SAMS Programming - Section C

Week 4 - Lecture 1:
“2-dimensional” lists
Pop Quiz
Fill in the blank:

Lists are **awesome**.

T/F: A variable stores the value of an object.

T/F: To make a copy of the list `a = [1, 2, 3]`, do

```python
b = a  # a and b are aliases
b = copy.copy(a)
```

What will the following print?

```python
a = [1, 2, 3]
b = copy.copy(a)
print(a == b, a is b)
```
Pop Quiz

Fill in the blank:

List parameters and arguments are **awesome**.
Pop Quiz

Fill in the blank:

List parameters and arguments are **awesome**.

```python
def fill(a, value):
    for i in range(len(a)):
        a[i] = value

x = [1, 2, 3]
fill(x, 42)
print(x)  # [42, 42, 42]
```

Destructive function
Pop Quiz

Fill in the blank:

List parameters and arguments are **awesome**.

```python
def fill(a, value):
    a = copy.copy(a)
    for i in range(len(a)):
        a[i] = value
    return a
```

```python
x = [1, 2, 3]
y = fill(x, 42)
print(x, y)  # [1, 2, 3] [42, 42, 42]
```

Nondestructive version
Is the sorted function destructive?

```python
a = [5, 4, 3, 2, 1]
b = sorted(a)
print(a, b)  # [5, 4, 3, 2, 1] [1, 2, 3, 4, 5]
```

Is the sort method destructive?

```python
a = [5, 4, 3, 2, 1]
b = a.sort()
print(a, b)  # [1, 2, 3, 4, 5] None
```
Pop Quiz

How do you convert a string to a list?

```python
s = "You suck anil!"
print(list(s))  # ['Y', 'o', 'u', ' ', 's', 'u', 'c', 'k', ' ', 'a', 'n', 'i', 'l', '!' ]
print(s.split(" ")))  # ['You', 'suck', 'anil!']
```

How do you convert a list of strings into one string?

```python
a = ["Stephen", "is", "awesome"]
print("".join(a))  # Stephenisawesome
print(" ").join(a))  # Stephen is awesome
print(",").join(a))  # Stephen,is,awesome
```
Pop Quiz

What does this print?

```python
a = [1, 2, 3]
b = a
a = a + [4]
print(a)    # [1, 2, 3, 4]
print(b)    # [1, 2, 3]
```

What does this print?

```python
a = [1, 2, 3]
b = a
a += [4]
print(a)    # [1, 2, 3, 4]
print(b)    # [1, 2, 3, 4]
```
Pop Quiz

What is the difference between pop and other destructive methods?

It makes a cool sound.
Pop Quiz

What is the difference between pop and other destructive methods?

It returns something.
“2d lists”
A list can contain any type of object.

```
a = [1, "hello", False]
```

Can also contain lists.

```
a = [[1, 3, 5], [6], [1, 5]]  # A list of lists
```

```
print(len(a))  3
```

- `a[0]` is a reference to the first list `[1, 3, 5]`
- `a[1]` is a reference to the second list `[6]`
- `a[2]` is a reference to the third list `[1, 5]`
- `a[0][0]` is a reference to the first element of the first list `[1, 3, 5]`
- `a[2][1]` is a reference to the second element of the third list `[1, 5]`
Example: Print all the elements

```
a = [[1, 3, 5], [6], [1, 5]]

for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j])
```
Most “2d lists” we deal with will have same length sublists.

\[
a = [[1, 3], [2, 4], [1, 5]]
\]

\[
a = [ [1, 3],
  [2, 4],
  [1, 5]
 ]
\]

Really like a table (or matrix)
2d list examples

A chess board: 8 lists of length 8 each (or 8 by 8 table)
   Each entry either contains a chess piece or is empty.

An image: a 2d list of points/pixels
   Each entry contains the color of the point.

A database: e.g. a list of users and various information about the users

| name | age | email | ...
|------|-----|-------|------
| user1|     |       |      |
| user2|     |       |      |
| user3|     |       |      |
|      |     |       |      |
Cool. Seems easy enough. Can we go home?

Unfortunately, no. 😞
Tricky thing about 2d lists

1d list: references to **immutable** objects.
   Aliases of elements not a problem.

2d list: references to **mutable** objects.
   We must be careful about aliases of elements !!
“Weird” Example 1

```python
a = [1, 2, 3]
b = copy.copy(a)
b[0] = 0
print(a)  # [1, 2, 3]
print(b)  # [0, 2, 3]

a = [[1, 2, 3], [4, 5, 6]]
b = copy.copy(a)
b[0][0] = 0
print(a)  # [[0, 2, 3], [4, 5, 6]]
print(b)  # [[0, 2, 3], [4, 5, 6]]```

a = [ [0]*2 ]*3
print(a)  # [[0, 0], [0, 0], [0, 0]]

a[0][0] = 9

print(a)  # [[9, 0], [9, 0], [9, 0]]
a = [1, 2, 3]
b = copy.copy(a)
b[0] = 0
print(a[0])
print(b[0])

Making a copy of the references.
a = [1, 2, 3]
b = copy.copy(a)
b[0] = 0
print(a[0])
print(b[0])

Making a copy of the references.
a = [[1, 2, 3], [4], [5, 6]]
b = copy.copy(a)
b[0][0] = 0
print(a[0][0])
print(b[0][0])
Understanding Example 1

