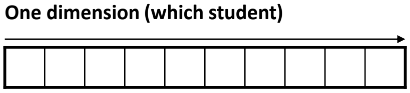
# Composite Types, Lists Part 2

- Multi-dimensional lists: when to use them, basic 2D list operations (creation, access, modification, display, copy).
- Using named constants to stay within list bounds.
- Dynamically creating 2D lists with the append function.

## When To Use Lists Of Different Dimensions

- It's determined by the data – the number of categories of information determines the number of dimensions to use.
- Examples:
- (1D list)
  - –Tracking grades for a class (previous example)
  - –Each cell contains the grade for a student i.e., `grades[i]`
  - –There is one dimension that specifies which student's grades are being accessed



- (2D list)
  - –Expanded grades program (table: grades for multiple lectures)
  - –Again there is *one dimension* that specifies which student's grades are being accessed
  - –The *other dimension* can be used to specify the lecture section

## When To Use Lists Of Different Dimensions (2)

- (2D list continued)

**Student**

**Lecture section**

| | First student | Second student | Third student | … |
|---|---|---|---|---|
| L01 | | | | |
| L02 | | | | |
| L03 | | | | |
| L04 | | | | |
| L05 | | | | |
| : | | | | |
| L0N | | | | |

## When To Use Lists Of Different Dimensions (3)

- (2D list continued)
- Notice that each row is merely a 1D list
- (A 2D list is a list containing rows of 1D lists)

**2D list access:**

- List elements are specified in the order of `[row] [column]`
- Specifying only a single set of brackets specifies the row

**Columns (e.g. grades)**

| | | [0] | [1] | [2] | [3] |
|---|---|---|---|---|---|
| [0] | L01 | | | | |
| [1] | L02 | | | | |
| [2] | L03 | | | | |
| [3] | L04 | | | | |
| [4] | L05 | | | | |
| [5] | L06 | | | | |
| [6] | L07 | | | | |

**Rows (e.g. lecture section)**

# Creating And Initializing A Multi-Dimensional List In Python (Fixed Size During Creation)

**General structure**

```
<list_name> = [ [<value 1>,  <value 2>, ... <value n>],
                [<value 1>,  <value 2>, ... <value n>],
                          : :        :
                          : :        :
                [<value 1>,  <value 2>, ... <value n>] ]
```

Rows

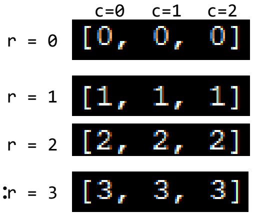
Columns

# Creating And Initializing A Multi-Dimensional List In Python (2): Fixed Size During Creation

**Name of the example program**: `1display2DList.py`

Learning: creating, displaying a fixed size 2D list

```
table = [ [0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]

for r in range (0, 4, 1):
   print (table[r])  #Each call to print displays a 1D list
```

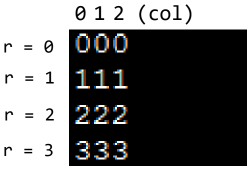


```
       c=0   c=1   c=2
r = 0  [0,  0,  0]
r = 1  [1,  1,  1]
r = 2  [2,  2,  2]
r = 3  [3,  3,  3]
```

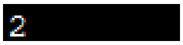
**2D list access:**

- List elements are specified in the order of `[row] [column]`
- Specifying only a single set of brackets specifies the row

```
for r in range (0,4,1):
   for c in range (0,3,1):
       print(table[r][c], end="")
   print()
            #Displays a list element

print(table[2][0]) #Displays 2 not 0
```



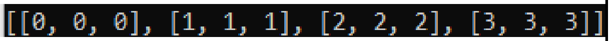
```
       0 1 2 (col)
r = 0  000
r = 1  111
r = 2  222
r = 3  333
```



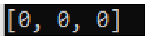
```
2
```

## 2D Lists: Levels Of Access

```
table = [ [0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]
print(table)  #Entire list
print(table[0]) #First row
print(table[3][1]) #4th row, 2nd column
print(table[0][0][0]) #What does this do?
```

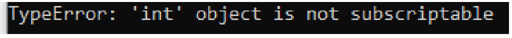


```
[[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3]]
```



```
[0, 0, 0]
```

```
3
```



```
TypeError: 'int' object is not subscriptable
```

```
table = [ [["a","b"], 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]

print(table[0][0][0]) #Now what does this do?
```

