Lecture 13: List Comprehensions, 2D Lists & Grids

Comp 102

Forman Christian University

List Comprehensions

- Applying a **function to every element of a sequence**, then creating a new list is a very common pattern

```python
def squares(L):
    '''Given a list L, return new list with squares of each
        number'''
    Lnew = []
    for e in L:
        Lnew.append(e**2)
    return Lnew

print(squares([1,2,3,4,5]))  # [1, 4, 9, 16, 25]
```

List Comprehensions

- Python provides a concise one-liner: a **list comprehension**
  - **Creates** a new list
  - **Applies** a function to every element of an iterable
  - *Optional*: only apply to elements that **Satisfy** a test

```
[function(e) for e in iterable if test]
```

List Comprehensions A **single expression** to create lists:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    Lnew.append(e**2)
```

```
L = [1,2,3,4]
Lnew = [ e**2 for e in L ]
```

List Comprehensions A **single expression** to create lists:

```python
L = [1,2,3,4]
Lnew = []
for e in L:
    Lnew.append(e**2)
```

New List

```python
L = [1,2,3,4]
Lnew = [ e**2 for e in L ]
```

List Comprehensions A single expression to create lists:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    Lnew.append(e**2)
```

Look at each elem

```
L = [1,2,3,4]
Lnew = [ e**2 for e in L ]
```

List Comprehensions A single expression to create lists:

```python
L = [1,2,3,4]
Lnew = []
for e in L:
    Lnew.append(e**2)
```

```python
L = [1,2,3,4]
Lnew = [ e**2 for e in L ]
```

Function to apply to each elem

List Comprehensions A **single expression** with a conditional test:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    if e%2 == 0:
        Lnew.append(e**2)

-----------------------------------------

Lnew = [e**2 for e in L if e%2==0]
```

List Comprehensions A **single expression** with a conditional test:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    if e%2 == 0:
        Lnew.append(e**2)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Lnew = [e**2 for e in L if e%2==0]
```

New List

List Comprehensions A single expression with a conditional test:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    if e%2 == 0:
        Lnew.append(e**2)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Lnew = [e**2 for e in L if e%2==0]
```

Look at each elem

List Comprehensions A single expression with a conditional test:

```
L = [1,2,3,4]
Lnew = []
for e in L:
    if e%2 == 0:
        Lnew.append(e**2)

------------------------------------------

Lnew = [e**2 for e in L if e%2==0]
```

Apply expr only if condition pass

List Comprehension Examples

```
L = [e**2 for e in range(4)]
print(L)     # [0, 1, 4, 9]

L = [e**2 for e in range(8) if e%2 == 0]
print(L)     # [0, 4, 16, 36]

L = [[e, e**2] for e in range(4) if e%2 != 0]
print(L)     # [[1,1], [3,9]]
```

You Try! What is the value returned by this expression?

- *Step 1:* what are **all values** in the sequence?
- *Step 2:* which **subset** does the condition keep?
- *Step 3:* **apply the function** to those values

```
L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
print(L)
```

You Try! What is the value returned by this expression?

- *Step 1:* what are **all values** in the sequence?
  `[‘xy’, ‘abcd’, 7, ‘4.0’]`
- *Step 2:* which **subset** does the condition keep?
- *Step 3:* **apply the function** to those values

```python
L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
print(L)
```

You Try! What is the value returned by this expression?

- *Step 1:* what are **all values** in the sequence?
  `['xy', 'abcd', 7, '4.0']`
- *Step 2:* which **subset** does the condition keep?
  `['xy', 'abcd', '4.0']`
- *Step 3:* **apply the function** to those values

```
L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
print(L)
```

You Try! What is the value returned by this expression?

- *Step 1:* what are **all values** in the sequence?
  `[‘xy’, ‘abcd’, 7, ‘4.0’]`
- *Step 2:* which **subset** does the condition keep?
  `[‘xy’, ‘abcd’, ‘4.0’]`
- *Step 3:* **apply the function** to those values
  `[2, 4, 3]`

