Lecture 13: List Comprehensions, 2D Lists & Grids

Comp 102

Fakhir Shaheen

Forman Christian University

List Comprehensions

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

```
1 def squares(L):
2     '''Given a list L, return new list with squares of each
          number'''
3     Lnew = []
4     for e in L:
5         Lnew.append(e**2)
6     return Lnew
7
8 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 ]
```

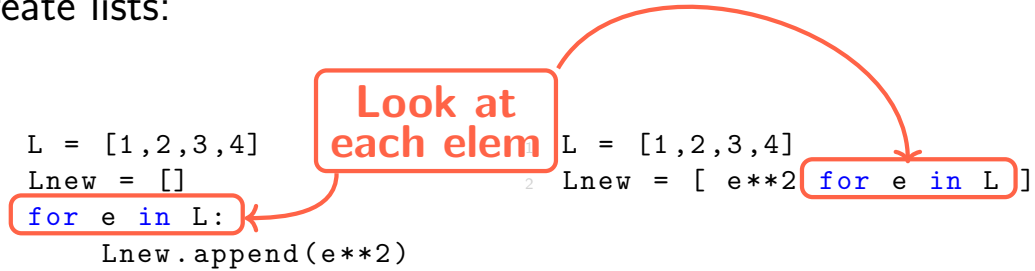
List Comprehensions A single expression to create lists:

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

New List

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

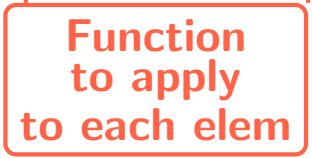
List Comprehensions A single expression to create lists:



List Comprehensions A single expression to create lists:

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

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



List Comprehensions A single expression with a conditional test:

```
1  L = [1,2,3,4]
2  Lnew = []
3  for e in L:
4      if e%2 == 0:
5          Lnew.append(e**2)
6
-----------------------------------------
7
8  Lnew = [e**2 for e in L if e%2==0]
```

List Comprehensions A single expression with a conditional test:

```
1  L = [1,2,3,4]
2  Lnew = []
3  for e in L:
4      if e%2 == 0:
5          Lnew.append(e**2)
6
-------------------------------------------
7
8  Lnew = [e**2 for e in L if e%2==0]
```

New List

List Comprehensions A single expression with a conditional test:

```
1  L = [1,2,3,4]
2  Lnew = []
3  for e in L:
4      if e%2 == 0:
5          Lnew.append(e**2)
6
-------------------------------------------
7
8  Lnew = [e**2 for e in L if e%2==0]
```

Look at each elem

List Comprehensions A single expression with a conditional test:

```
1 L = [1,2,3,4]
2 Lnew = []
3 for e in L:
4     if e%2 == 0:
5         Lnew.append(e**2)
6
7
8 Lnew = [e**2 for e in L if e%2==0]
```



List Comprehension Examples

```
1 L = [e**2 for e in range(4)]
2 print(L)     # [0, 1, 4, 9]
3
4 L = [e**2 for e in range(8) if e%2 == 0]
5 print(L)     # [0, 4, 16, 36]
6
7 L = [[e, e**2] for e in range(4) if e%2 != 0]
8 print(L)     # [[1,1], [3,9]]
```

Bug Hunt! *(write the line number and the fix, 30 s)*

Keep only the passing marks (50 or more):

```
1 marks = [45, 72, 30, 50]
2 passed = [m if m >= 50 for m in marks]
3 print(passed)
```

Symptom:

```
SyntaxError: expected 'elsé after 'if' expression
```

Bug Hunt! *(write the line number and the fix, 30 s)*

Keep only the passing marks (50 or more):

```
1 marks = [45, 72, 30, 50]
2 passed = [m if m >= 50 for m in marks]
3 print(passed)
```

Symptom:

```
SyntaxError: expected 'elsé after 'if' expression
```

Line 2. A comprehension has **two** `if`s, and they differ:

```
passed = [m for m in marks if m >= 50]                # [72, 50]
grades = ['P' if m >= 50 else 'F' for m in marks]   # F P F P
```

An `if` **after** the `for` drops elements. An `if ... else` **before** it picks a value and needs its `else`. Corner case: 50 stays, the test is `>=`.