```python
a = [[1, 2, 3], [4], [5, 6]]
b = copy.copy(a)
b[0][0] = 0
print(a[0][0])
print(b[0][0])
```
a = [[1, 2, 3], [4], [5, 6]]
b = a[:]
b[0][0] = 0
print(a[0][0])
print(b[0][0])
Understanding Example 1

```python
a = [[1, 2, 3], [4], [5, 6]]
b = a + []
b[0][0] = 0
print(a[0][0])
print(b[0][0])
```

**copy.copy**, **+** and **list slices** make **shallow copies**
Understanding Example 1

```python
a = [[1, 2, 3], [4], [5, 6]]
b = copy.deepcopy(a)
b[0][0] = 0
print(a[0][0])
print(b[0][0])
```

```
1 2 3
4
5 6
```
Understanding Example 1

\[ a = \begin{bmatrix} 1, 2, 3, & 4, & 5, 6 \end{bmatrix} \]

\[ b = \text{copy.deepcopy}(a) \]

\[ b[0][0] = 0 \]

\[ \text{print}(a[0][0]) \]

\[ \text{print}(b[0][0]) \]
Understanding Example 2

\[ a = [0]*2 \]
a = [0]*4

a[0] = 1
# Create a 3 by 2 list

```python
a = [ [0]*2 ]*3
```

```
a
```

```
0
```
# Create a 3 by 2 list

```python
a = [ [0]*2 ]*3
[ [0, 0], [0, 0], [0, 0] ]
```

```python
a[0][0] = 1
```

print(a)

```python
[ [1, 0], [1, 0], [1, 0] ]
```

*a[0], a[1], and a[2] are aliases!*
Creating a rows by cols 2d list

cols = 3
rows = 2

```python
a = []
for row in range(rows):
    a += [[0]*cols]
```

```python
a += [[0, 0, 0]]
a += [[0, 0, 0]]
```

```
a
```

```
  0
```

```
```
Creating a rows by cols 2d list

Define a function for this task.

```python
def make2dList(rows, cols):
a = []
    for row in range(rows):
        a += [[0]*cols]
return a
```
# Create a 3 by 2 list

```python
a = [ [0]*2 ]*3
```

Trying to break aliasing with `deepcopy`:

```python
a = copy.deepcopy(a)
```

deprecated preserves alias structure!

see `myDeepCopy` in the notes.
A list operation or function that makes a copy (e.g. +, list slicing, sorted function) makes a **shallow copy**.

* operation creates aliases. Don’t use it to create 2d lists.

Never use **copy** with 2d lists.

- creates aliases
- ok to use with 1d lists since elements are immutable.

Remember: **deepcopy** does not break alias structure within the list.
3d Lists

\[a1 = \begin{bmatrix} [1, 2] \\ [3, 4] \end{bmatrix}\]
\[a2 = \begin{bmatrix} [5, 6, 7] \\ [8, 9] \end{bmatrix}\]
\[a3 = \begin{bmatrix} [10] \end{bmatrix}\]

**3d list:**
\[a = [a1, a2, a3]\]

**4d list:**
\[a = [a, a]\]
3d Lists

```
a = [
    [[1, 2],
     [3, 4]],
    [[5, 6, 7],
     [8, 9]],
    [[10]]
]

Printing elements of 3d lists:

for i in range(len(a)):
    for j in range(len(a[i])):
        for k in range(len(a[i][j])):
            print("a[%d][%d][%d] = %d" % (i, j, k, a[i][j][k]))
```
Example Problem: Word Search

def testWordSearch():
    board = [ [ 'd', 'o', 'g' ],
              [ 't', 'a', 'c' ],
              [ 'o', 'a', 't' ],
              [ 'u', 'r', 'k' ],
            ]
    print(wordSearch(board, "dog"))        # ('dog', (0, 0), 'right')
    print(wordSearch(board, "cat"))         # ('cat', (1, 2), 'left')
    print(wordSearch(board, "tad"))         # ('tad', (2, 2), 'up-left')
    print(wordSearch(board, "cow"))         # None
def wordSearch(board, word):
    # ...

Algorithm:  wordSearch(board, word)

- go through each cell of the board one by one:
  - check if word appears starting at that cell
def wordSearch(board, word):
    # ...

Algorithm: \texttt{wordSearch(board, word)}

- go through each \texttt{cell} of the board one by one:
  - check if word appears starting at that \texttt{cell}

needs to be broken down further
Example Problem: Word Search

def wordSearchFromCell(board, word, startRow, startCol):
    # ...

**Algorithm:** wordSearchFromCell(board, word, startRow, startCol)

- go through each **direction** one by one:
  - check if word appears in that **direction** starting at the given cell
def wordSearchFromCell(board, word, startRow, startCol):
    # ...

Algorithm: wordSearchFromCell(board, word, startRow, startCol)

- go through each direction one by one:
  - check if word appears in that direction starting at the given cell

needs to be broken down further
it is important how you represent direction.

let's see an elegant way of doing it…