```
L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
print(L)
```

You Try! What is the value returned by this expression?

- *Step 1:* what are **all values** in the sequence?
  `[‘xy’, ‘abcd’, 7, ‘4.0’]`
- *Step 2:* which **subset** does the condition keep?
  `[‘xy’, ‘abcd’, ‘4.0’]`
- *Step 3:* **apply the function** to those values
  `[2, 4, 3]`

```
L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
print(L)     -> [2, 4, 3]
```

2D Lists / Grids

What is a 2D List?

A **list of lists** — each element is itself a list.

Real-world grids:

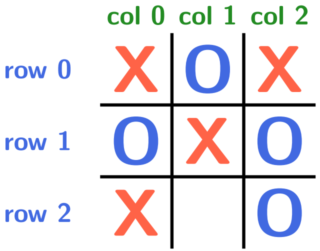
- **Tic-tac-toe** / **Battleship** board
- **Spreadsheet** cells (Excel)
- **Image** pixels
- **Game maps**

What is a 2D List?

A **list of lists** — each element is itself a list.

Real-world grids:

- **Tic-tac-toe** / **Battleship** board
- **Spreadsheet** cells (Excel)
- **Image** pixels
- **Game maps**



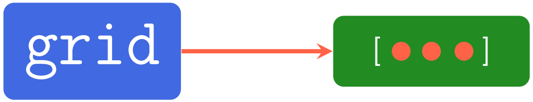
```
board = [['X', 'O', 'X'],
         ['O', 'X', 'O'],
         ['X', ' ', 'O']]
```

Creating a 2D List

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

Creating a 2D List

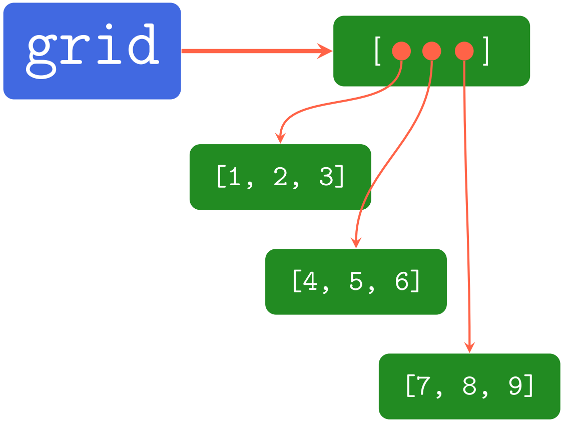
```
grid = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]
```



- grid points to an **outer list** with 3 slots

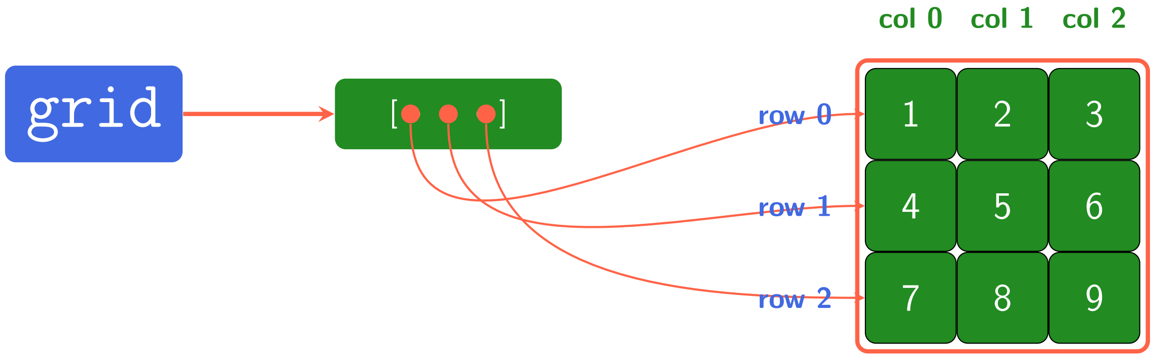
Creating a 2D List

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



- `grid` points to an **outer list** with 3 slots
- Each slot is a **pointer** to an inner list (a **row**)

2D List as a Grid



Indexing a 2D List

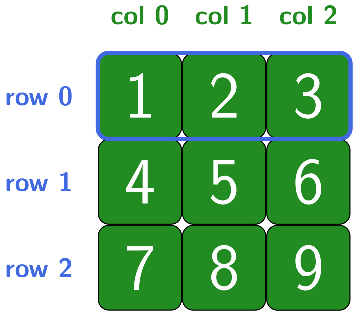
`grid[row][col]`

$$\begin{array}{c|ccc}
 & \text{col } 0 & \text{col } 1 & \text{col } 2 \\
\hline
\text{row } 0 & 1 & 2 & 3 \\
\text{row } 1 & 4 & 5 & 6 \\
\text{row } 2 & 7 & 8 & 9
\end{array}$$

Indexing a 2D List

`grid[row][col]`

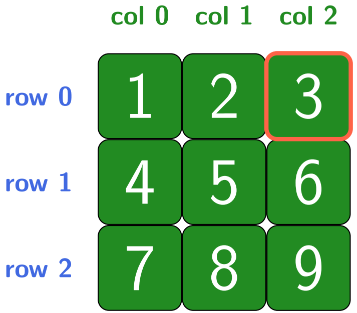
- `grid[0]` → `[1,2,3]` *(whole row)*



Indexing a 2D List

`grid[row][col]`

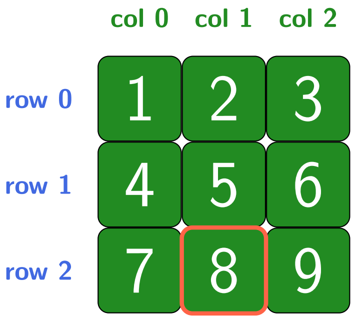
- `grid[0]` → `[1,2,3]` *(whole row)*
- `grid[0][2]` → 3



Indexing a 2D List

`grid[row][col]`

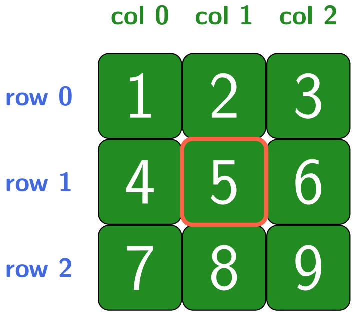
- `grid[0]` → `[1,2,3]` *(whole row)*
- `grid[0][2]` → 3
- `grid[2][1]` → 8



Indexing a 2D List

`grid[row][col]`

- `grid[0]` → `[1,2,3]` *(whole row)*
- `grid[0][2]` → 3
- `grid[2][1]` → 8
- `grid[1][1]` → 5



Two-Step Indexing

`grid[1][2]` is really **two steps**:

```
grid[1][2]
```

Two-Step Indexing

`grid[1][2]` is really **two steps**:

$\boxed{\texttt{grid[1]}}\texttt{[2]}$

Get row 1

- **Step 1:** `grid[1]` → `[4, 5, 6]` *(select row 1)*

Two-Step Indexing

`grid[1][2]` is really **two steps**:

$$\texttt{grid[1]}\boxed{\texttt{[2]}} \xleftarrow{\text{Get elem at idx 2}}$$

- **Step 1:** `grid[1]` → `[4, 5, 6]` *(select row 1)*
- **Step 2:** `[4,5,6][2]` → `6` *(select index 2)*

You Try!

```
M = [[10, 20, 30],
     [40, 50, 60],
     [70, 80, 90]]
```

- What is `M[2][0]`?
- What is `M[0]`?
- What is `M[1][-1]`?
- How do you get 50?

You Try!

```
M = [[10, 20, 30],
     [40, 50, 60],
     [70, 80, 90]]
```

- What is `M[2][0]`? $\rightarrow$ **70**
- What is `M[0]`?
- What is `M[1][-1]`?
- How do you get 50?

You Try!

```
M = [[10, 20, 30],
     [40, 50, 60],
     [70, 80, 90]]
```

- What is `M[2][0]`? $\rightarrow$ **70**
- What is `M[0]`? $\rightarrow$ **[10, 20, 30]**
- What is `M[1][-1]`?
- How do you get 50?

You Try!