James Tam

## Creating 2D Lists Via The Repetition Operator

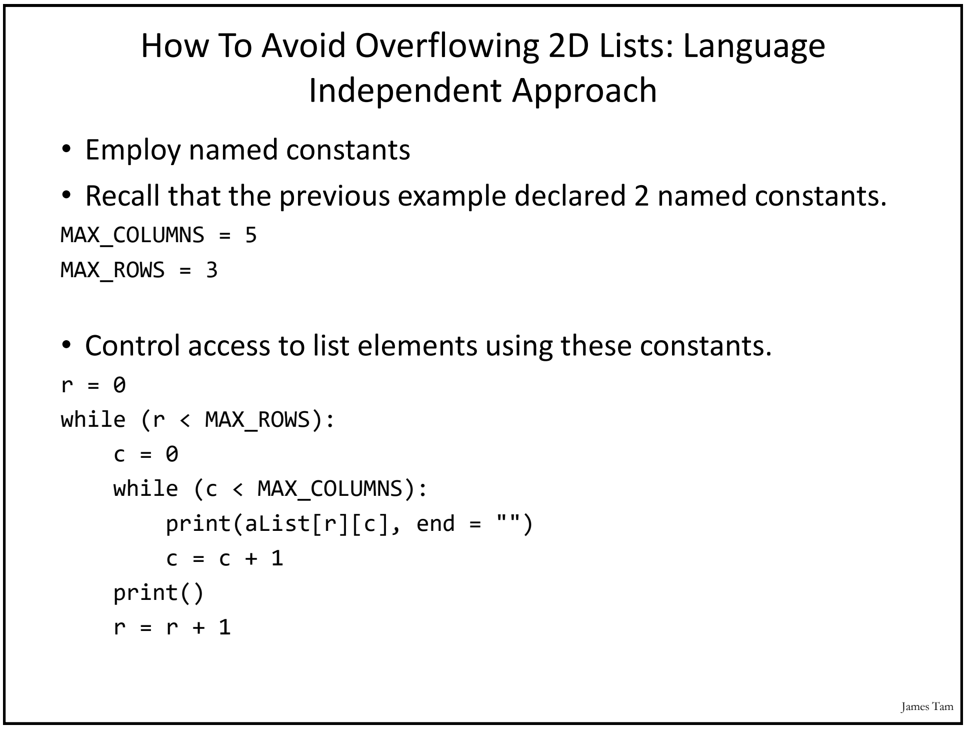
**Name of the example program**: `2creatingListViaRepetition.py`

Learning:

- Creating a variable sized 2D list using the repetition operator and the append method.
- The 2D list is created by **creating a 1D list** and **appending the 1D list to the end of the 2D list**.

```
MAX_COLUMNS = 5
MAX_ROWS = 3
ELEMENT = "*"
aList = []
r = 0
while (r < MAX_ROWS):
   tempList = [ELEMENT] * MAX_COLUMNS
   aList.append(tempList)
   r = r + 1
```

James Tam



## How To Avoid Overflowing 2D Lists: Language Independent Approach

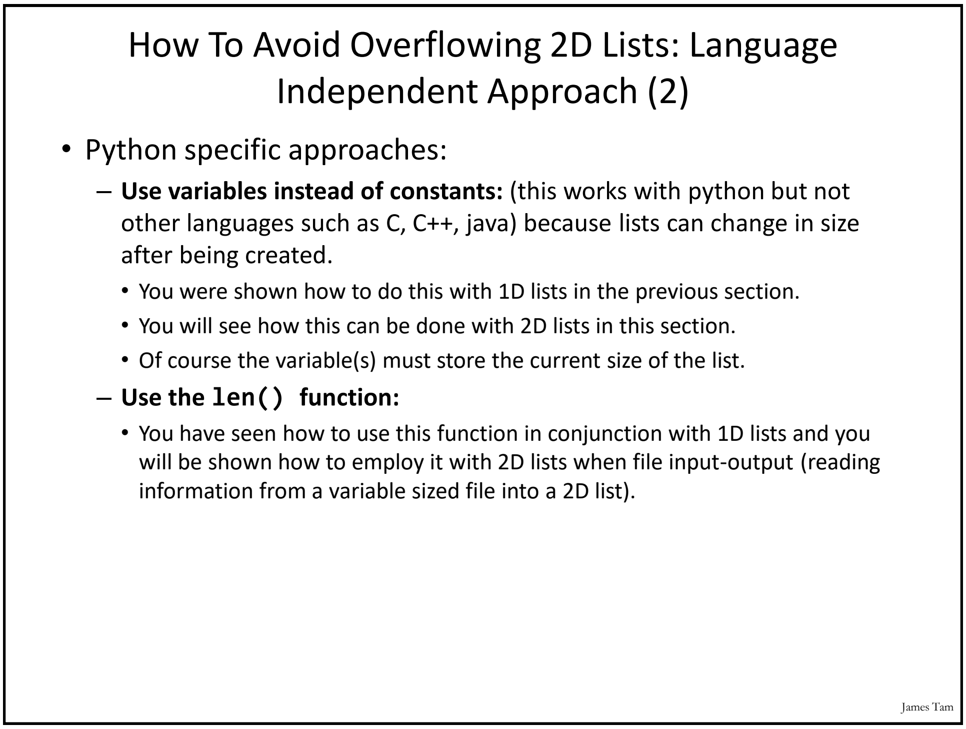
- Employ named constants
- Recall that the previous example declared 2 named constants.

```
MAX_COLUMNS = 5
MAX_ROWS = 3
```

- Control access to list elements using these constants.

```
r = 0
while (r < MAX_ROWS):
    c = 0
    while (c < MAX_COLUMNS):
        print(aList[r][c], end = "")
        c = c + 1
    print()
    r = r + 1
```

James Tam



## How To Avoid Overflowing 2D Lists: Language Independent Approach (2)

- Python specific approaches:
  - **Use variables instead of constants:** (this works with python but not other languages such as C, C++, java) because lists can change in size after being created.
    - You were shown how to do this with 1D lists in the previous section.
    - You will see how this can be done with 2D lists in this section.
    - Of course the variable(s) must store the current size of the list.
  - **Use the `len()` function:**
    - You have seen how to use this function in conjunction with 1D lists and you will be shown how to employ it with 2D lists when file input-output (reading information from a variable sized file into a 2D list).

