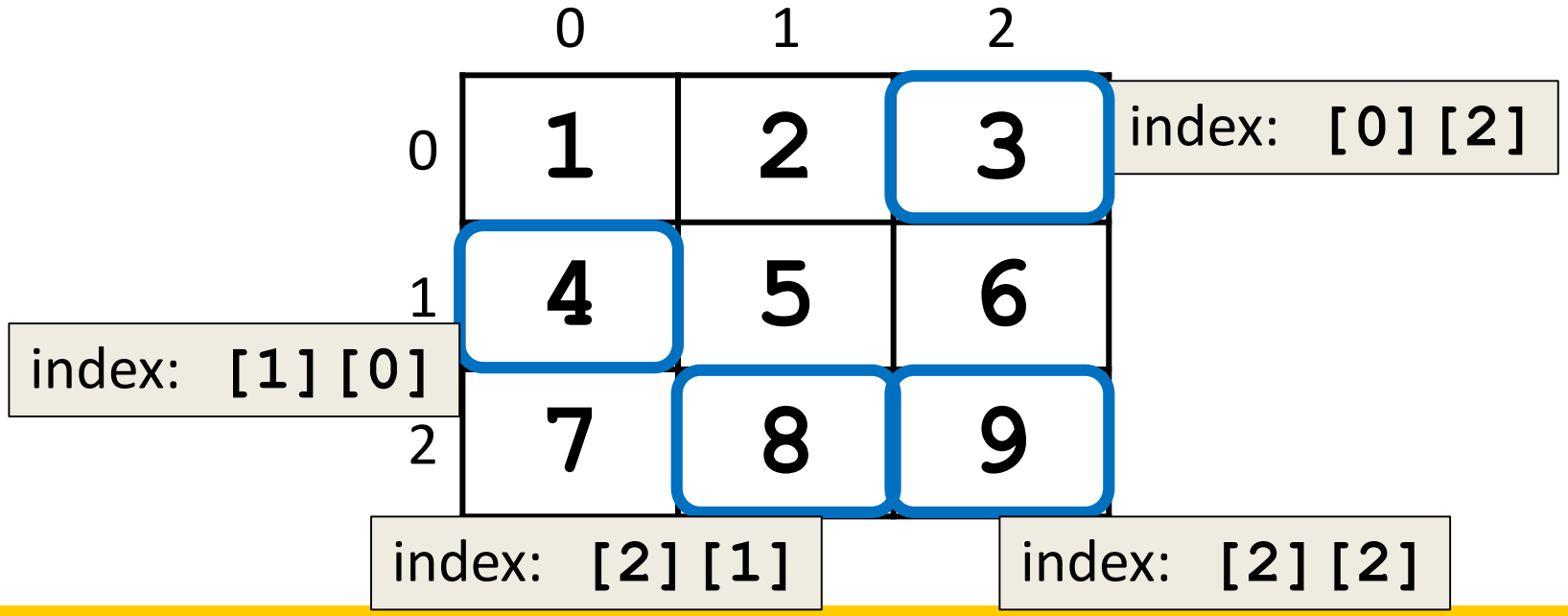
Two-Dimensional Lists: A Grid

- You access an element by the index of its row, and then the column
  - Remember – indexing starts at 0!

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Two-Dimensional Lists: A Grid

- You access an element by the index of its row, and then the column
  - Remember – indexing starts at 0!



Lists of Strings

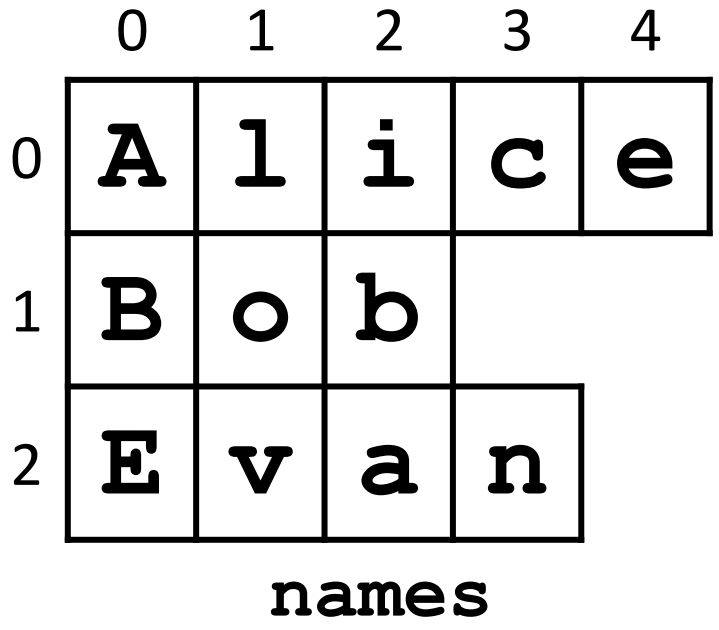
- Remember, a string is like a list of characters
- So what is a list of strings?
  - Like a two-dimensional list!

