Finger Exercises 13

List Comprehensions & 2D Grids

≈ 60 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Most of the lists and grids here are new, so you have to *work them out*, not remember them. For a comprehension, ask: *which values come in, which are kept, and what is made of each one?* For a grid: *which row, and then which column?*

How to do it. On **paper**, **no computer**, in one sitting. If an item takes you more than a minute, skip it and come back. Parts A–K are for *everyone*; Part L is a **stretch**. The Console Quests are for tonight, *at* a computer. The answer key is on the last pages — do not peek until you are done.

Predict first, then check. Write down what you think *before* you look anything up or type anything in — even for the Console Quests. A guess that turns out wrong teaches you far more than an answer you only watched the computer produce.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. A list comprehension always does two things, and may do a third. It builds a brand-new _____, it applies an _____ to every element of the sequence, and *optionally* it keeps only the elements that pass a _____.

A2. A 2D list is a _____ of _____. To read the value in row 2, column 0 of a grid `G` you write `G[___][___]`.

A3. For a grid `G`, `len(G)` gives the number of _____ and `len(G[0])` gives the number of _____.

A4. A comprehension can hold two different `if`s. An `if` written **after** the `for` _____ the elements that fail it, so the new list can come out _____. An `if ... else` written **before** the `for` picks a _____ for *every* element, and it cannot be written without its _____.

A5. `G[1][2]` is worked out in two steps. First `G[1]` gives a whole _____, which is a _____. Then `[2]` takes one _____ out of it.

A6. Circle **True** or **False**:

(a) A list comprehension changes the list you loop over. True / False

(b) `G[0]` gives you a whole row, not one number. True / False

(c) `[[0] * 3] * 3` builds three separate rows. True / False

(d) Swapping the two loops turns row-major order into column-major order. True / False

(e) `G[1][2]` and `G[2][1]` are always the same cell. True / False

(f) A comprehension with a test can produce an empty list. True / False

(g) In `for row in G:` the name `row` holds one number at a time. True / False

(h) For a grid of 2 rows and 5 columns, `len(G)` is 5. True / False

Part B • What Does It Print?

TRACE ≈ 6 min

Write the exact list that `print(L)` shows. From B4 on, write every line of output.

B1.

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

Output: _____

B2. One test added at the end. What changes?

```
1 L = [e*2 for e in range(4) if e%2==1]
2 print(L)
```

Output: _____

B3.

```
1 L = [[e, e*e] for e in range(3)]
2 print(L)
```

Output: _____

B4. Lines 2 and 3 hold the two `if`s side by side.

```
1 nums = [3, -1, 4, -5]
2 A = [n for n in nums if n > 0]
3 B = [n if n > 0 else 0 for n in nums]
4 print(A)
5 print(B)
6 print(nums)
```

Output:

_____

_____

_____

B5. The sequence does not have to be a list of numbers.

```
1 words = ['row', 'grid', 'hi', 'list']
2 A = [len(w) for w in words]
3 B = [w[0] for w in words if len(w) > 3]
4 C = [ch for ch in 'banana' if ch != 'a']
5 print(A)
6 print(B)
7 print(C)
```

Output:

_____

_____

_____

B6.

```
1 L = [1, 2, 3]
2 K = L
3 L = [e + 1 for e in L]
4 K.append(9)
5 print(K, L)
```

Output:

_____

List objects when line 5 runs: _____

Part C • Values In, Values Kept, List Out

TRACE ≈ 5 min

C1. Read a comprehension in three steps, the way the "You Try" slide did: **1.** every value the sequence gives, **2.** the values the test keeps, **3.** the expression worked out for each value kept. Here `xs = [9, 4, 7, 10]` and `ws = ['list', 'of', 'rows']`. Row 1 is done for you.

Comprehension	1. Values in	2. Kept	3. New list
`[n*n for n in range(6) if n%2 == 1]`	`0 1 2 3 4 5`	`1 3 5`	`[1, 9, 25]`
`[x//2 for x in xs if x > 5]`	_____	_____	_____
`[w[-1] for w in ws if len(w) > 2]`	_____	_____	_____
`[10 - d for d in range(3, 0, -1)]`	_____	_____	_____
`[n for n in range(9, 14) if n%3 == 0]`	_____	_____	_____

C2. How long? Do not build the list. Write only the number of items it will have.

Comprehension	Items	Comprehension	Items
`[0 for e in range(7)]`	_____	`[e for e in range(7) if e > 4]`	_____
`[[e, e] for e in range(4)]`	_____	`[e if e > 4 else 0 for e in range(7)]`	_____
`[[0] * 5 for r in range(3)]`	_____	`[c for c in 'grid' if c in 'aeiou']`	_____
`[e*e for e in [5, 5, 5]]`	_____	`[e for e in range(7) if e > 9]`	_____

Part D • Loop to Comprehension, and Back

TRANSLATE ≈ 4 min

D1. Each loop builds a list. Write the **one line** that builds the same list.