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

```
1 L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
2 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

```
1 L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
2 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]`

```
1 L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
2 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]`

```
1 L = [len(x) for x in ['xy','abcd',7,'4.0'] if type(x) == str]
2 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:

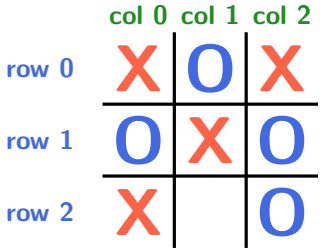
- **Tic-tac-toe** / **Battleship** board
- **Spreadsheet** cells (Excel)
- **Image** pixels
- **Game maps**

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

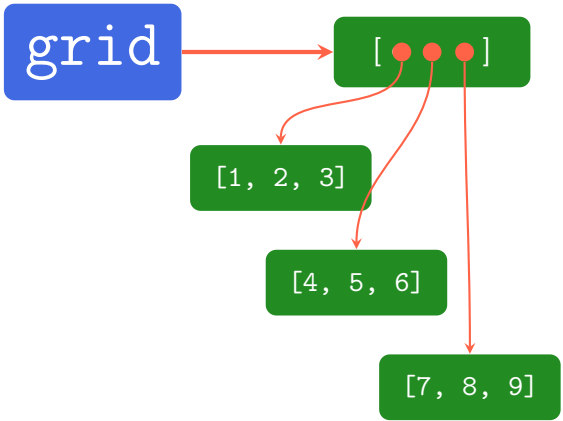
```
grid = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]
```



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

Creating a 2D List

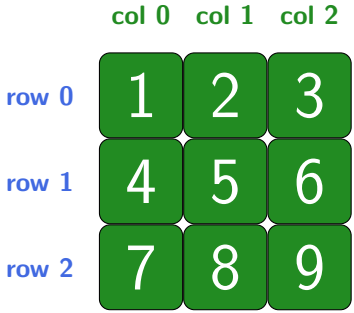
```
grid = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]
```



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

Indexing a 2D List

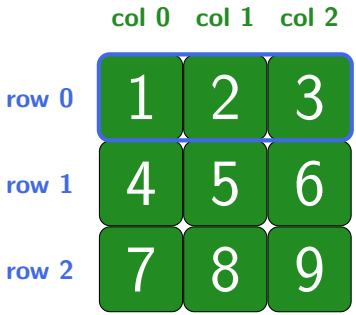
`grid[row][col]`



Indexing a 2D List

`grid[row][col]`

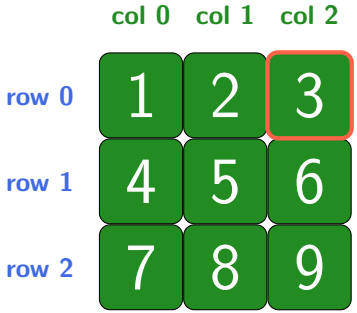
- `grid[0]` → `[1,2,3]` *(whole row)*



Indexing a 2D List

`grid[row][col]`

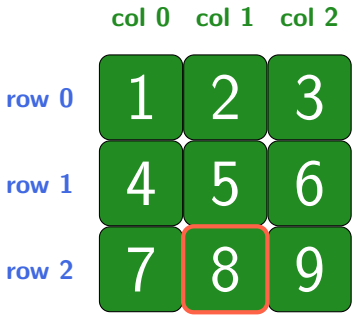
- `grid[0]` → `[1,2,3]` *(whole row)*
- `grid[0][2]` → `3`



Indexing a 2D List

`grid[row][col]`

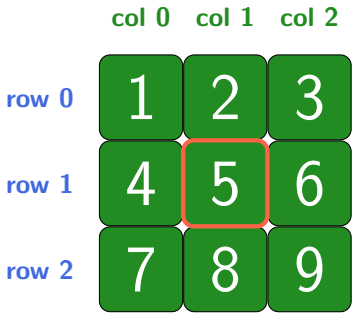
- `grid[0]` → `[1,2,3]` *(whole row)*
- `grid[0][2]` → `3`
- `grid[2][1]` → `8`