James Tam

## Copying Lists

- Important: A variable that appears to be a list is really a reference to a list.
  - Recall: the reference and the list are two separate memory locations!

```
matrix = [ [0, 0, 0],
           [1, 1, 1],
           [2, 2, 2],
           [3, 3, 3]]
```

  - Wrong way to 'copy' a 2D list

`aList1 = aList2` (Why is this wrong? Hint: recall what is stored in `aList1` and `aList1`)

James Tam

## Copying Lists: Example

- **Name of the example program:** `3copyingListsBothWays.py`
- This is the **wrong way**.

```
aGrid1 = create()
aGrid2 = aGrid1
aGrid1[3][3] = "!"
print("First list")
display(aGrid1)
print("Second list")
display(aGrid2)
```

```
# FYI:
def create():
    aGrid = [["*","*","*","*"],
             ["*","*","*","*"],
             ["*","*","*","*"],
             ["*","*","*","*"]]
    return(aGrid)
```

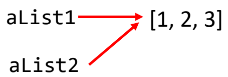
James Tam

# New Terminology

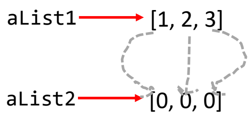
- **Shallow copy ("wrong way")**: copies what's stored in the reference (location of a list).

**Code**

```
aList1 = [1,2,3]
aList2 =aList1
```



- **Deep copy (correct way)**: copies the data from one list to another.
  - Create a new list e.g. aList2 = [0]*3
  - Copy each piece of data (list elements) from one list to another e.g. `aList2[0] = aList1[0]` (use a loop to copy all elements)



## Creating A New List By Copying An Existing List

- This is not a comprehensive list of approaches for copying
- Assume we have this list:

```
list1 = [1,2,3]
```

  - **Method 1 (python specific):** Utilize one of the prebuilt python methods for copying a list (if you don't know which one to use then make sure it performs a "deep copy").
    - **Check assignment requirements to see if this approach is allowed.**
  - **Method 2 (python specific):** write the code yourself using a FOR-loop

```
for element in list1:
    list2.append(element) #Append element from one list to another
```

  - **Method 3(language independent):** write the code yourself using a WHILE-loop.

```
i = 0
list2 = []
size = len(list1)
while(i<size):
    list2.append(list1[i]) #Append element from one list to another
    i = i + 1
```

## **Copying Lists**: Example (2)

- This is the **right way**.

```
aGrid1 = create()
aGrid2 = create()
copy(aGrid1,aGrid2)
```

```
def copy(destination,source):
    for r in range (0,SIZE,1):
        for c in range (0,SIZE,1):
            destination[r][c] = source[r][c]
```

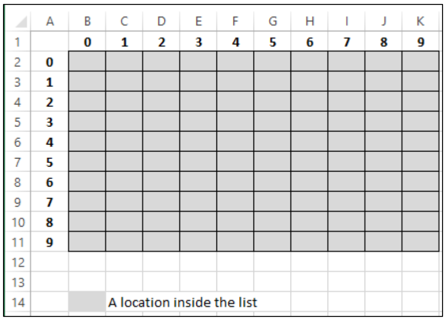
```
copy(aGrid1,aGrid2)
aGrid1[0][0] = "?" #These statements prove there’s two lists
aGrid1[3][3] = "?"
print("First list")
display(aGrid1)
print("Second list")
display(aGrid2)
```

## Copying Lists: Write The Code Yourself

- General rule of thumb: you should not use some else’s pre-created list copy method (e.g. those defined when you “`import copy`”)
- Why do all this work?
  - Not all programming languages have this capability (you will need to know how to do it yourself).
  - Writing the code yourself will provide you with extra practice and help you become more familiar with list (in other languages ‘array’) operations.

## Boundary Checking Lists

- Checking if a particular location (row, column) for a 2D list is inside the bounds of the list is a common program task.