```
M = [[10, 20, 30],
     [40, 50, 60],
     [70, 80, 90]]
```

- What is `M[2][0]`? $\rightarrow$ **70**
- What is `M[0]`? $\rightarrow$ **[10, 20, 30]**
- What is `M[1][-1]`? $\rightarrow$ **60**
- How do you get 50?

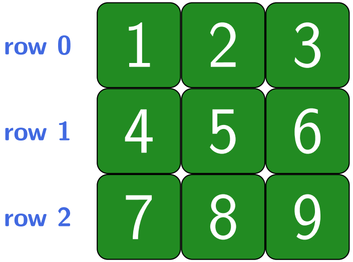
You Try!

```
M = [[10, 20, 30],
     [40, 50, 60],
     [70, 80, 90]]
```

- What is `M[2][0]`? → **70**
- What is `M[0]`? → **[10, 20, 30]**
- What is `M[1][-1]`? → **60**
- How do you get 50? → **M[1][1]**

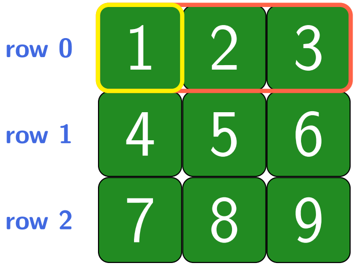
Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```



Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

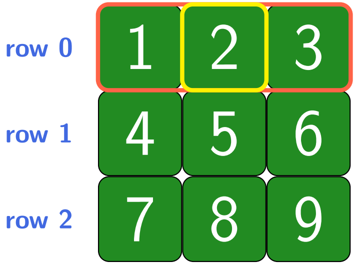


Output:

```
1
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

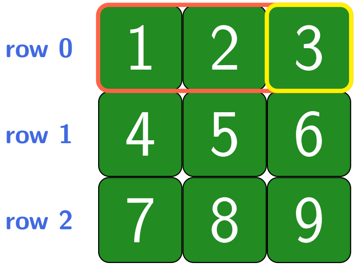


Output:

```
1 2
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

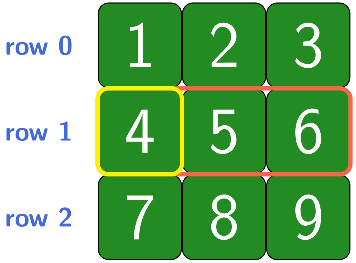


Output:

```
1 2 3
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

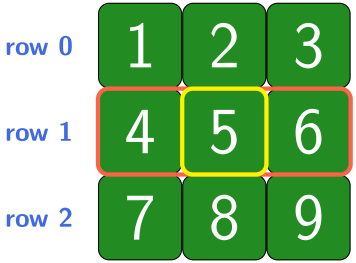


Output:

```
1 2 3
4
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

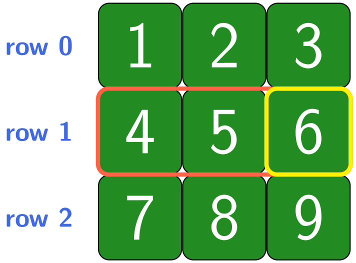


Output:

```
1 2 3
4 5
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

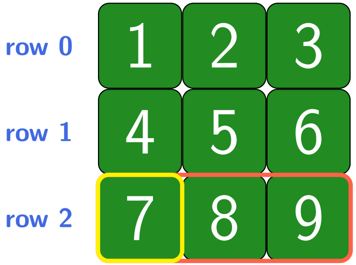


Output:

```
1 2 3
4 5 6
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```



Output:

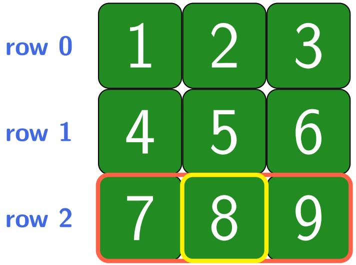
```
1 2 3
4 5 6
7
```

Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

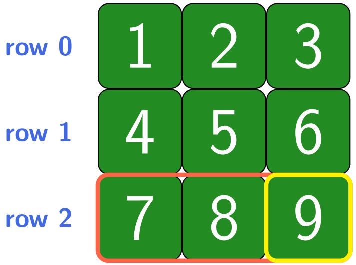
Output:

```
1 2 3
4 5 6
7 8
```



Iterating: Nested For Loops

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

for row in grid:
    for elem in row:
        print(elem, end=' ')
    print()
```