- We have the index of the string (the row)
- And the index of the character (the column)

Lists of Strings

- Lists in Python don't have to be rectangular
  - They can also be jagged (rows different lengths)

- Anything we could do with a one-dimensional list, we can do with a two-dimensional list
  - Slicing, index, appending



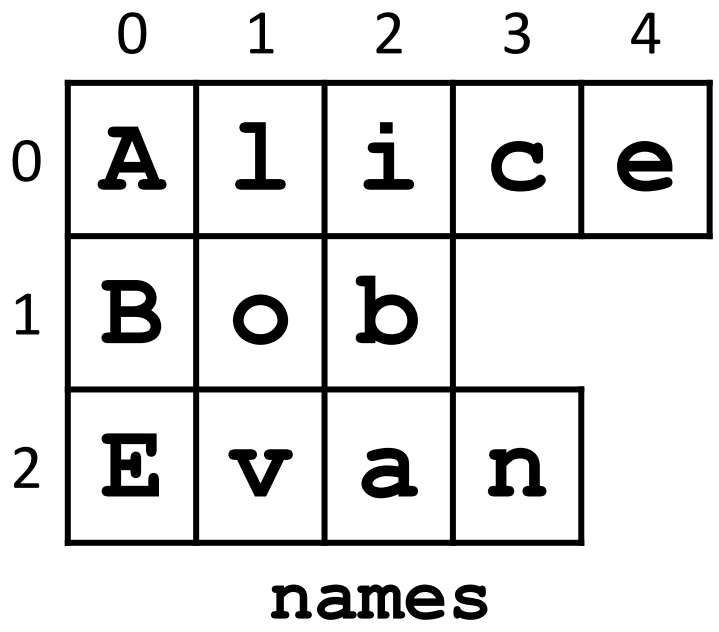
Practice: List of Strings

1. Add a “b” and a “y” to the end of “Bob”
2. Print out the second letter in “Evan”
3. Change “Alice” to “Alyce”

```
names[1] = names[1] + "b"
names[1] = names[1] + "y"

print(names[2][1])

names[0] = "Alyce"
```



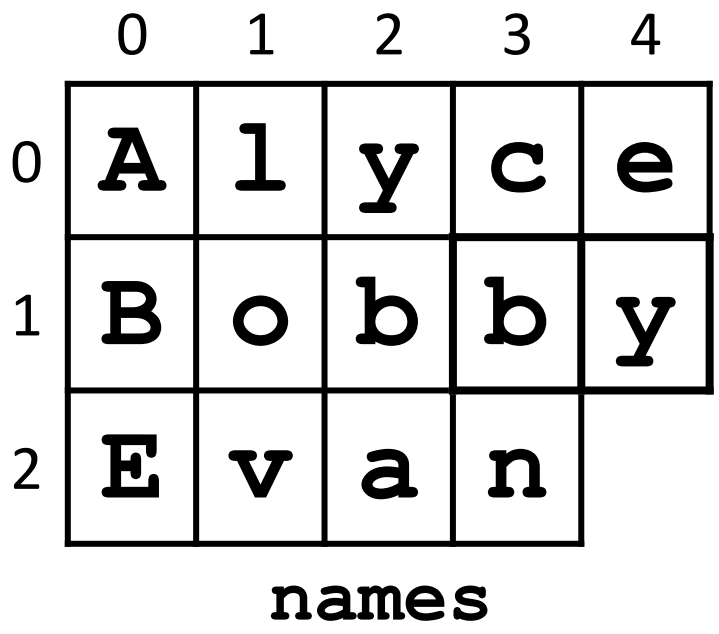
Practice: List of Strings (Advanced)

1. Add “Ahmed” to the end of the list of names
2. Add “Eve” to the front of the list of names
3. Change “Evan” to “Kevin”

```
names.append("Ahmed")

names = ["Eve"] + names

names[3] = "Kevin"
# the location changed!
```



Practice Problems

- Create a directory inside your "**`201`**" folder, called "**`practice`**"; go into the new folder
  - If you already created "**`practice`**", no need to do so again
- Copy this file into your new folder

  /afs/umbc.edu/users/k/k/k38/pub/cs201/listPractice.py
- Complete the files according to its instructions

- Remember, the command to copy is "**`cp`**":

  cp /afs/umbc.edu/users/k/k/k38/pub/cs201/listPractice.py .