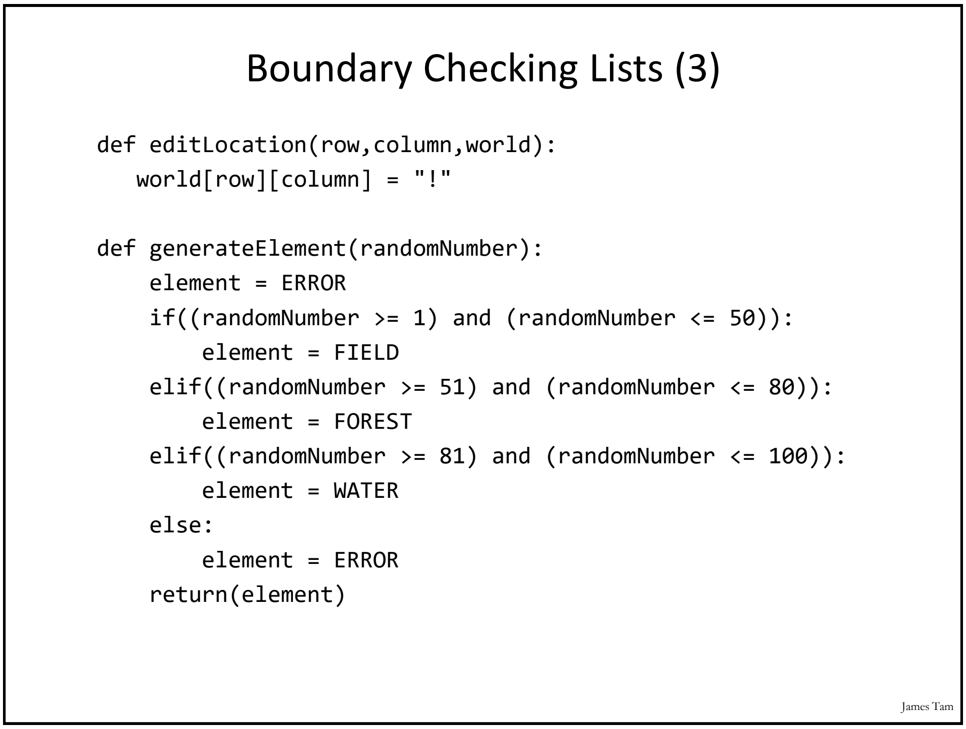
| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 2 | **0** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 3 | **1** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 4 | **2** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 5 | **3** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 6 | **4** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 7 | **5** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 8 | **6** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 9 | **7** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 10 | **8** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 11 | **9** | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ |
| 12 | | | | | | | | | | | |
| 13 | | | | | | | | | | | |
| 14 | | ▨ | A location inside the list | | | | | | | | |

- Rather than repeating the check it may be more efficient to write one Boolean function to implement this task.

## Boundary Checking Lists (2)

- **Name of the example**: `4boundary_checking`

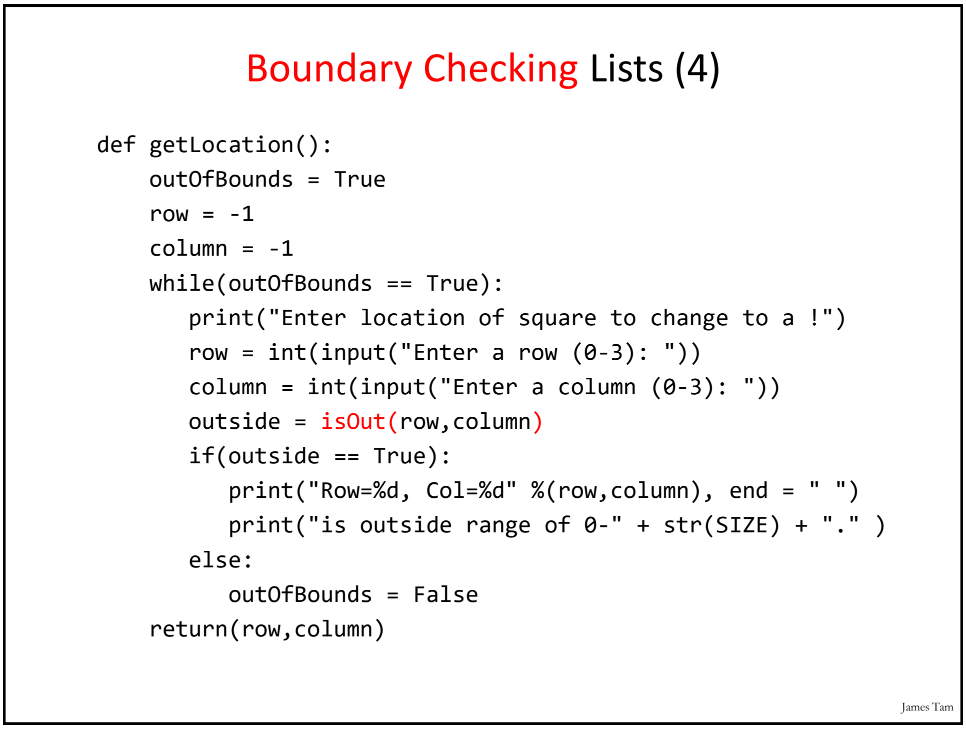
```
SIZE = 4
FIELD = " "
FOREST = "^"
WATER = "W"
BURNT = "F"
ERROR = "!"

def display(world):
   r = -1
   c = -1
   for r in range (0,SIZE,1):
      for c in range (0,SIZE,1):
         print(world[r][c], end="")
      print()
   print()
```



## Boundary Checking Lists (3)

```
def editLocation(row,column,world):
   world[row][column] = "!"

def generateElement(randomNumber):
    element = ERROR
    if((randomNumber >= 1) and (randomNumber <= 50)):
        element = FIELD
    elif((randomNumber >= 51) and (randomNumber <= 80)):
        element = FOREST
    elif((randomNumber >= 81) and (randomNumber <= 100)):
        element = WATER
    else:
        element = ERROR
    return(element)
```

