CS 134 Lecture 9:

Nested Lists

Announcements & Logistics

- **HW 4** due Monday at 10 pm
- **Lab 4** will be released today
  - Prelab will be posted but no penalty if not completed by start of lab
  - We will review the code for the prelab together at the start of lab
- **Lab 2 graded feedback**
  - Let us know if you have questions or concerns
  - Comments and coding style: comments (start with #) are one important part of writing good code --- documentation is essential
  - Comments vs docstrings: docstrings document the function *interface* (input parameters, expected return), comments document the function *body* (logic used to implement the interface

Do You Have Any Questions?

Last Time

- Introduced nested `for` loops
    - Discussed how to trace the execution of loops
    - Use more examples of the `range` sequence type
- Reviewed the role of `return` statements in code

Today's Plan

- Introduce and use **nested lists**
- More examples of iteration:
    - Iterate over nested sequences and collect/filter useful statistics
- Module vs scripts (if time)
    - How to import and test functions
    - Role of the special `if __name__ == "__main__":` code block

Nested Lists

Nested Lists

- Remember, any object can be an element of a list. This includes other lists!
- That is, we can have **lists of lists** (sometimes called a two-dimensional list)!
- Suppose we have a **list of lists of strings** called `myList`

Nested Lists

- Remember, any object can be an element of a list. This includes other lists!
- That is, we can have **lists of lists** (sometimes called a two-dimensional list)!
- Suppose we have a **list of lists of strings** called `myList`
- `word = myList[row][element] (# word is a string)`
  - `row` is index into "***outer***" list (identifies ***which inner list*** we want). In other words, defines the "row" you want.
  - `element` is index into "***inner***" list (identifies ***which element*** within the inner list). In other words, defines the "column" you want.

```
                  element
                     ↓
myList = [ ['cat', 'frog'],
           ['dog', 'toad'],  ←— row
           ['cow', 'duck'] ]
```

`myList[1][0]?`
`'dog'`

Lists and Data Types

- Python is a loosely typed programming language
  - We don't explicitly declare data types of variables
    - But every value still has a data type!
  - It's important to make sure we pay attention to what a function expects, especially with lists and strings! (remember this in Lab 4)

- **Lists of <u>lists</u> of strings** versus **list of strings**:

```
myList = [ ['cat', 'frog'],
           ['dog', 'toad'],
           ['cow', 'duck'] ]
```

```
myList = ['cat', 'frog',
          'dog', 'toad',
          'cow', 'duck']
```

```
myList[1][0] is 'dog'
```

```
myList[1][0] is 'f'
```

Sequence Operations

```
characters = [['Elizabeth Bennet', 'Fitzwilliam Darcy'],
              ['Harry Potter', 'Ron Weasley'],
              ['Frodo Baggins', 'Samwise Gamgee'],
              ['Julius Ceasar', 'Brutus']]

>>> len(characters)  # what is this?
4

>>> len(characters[0]) # what is this?
2

>>> characters += ['Rhett Butler', 'Scarllet O Hara']
[['Elizabeth Bennet', 'Fitzwilliam Darcy'],
 ['Harry Potter', 'Ron Weasley'],
 ['Frodo Baggins', 'Samwise Gamgee'],
 ['Julius Ceasar', 'Brutus'],
 'Rhett Butler',
 'Scarllet O Hara']
```

Be careful when concatenating lists of two different types

Looping Over Nested Lists

```
characters =
[['Elizabeth Bennet', 'Fitzwilliam Darcy', 'Charles Bingley'],
['Harry Potter', 'Ron Weasley', 'Hermoine Granger'],
['Frodo Baggins', 'Samwise Gamgee', 'Gandalf']]
```

```
for char_list in characters:
    print(char_list)
    for name in char_list:
        print(name)
```

Loops over the "outer lists"

Prints each inner list one by one

Prints each individual name one by one

Loops over the names in each "inner list"

Why Nested Lists?