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`



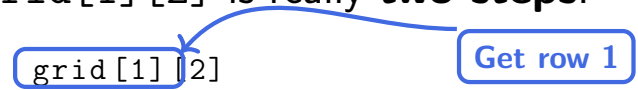
Two-Step Indexing

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

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

Two-Step Indexing

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



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

Two-Step Indexing

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



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

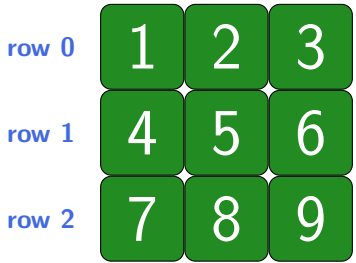
You Try!

```
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]**

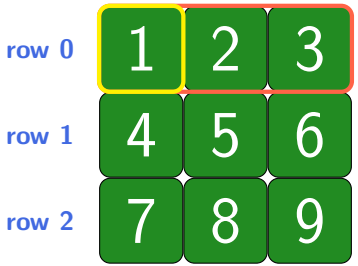
Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```



Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

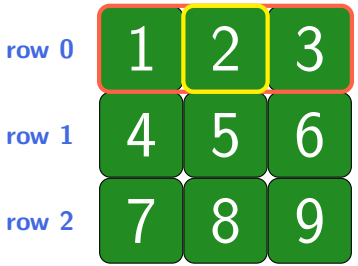


Output:

```
1
```

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

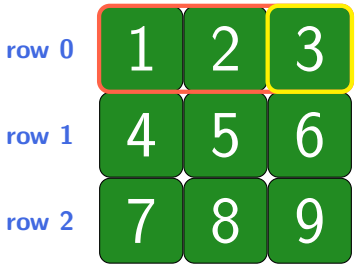


Output:

```
1 2
```

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

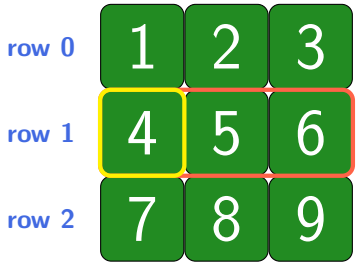


Output:

```
1 2 3
```

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

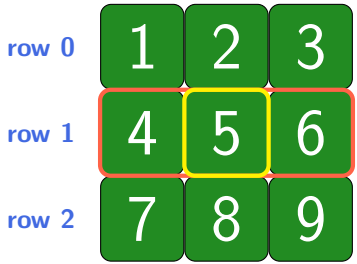


Output:

```
1 2 3
4
```

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

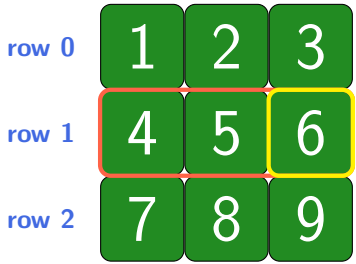


Output:

```
1 2 3
4 5
```

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

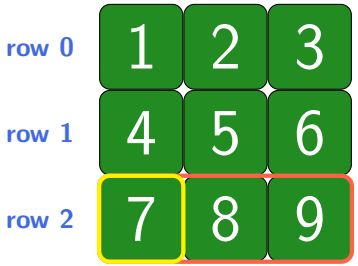


Output:

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

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

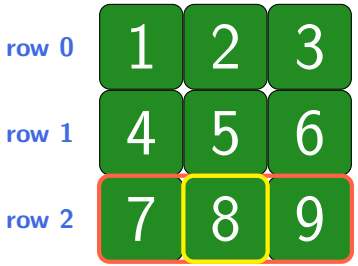


Output:

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

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

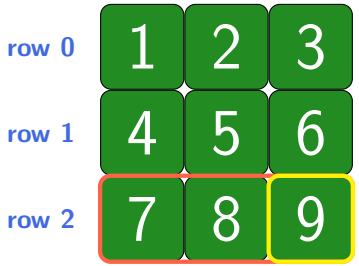


Output:

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

Iterating: Nested For Loops