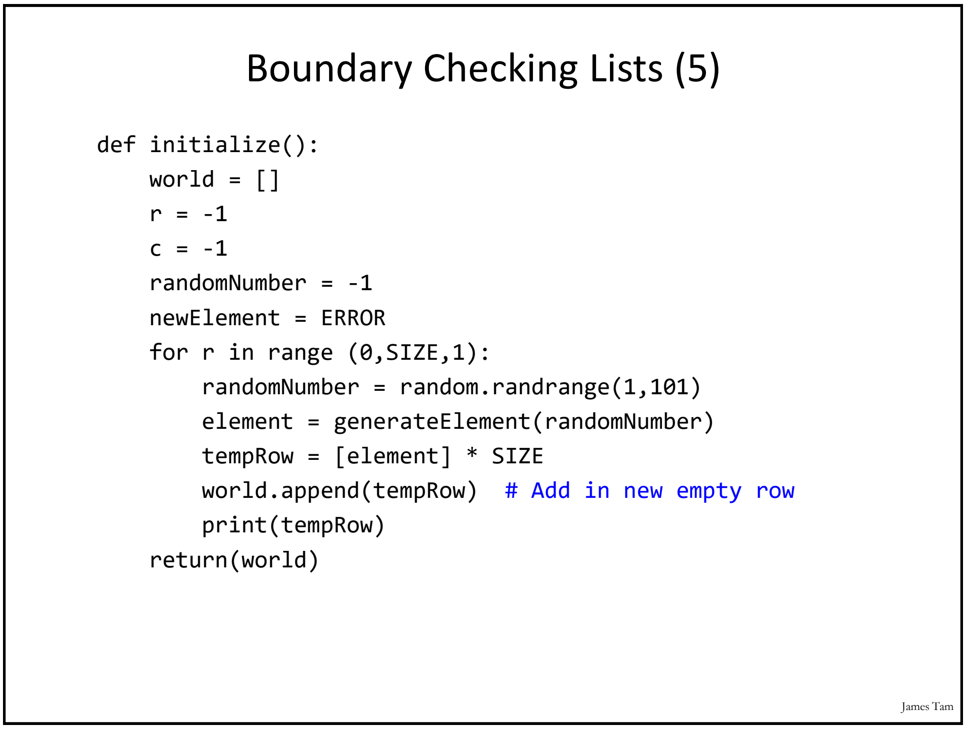
James Tam



## Boundary Checking Lists (4)

```
def getLocation():
    outOfBounds = True
    row = -1
    column = -1
    while(outOfBounds == True):
       print("Enter location of square to change to a !")
       row = int(input("Enter a row (0-3): "))
       column = int(input("Enter a column (0-3): "))
       outside = isOut(row,column)
       if(outside == True):
          print("Row=%d, Col=%d" %(row,column), end = " ")
          print("is outside range of 0-" + str(SIZE) + "." )
       else:
          outOfBounds = False
    return(row,column)
```

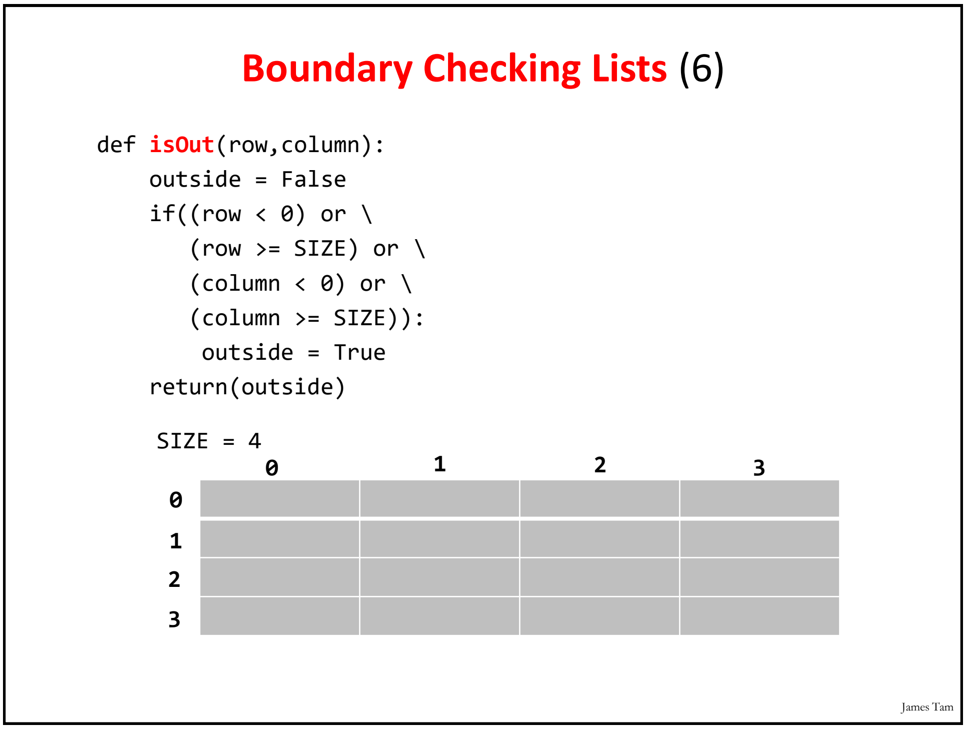
James Tam



## Boundary Checking Lists (5)

```
def initialize():
    world = []
    r = -1
    c = -1
    randomNumber = -1
    newElement = ERROR
    for r in range (0,SIZE,1):
        randomNumber = random.randrange(1,101)
        element = generateElement(randomNumber)
        tempRow = [element] * SIZE
        world.append(tempRow)  # Add in new empty row
        print(tempRow)
    return(world)
```