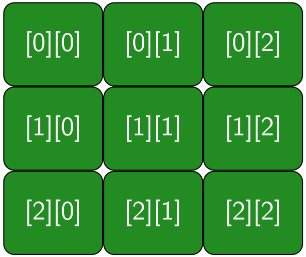


Output:

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

Iterating with Indices

```python
for r in range(len(grid)):
  for c in range(len(grid[r])):
    print(f'grid[{r}][{c}]', end=' ')
  print()
```

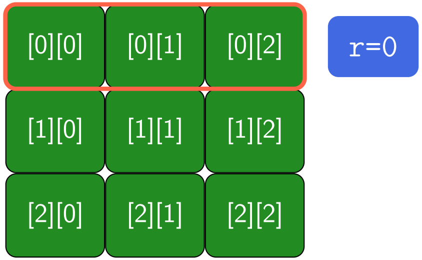


Iterating with Indices

```python
for r in range(len(grid)):
  for c in range(len(grid[r])):
    print(f'grid[{r}][{c}]', end=' ')
  print()
```

Output:

```
grid[0][0] grid[0][1] grid[0][2]
```

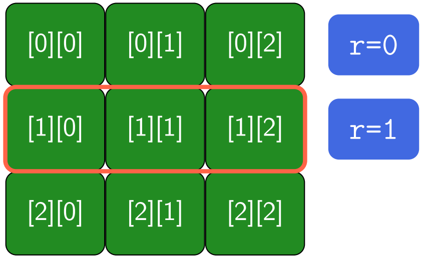


Iterating with Indices

```python
for r in range(len(grid)):
  for c in range(len(grid[r])):
    print(f'grid[{r}][{c}]', end=' ')
  print()
```

Output:

```
grid[0][0] grid[0][1] grid[0][2]
grid[1][0] grid[1][1] grid[1][2]
```

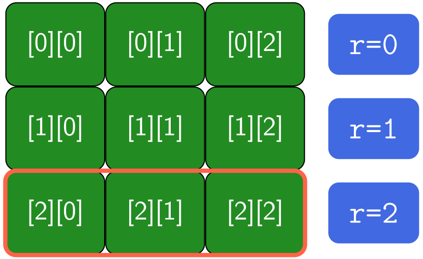


Iterating with Indices

```python
for r in range(len(grid)):
  for c in range(len(grid[r])):
    print(f'grid[{r}][{c}]', end=' ')
  print()
```

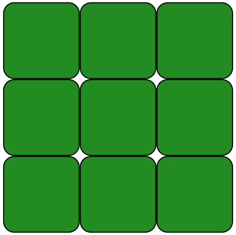
Output:

```
grid[0][0] grid[0][1] grid[0][2]
grid[1][0] grid[1][1] grid[1][2]
grid[2][0] grid[2][1] grid[2][2]
```



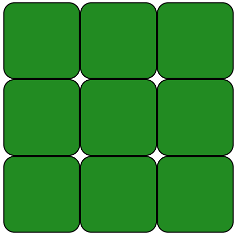
Row vs Column Traversal

Row-major



```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```

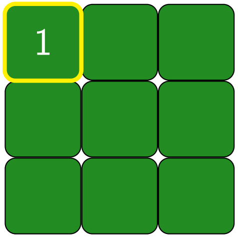
Column-major



```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

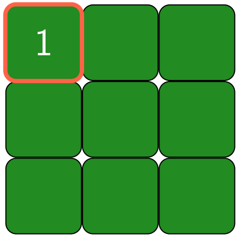
Row vs Column Traversal

Row-major



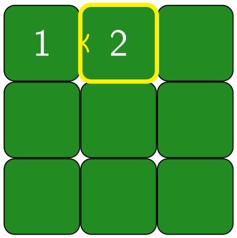
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```

Column-major



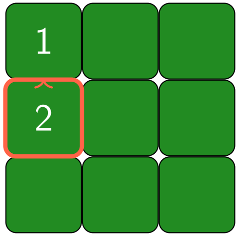
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```