```
1  grid = [[1, 2, 3],
2          [4, 5, 6],
3          [7, 8, 9]]
4
5  for row in grid:
6      for elem in row:
7          print(elem, end=' ')
8      print()
```

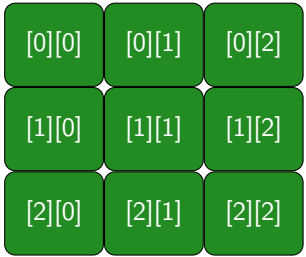


Output:

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

Iterating with Indices

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

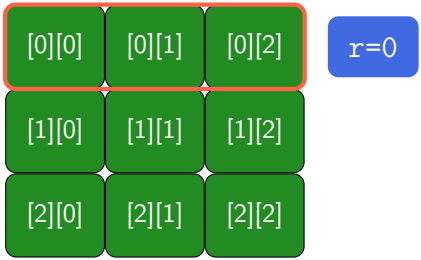


Iterating with Indices

```
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]
```

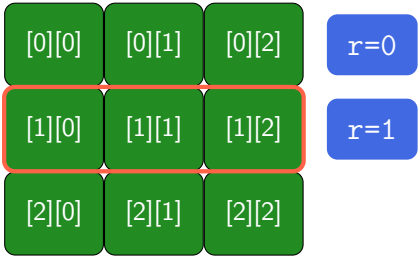


Iterating with Indices

```
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]
```

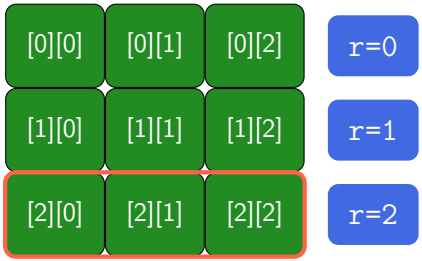


Iterating with Indices

```
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]
```



Bug Hunt! *(write the line number and the fix, 30 s)*

Print every number in the grid, one per line:

```
1 grid = [[1, 2, 3, 4],
2         [5, 6, 7, 8]]
3 for r in range(len(grid)):
4     for c in range(len(grid)):
5         print(grid[r][c])
```

Symptom: half the grid

```
1
2
5
6
```

Four numbers of eight.

Bug Hunt! *(write the line number and the fix, 30 s)*

Print every number in the grid, one per line:

```
1 grid = [[1, 2, 3, 4],
2         [5, 6, 7, 8]]
3 for r in range(len(grid)):
4     for c in range(len(grid)):
5         print(grid[r][c])
```

Symptom: half the grid