James Tam



## Boundary Checking Lists (6)

```
def isOut(row,column):
    outside = False
    if((row < 0) or \
       (row >= SIZE) or \
       (column < 0) or \
       (column >= SIZE)):
        outside = True
    return(outside)
```

SIZE = 4

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

James Tam

# Boundary Checking Lists (7)

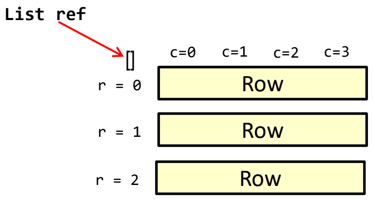
```
def start():
    stillRunning = True
    answer = ""
    row = -1
    column = -1
    world = initialize()
    while(stillRunning): #while(stillRunning == True):
        display(world)
        row,column = getLocation()
        editLocation(row,column,world)
        answer = input("Hit enter to continue,'q' to quit: ")
        if((answer == "q") or (answer == "Q")):
            stillRunning = False

start()
```

James Tam

# Creating And Initializing A Multi-Dimensional List In Python: Dynamic Creation

**General structure (Using loops)**:

- Create a variable that refers to an empty list
- Create list:
  - One loop (outer loop) traverses the rows.
  - **Each iteration of the outer loop creates a new 1D list (empty at start)**
  - **Then the inner loop traverses the elements of the newly created 1D list** creating and **initializing each element** in a fashion similar to how a single 1D list was created and initialized (add to end)
- Repeat the process for each row in the list



Etc.

```
aGrid = []
for r in range (0, 3, 1):
    aGrid.append ([])
    for c in range (0, 3, 1):
        aValue = <Some source>
        aGrid[r].append(aValue)
```

## Repeating Just The Steps In The Code Creating The List

Recall 'append' is unique to a list. **Append won't work if for other types of variables except list** but even an empty list can have new elements appended.

```
num = 123
num.append(4) #error
```

1. Create a variable that refers to an empty list

```
aGrid = []
```

2. Successively create rows in the list

```
for r in range (0,noRows,1):
    aGrid.append ([])
```

3. Each row is a 1D list, add elements to the end of the 1D list (empty list needed in #2 so that the append method can be called to add elements to the end).

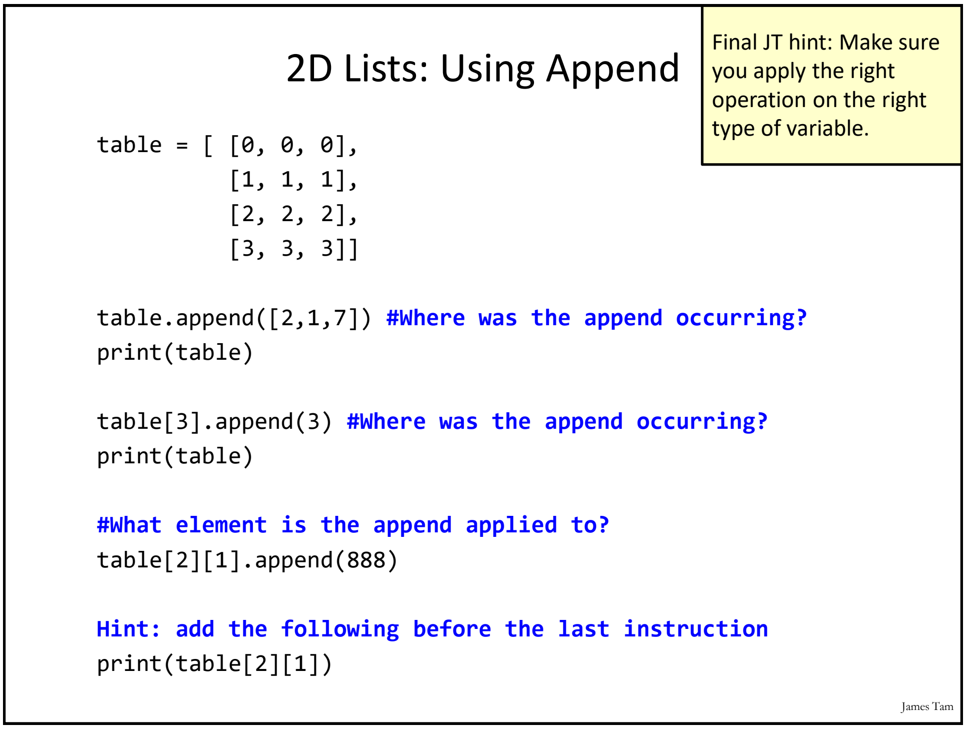
```
    for c in range (0,noColumns,1):
       aGrid[r].append("*")
```

- The [r] part of specifies which row the loop will add elements on the end. `aGrid[r].append("*")`