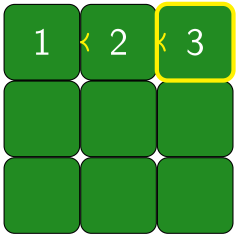


Column-major

```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

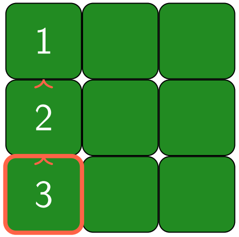
Row vs Column Traversal

Row-major



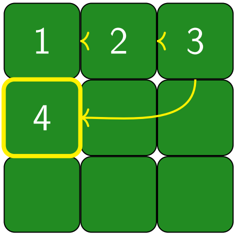
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```

Column-major



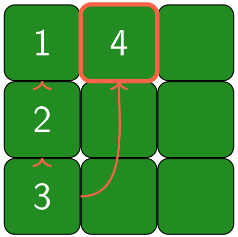
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

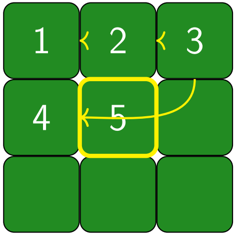
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```



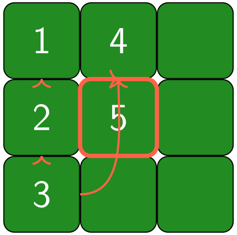
Column-major

```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

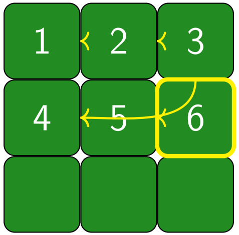


Column-major

```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```

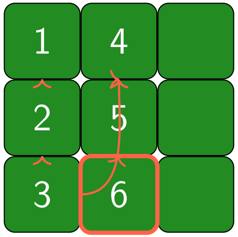
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

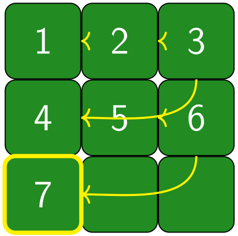
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```



Column-major

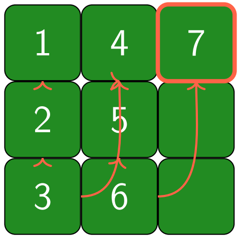
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-majør

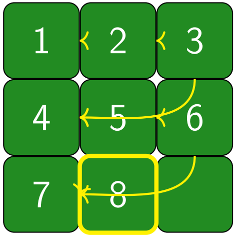
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```



Column-major

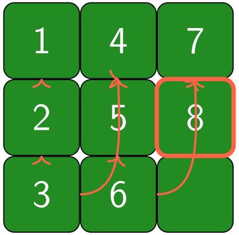
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

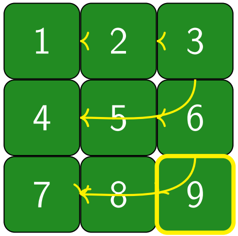
```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```



Column-major

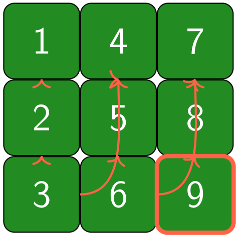
```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Row vs Column Traversal



Row-major

```
for r in range(3):
    for c in range(3):
        visit(grid[r][c])
```



Column-major

```
for c in range(3):
    for r in range(3):
        visit(grid[r][c])
```

Sum All Elements

Using nested loops:

```
def sum_grid(grid):
    total = 0
    for row in grid:
        for elem in row:
            total += elem
    return total
```

Sum All Elements

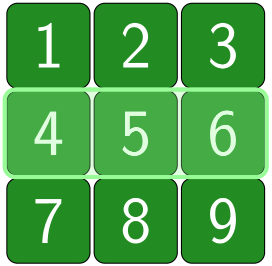
Using nested loops:

```
def sum_grid(grid):
    total = 0
    for row in grid:
        for elem in row:
            total += elem
    return total
```

One-liner using `sum()` on each row:

```
total = sum([sum(row) for row in grid])
```

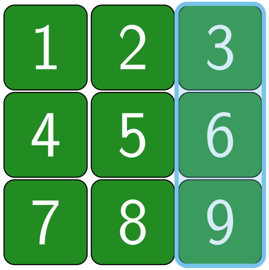
Sum a Row / Sum a Column



Sum of row `r`:

```
sum(grid[r])
# sum(grid[1]) -> 15
```

Sum a Row / Sum a Column



Sum of row `r`:

```
sum(grid[r])
# sum(grid[1]) -> 15
```

Sum of column `c`:

```
c = 2
sum([grid[r][c]
     for r in range(len(grid))])
# 3+6+9 = 18
```

Creating 2D Lists

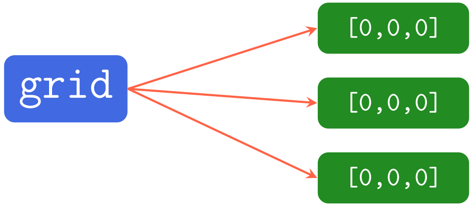
Correct — list comprehension:

```python
grid = [[0] * 3 for _ in range(3)]
```

Creating 2D Lists

Correct — list comprehension:

```
grid = [[0] * 3 for _ in range(3)]
```



3 separate lists

The Aliasing Trap

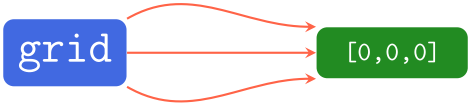
Wrong — do **not** do this!

```
grid = [[0] * 3] * 3
```

The Aliasing Trap

Wrong — do **not** do this!

```
grid = [[0] * 3] * 3
```

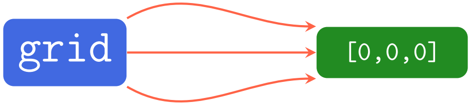


ALL SAME list!

The Aliasing Trap

Wrong — do **not** do this!

```
grid = [[0] * 3] * 3
```



ALL SAME list!

```
grid[0][0] = 5    # changes ALL rows!
# grid -> [[5,0,0], [5,0,0], [5,0,0]]
```

2D List Comprehensions

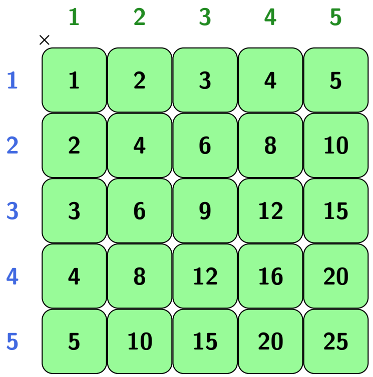
Creating grids with comprehensions:

```
# 3x3 grid of zeros
grid = [[0 for c in range(3)]
        for r in range(3)]

# Multiplication table
table = [[r*c for c in range(1,6)]
         for r in range(1,6)]
```

2D List Comprehensions

Creating grids with comprehensions:

```
# 3x3 grid of zeros
grid = [[0 for c in range(3)]
        for r in range(3)]

# Multiplication table
table = [[r*c for c in range(1,6)]
         for r in range(1,6)]
```



×	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

You Try!

Write a function that checks if a player has won **tic-tac-toe**:

```
board = [['X', 'O', 'X'],
         ['O', 'X', 'O'],
         ['X', ' ', 'X']]

def check_winner(board, player):
    '''Return True if player has 3 in a
    row, column, or diagonal'''
    pass
```

You Try!

Write a function that checks if a player has won **tic-tac-toe**:

```python
board = [['X', 'O', 'X'],
         ['O', 'X', 'O'],
         ['X', ' ', 'X']]

def check_winner(board, player):
    '''Return True if player has 3 in a
    row, column, or diagonal'''
    pass
```

Hints:

- Check each **row**: `all(board[r][c] == player for c in range(3))`
- Check each **column**: loop over c, check all rows
- Check **diagonals**: `board[0][0], board[1][1], board[2][2]`

You Try!

Write a function to **find a value** in a 2D grid:

```python
def find_in_grid(grid, target):
    '''Returns (row, col) of first occurrence
    of target. Returns None if not found.'''
    pass
```

You Try!

Write a function to **find a value** in a 2D grid:

```python
def find_in_grid(grid, target):
    '''Returns (row, col) of first occurrence
    of target. Returns None if not found.'''
    pass
```

Solution:

```python
def find_in_grid(grid, target):
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == target:
                return (r, c)
    return None
```

Questions?