- Nested Lists are useful to represent tabular data
    - Example: data stored in google sheets
- Each inner list is a row
- List of lists: collection of all rows (the whole table)
- Lets take an example of real data that we can store as list of lists

Oscar 2024 Example

Accumulation Pattern: `most_so_far`

- So far, we have seen examples of accumulation variable
    - Count number of occurrences of something: count_vowels
    - Collect sequences: vowel_seq, madlibs_puzzle_solution
- Often, we need to find more information about a list of data we are storing such as:
    - find the earliest publication date in a data about books
    - find the largest stat in data about sports, etc.
- To do so, we need to iterate through the list and maintain a new type of accumulation variable that keeps track of this information
    - We need to update it as we find out more information

Exercise: count_nominations

Write a function that takes a table and returns the number of times a target string appears as an entry in that table.

Exercise: most_nominations

Write a function that takes a table and returns the string that appears as an entry in that table *the most times*.

Modules vs Scripts

Importing Functions vs Running as a Script

- **Question.** If you only have function definitions in a file `funcs.py`, and run it as a script, what happens?

```
% python3 funcs.py
```

- For testing functions, we want to call /invoke them on various test cases, in Labs, we do this in a separate file called `runtests.py`
  - To add function calls in `runtests.py`, we put them inside the guarded block `if __name__ == "__main__":`
- The statements within this special guarded are only run when the file is run as a ***script*** but not when it is imported as a ***module***
- Let's see an example

```python
# foo.py
# test the role of __name__ variable
print("__name__ is set to", __name__)
```

Running foo.py as a **script**

```
shikhasingh@Shikhas-iMac cs134 % python3 foo.py
__name__ is set to __main__

shikhasingh@Shikhas-iMac cs134 % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021,
14:59:20) [Clang 12.0.5 (clang-1205.0.22.11)] on
darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> import foo
__name__ is set to foo
```

Importing it as a **module**

Takeaway: `if __name__ == "__main__"`

- If you want some statements (like test calls) to be run **ONLY when the file is run as a script**
    - Put them inside the guarded `if __name__ == "__main__"` block

- When we run our automatic tests on your functions we **import them** and this means name is NOT set to main
    - So nothing inside the guarded `if __name__ == "__main__"` block is executed
- This way your testing /debugging statements do not get in the way

Nested Lists Additional Examples

Nested Loops and Nested Lists

- Let us trace through the code below:

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

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

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
[[1,4,9]]	[1,2,3]	[1,4,9]	3

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

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

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
[[1 ,4 ,9], [16,25,36]]	[4,5,6]	[16,25,36]	6

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

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

Nested Loops

new_lstlsts	row	new_row	item
`[]`	`[1,2,3]`	`[]`	
		`[1]`	1
	`[1,2,3]`	`[1,4]`	2
	`[1,2,3]`	`[1,4,9]`	3
`[[1,4,9]]`	`[4,5,6]`	`[]`	
		`[16]`	4
	`[4,5,6]`	`[16,25]`	5
	`[4,5,6]`	`[16,25,36]`	6
`[[1 ,4 ,9 ],` `[16,25,36]]`	`[7,8,9]`	`[]`	
		`[49]`	7

`[[1 ,4 ,9 ],`
`[16,25,36],`
`[49,64,81]]`
	`[7,8,9]`	`[49,64]`	8

	`[7,8,9]`	`[49,64,81]`	9

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

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

Nested Loops

new_lstlsts	row	new_row	item
`[]`	`[1,2,3]`	`[]`	
		`[1]`	1
	`[1,2,3]`	`[1,4]`	2
`[[1,4,9]]`	`[1,2,3]`	`[1,4,9]`	3
	`[4,5,6]`	`[]`	
		`[16]`	4
	`[4,5,6]`	`[16,25]`	5
`[[1 ,4 ,9 ],` `[16,25,36]]`	`[4,5,6]`	`[16,25,36]`	6
	`[7,8,9]`	`[]`	
		`[49]`	7

`[[1 ,4 ,9 ],`
`[16,25,36],`
`[49,64,81]]`
	`[7,8,9]`	`[49,64]`	8

	`[7,8,9]`	`[49,64,81]`	9

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

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

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

```
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

```
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Why 2 accumulation variables?!

The square brackets ensure that we're adding a **`list to a list`**!

The inner loop accumulates the items for the row, the outer loop accumulates the rows

What would be a good function name for `mystery2`?

Something like `power_table`