## Example 2D List Program: A Variable Sized 2D List (Dynamic)

- **Name of the example program:** `5variableSize2DList.py`

```
aGrid = []
noRows = int(input("Number rows: "))
noColumns = int(input("Number columns: "))
#Create list
for r in range (0,noRows,1):
    aGrid.append ([]) #Create empty row, add to list
    for c in range (0,noColumns,1):
        element = input("Type in a single character: ")
        aGrid[r].append(element) #Add to the end of new row
#Display list
for r in range (0,noRows,1):
   for c in range (0,noColumns,1):
      print(aGrid[r][c], end="")
   print()
```

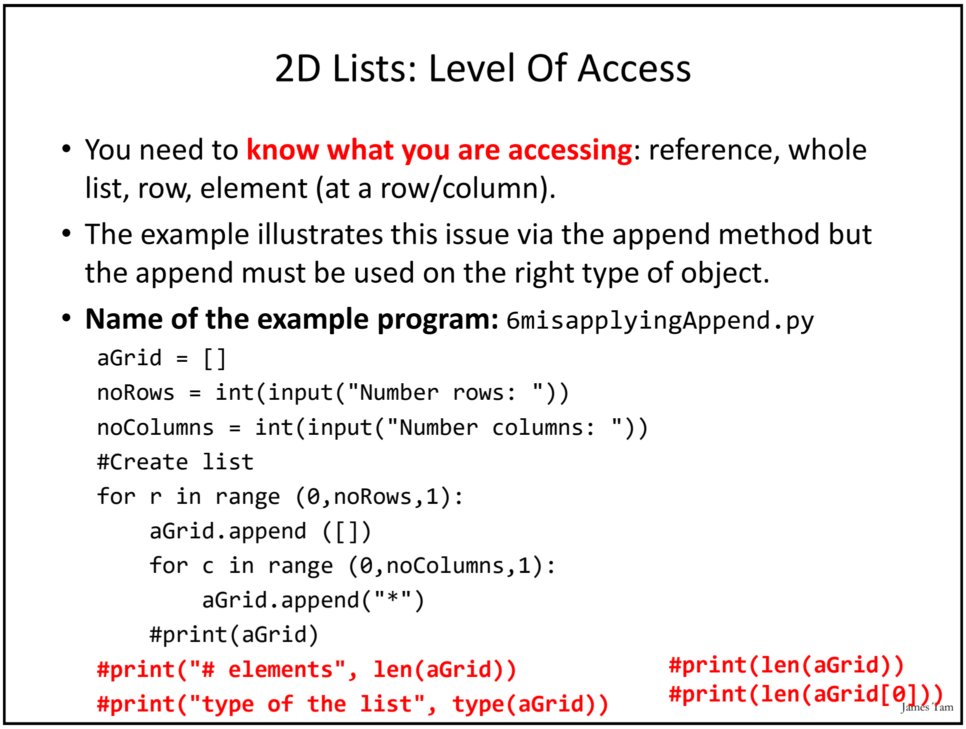


## 2D Lists: Using Append

Final JT hint: Make sure you apply the right operation on the right type of variable.

```
table = [ [0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]

table.append([2,1,7]) #Where was the append occurring?
print(table)

table[3].append(3) #Where was the append occurring?
print(table)

#What element is the append applied to?
table[2][1].append(888)

Hint: add the following before the last instruction
print(table[2][1])
```