```
1
2
5
6
```

Four numbers of eight.

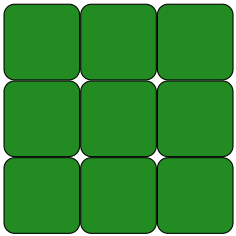
Line 4. `len(grid)` is the number of **rows** (2), not columns (4):

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

Corner case: on a **square** grid the bug is invisible. Test grids that are wider than tall.

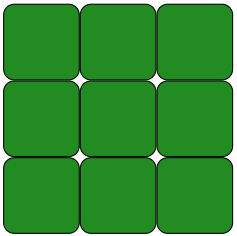
Row vs Column Traversal

Row-major



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

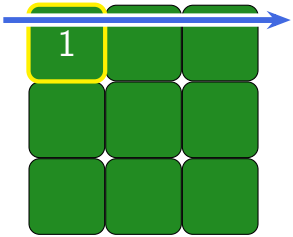
Column-major



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

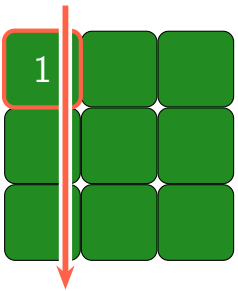
Row vs Column Traversal

Row-major




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

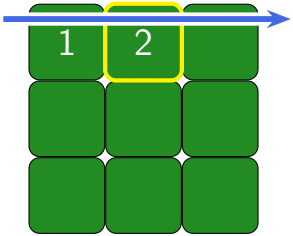
Column-major




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

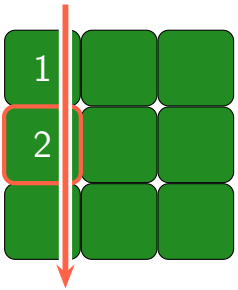
Row vs Column Traversal

Row-major




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

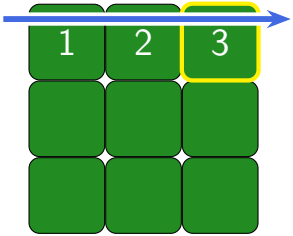
Column-major




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

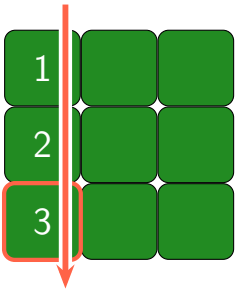
Row vs Column Traversal

Row-major




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

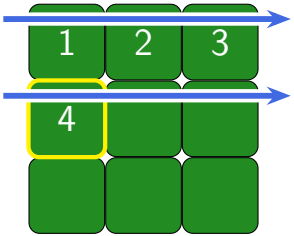
Column-major




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

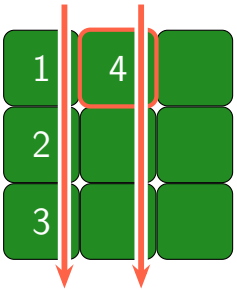
Row vs Column Traversal

Row-major




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

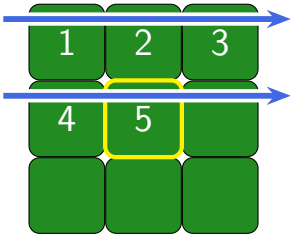
Column-major




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

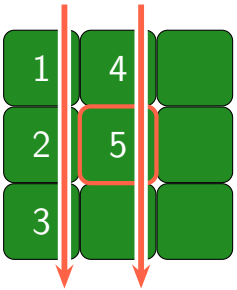
Row vs Column Traversal

Row-major




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

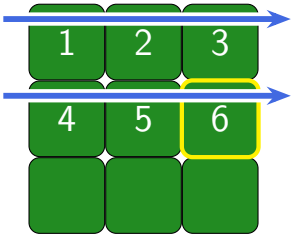
Column-major




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

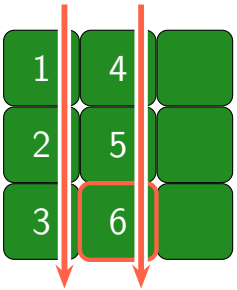
Row vs Column Traversal

Row-major




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

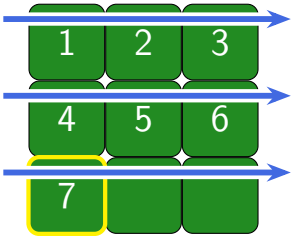
Column-major




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

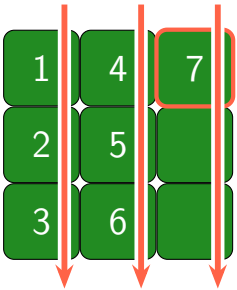
Row vs Column Traversal

Row-major




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

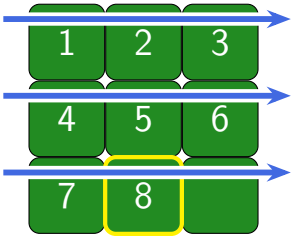
Column-major




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

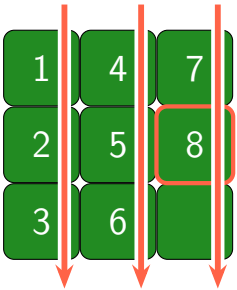
Row vs Column Traversal

Row-major




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

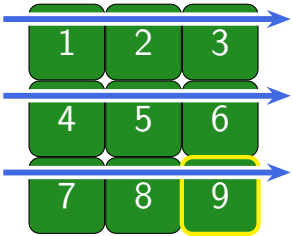
Column-major




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

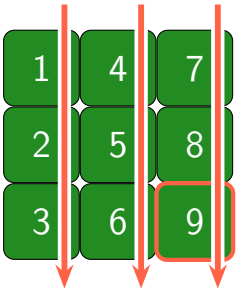
Row vs Column Traversal

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])
```

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
```

Bug Hunt! *(write the line number and the fix, 30 s)*

Add up column c:

```
1 grid = [[1, 2, 3],
2         [4, 5, 6]]
3 def col_sum(grid, c):
4     total = 0
5     for r in range(len(grid)):
6         total += grid[c][r]
7     return total
8 print(col_sum(grid, 0))
9 print(col_sum(grid, 2))
```

Symptom:

```
3
IndexError: list index
out of range
```

Column 0 should be `1 + 4 = 5`.

Bug Hunt! *(write the line number and the fix, 30 s)*

Add up column c:

```
1  grid = [[1, 2, 3],
2          [4, 5, 6]]
3  def col_sum(grid, c):
4      total = 0
5      for r in range(len(grid)):
6          total += grid[c][r]
7      return total
8  print(col_sum(grid, 0))
9  print(col_sum(grid, 2))
```

Symptom:

```
3
IndexError: list index
out of range
```

Column 0 should be `1 + 4 = 5`.

Line 6. Row first, then column: `grid[r][c]`. As written it summed **row** 0, and `grid[2]` does not exist.

Corner case: on a 3×3 grid nothing crashes; the wrong number comes back silently.

Creating 2D Lists

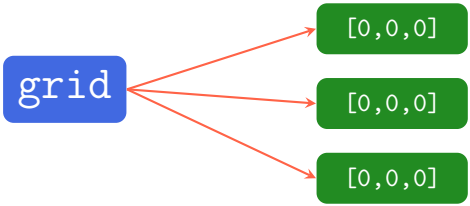
Correct — list comprehension:

```
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**:

```
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:

```
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:

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

Solution:

```
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
```

Bug Hunt! *(write the line number and the fix, 30 s)*

The same search, typed again by a friend:

```
1  def find(grid, target):
2      for r in range(len(grid)):
3          for c in range(len(grid[r])):
4              if grid[r][c] == target:
5                  return (r, c)
6          return None
7  g = [[1, 2], [3, 4]]
8  print(find(g, 2))
9  print(find(g, 3))
```

Symptom:

```
(0, 1)
None
```

3 is there, at `(1, 0)`.

Bug Hunt! *(write the line number and the fix, 30 s)*

The same search, typed again by a friend:

```
1  def find(grid, target):
2      for r in range(len(grid)):
3          for c in range(len(grid[r])):
4              if grid[r][c] == target:
5                  return (r, c)
6          return None
7  g = [[1, 2], [3, 4]]
8  print(find(g, 2))
9  print(find(g, 3))
```

Symptom:

```
(0, 1)
None
```

3 is there, at `(1, 0)`.

Line 6. `return None` is inside the **outer** loop: it gives up after row 0. Unindent it to the level of `for r`.

Corner case: row 0 always works, so test the **last** row and a value that is absent.

Summary

- A 2D list is a **list of lists**; each inner list is a row.
- Access elements with `grid[row][col]` (two-step indexing).
- Use **nested for loops** to iterate over all elements.
- Row-major vs. column-major: swap the loop order.
- Create 2D lists with **list comprehensions**, not `[[0]*n]*m` (aliasing trap!).

Questions?

Exit Ticket

Today's question *(2 minutes)*

Write down the value of each expression:

(a) `[n * 2 for n in [1, 2, 3]]`

(b) let `g = [[1, 2], [3, 4]]`;
what is `g[1][0]`?

COMP 102 — Exit Ticket #13

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.