James Tam



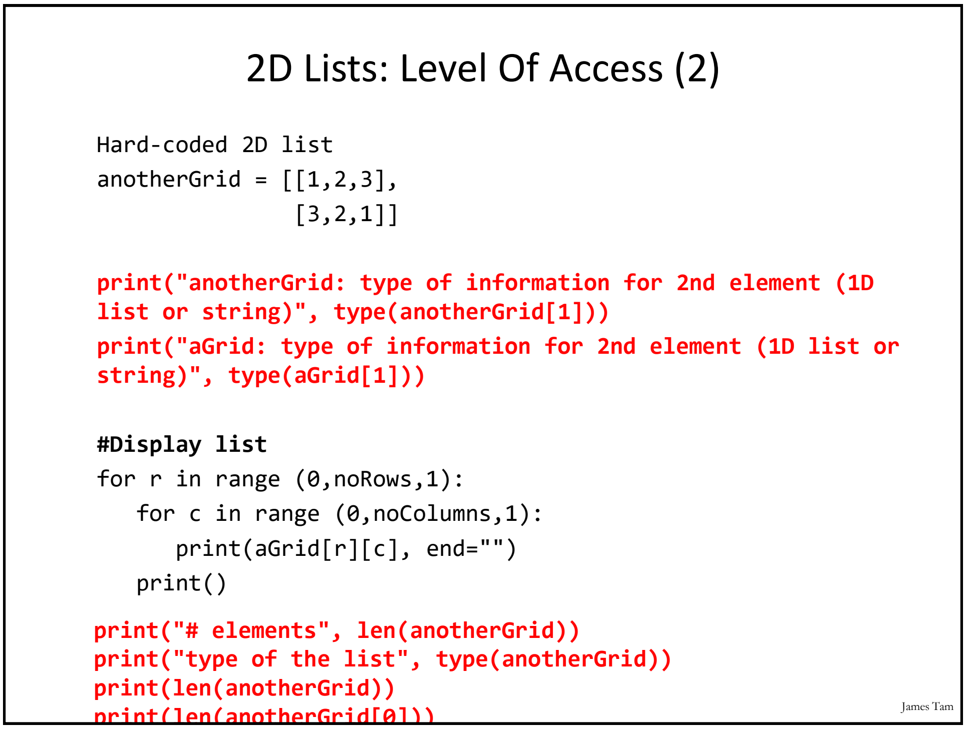
## 2D Lists: Level Of Access

- You need to **know what you are accessing**: reference, whole list, row, element (at a row/column).
- The example illustrates this issue via the append method but the append must be used on the right type of object.
- **Name of the example program:** `6misapplyingAppend.py`

```
aGrid = []
noRows = int(input("Number rows: "))
noColumns = int(input("Number columns: "))
#Create list
for r in range (0,noRows,1):
    aGrid.append ([])
    for c in range (0,noColumns,1):
        aGrid.append("*")
    #print(aGrid)
#print("# elements", len(aGrid))
#print("type of the list", type(aGrid))
```

```
#print(len(aGrid))
#print(len(aGrid[0]))
```

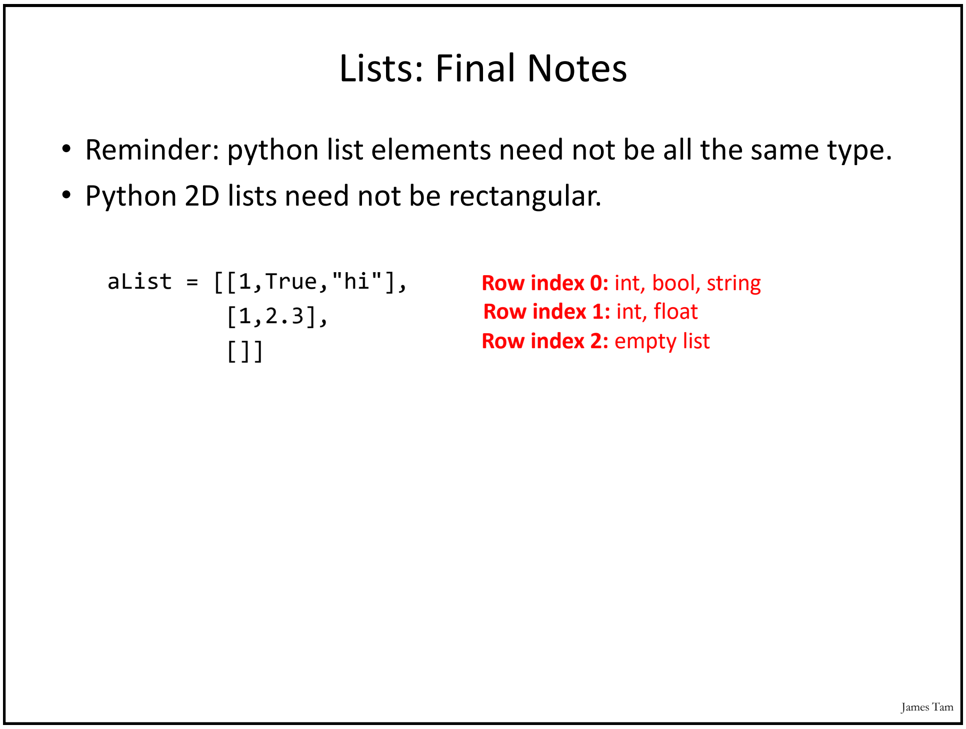
James Tam



## 2D Lists: Level Of Access (2)

```
Hard-coded 2D list
anotherGrid = [[1,2,3],
               [3,2,1]]

print("anotherGrid: type of information for 2nd element (1D
list or string)", type(anotherGrid[1]))
print("aGrid: type of information for 2nd element (1D list or
string)", type(aGrid[1]))

#Display list
for r in range (0,noRows,1):
   for c in range (0,noColumns,1):
      print(aGrid[r][c], end="")
   print()
print("# elements", len(anotherGrid))
print("type of the list", type(anotherGrid))
print(len(anotherGrid))
print(len(anotherGrid[0]))
```

James Tam



## Lists: Final Notes

- Reminder: python list elements need not be all the same type.
- Python 2D lists need not be rectangular.

```
aList = [[1,True,"hi"],
         [1,2.3],
         []]
```

**Row index 0:** int, bool, string
**Row index 1:** int, float
**Row index 2:** empty list

James Tam

# Extra Practice

List operations:

- For a numerical list: implement some common mathematical functions (e.g., average, min, max, mode – last one is challenging).
- For any type of list: implement common list operations (e.g., displaying all elements one at a time, inserting elements at the end of the list, insert elements in order, searching for elements, removing an element, finding the smallest and largest element).
  - In order to develop your programming skills you should write the code yourself rather than using predefined python methods such as append, `min`, `max` etc.

# After This Sub-Section You Should Now Know

- When to use lists of different dimensions
- Basic operations on a 2D list
- How to create a 2D list: fixed size and a variable sized list by using the repetition operator.
- How to access a 2D list: the whole list, rows in the list and individual elements.
- How to properly copy the contents of a 2D list into another 2D list as well as a common mistake when copying lists.
- The use of a named constant to ensure that list boundaries are adhered to.
- The ability to dynamically creating 2D lists using the append function for both the rows and columns.