(a)

```
big = []
for m in marks:
    if m >= 50:
        big.append(m)
```

_____

(b)

```
halves = []
for n in range(10):
    halves.append(n / 2)
```

_____

D2. Now the other way round. Fill in the loop that does the same job as the one line above it.

(a) `tens = [n * 10 for n in nums if n > 2]`

```
tens = _______
for _____ in ________:
    if _____________:
        tens._______(_______)
```

(b) `col = [row[1] for row in G]`

```
col = _______
for ______ in ______:
    _________________________
```

Part E • Reading a Grid

TRACE ≈ 4 min

```
G = [[3, 1, 4],
     [1, 5, 9],
     [2, 6, 5]]
```

E1. `G[0][2]` is _____ and `G[2][0]` is _____

E2. `G[1]` is _____

E3. `sum(G[1])` is _____ and `sum([G[r][0] for r in range(3)])` is _____

E4. `G[-1][-1]` is _____ `G[1][-1]` is _____ `G[-1]` is _____

E5. Now the other way round. Write an expression with two indices for each cell.

the 9: _____ the 4: _____ the 5 in the middle: _____ the other 5: _____

Part F • What Kind of Value?

CLASSIFY ≈ 3 min

Using the same grid `G` as in Part E, tick one column for each.

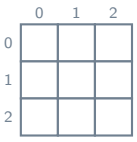
Expression	One number	A flat list	A list of lists
`G[1][1]`	☐	☐	☐
`G[1]`	☐	☐	☐
`sum(G[0])`	☐	☐	☐
`[G[r][0] for r in range(3)]`	☐	☐	☐
`[[0] * 3 for r in range(3)]`	☐	☐	☐
`G[0] + G[1]`	☐	☐	☐
`[sum(row) for row in G]`	☐	☐	☐
`G[1:]`	☐	☐	☐
`[[x * 2 for x in row] for row in G]`	☐	☐	☐

Part G • Which Cell Comes Next? TRACE ≈ 8 min

G1. Number the cells. Each loop visits cells of a grid with 3 rows and 3 columns. Write 1 in the cell it visits first, 2 in the next one, and so on. A cell that the loop never visits stays empty.

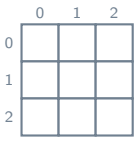
(a)

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



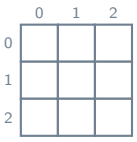
(b)

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



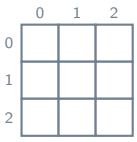
(c)

```
1 for r in range(2, -1, -1):
2     for c in range(3):
3         visit(grid[r][c])
```



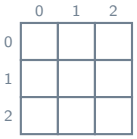
(d)

```
1 for r in range(3):
2     for c in range(r + 1):
3         visit(grid[r][c])
```



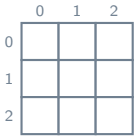
(e)

```
1 for i in range(3):
2     visit(grid[i][i])
```



(f)

```
1 for i in range(3):
2     visit(grid[i][2 - i])
```



G2. One program, run three ways: **(a)** as it stands, **(b)** with line 5 indented one level deeper, in line with line 4, **(c)** with line 5 deleted.

```
1 g = [[1, 2], [3, 4]]
2 for row in g:
3     for x in row:
4         print(x, end=' ')
5     print()
```

Run	Lines of output	The first line
(a)	_____	_____
(b)	_____	_____
(c)	_____	_____

G3. The search from the slides returns at the *first* match, so the order of the visit decides the answer. Line 4 is the comparison. Count how many times it runs.

```
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
8 g = [[4, 1, 8],
9      [8, 3, 6]]
```

Call	Returns	Line 4 ran
`find(g, 4)`	_____	_____
`find(g, 8)`	_____	_____
`find(g, 3)`	_____	_____
`find(g, 7)`	_____	_____

Lines 2 and 3 are rewritten to visit column by column: `for c in range(len(grid[0])):` and then `for r in range(len(grid)):`.

Now `find(g, 8)` returns _____ after line 4 has run _____ times.

Part H • Rows, Columns, Diagonals

TRACE ≈ 6 min

H1. A grid does not have to be square. In this marks table each row is one student and each column is one quiz. Write the value, or *error*.

```
1 T = [[7, 9, 4],
2      [8, 3, 10]]
```

`len(T)` _____ `len(T[0])` _____ `T[1][2]` _____

`T[2][1]` _____ `T[0][3]` _____ `T[-1][0]` _____

H2. The same table `T`: a row is one student, a column is one quiz. Write each value.

	0	1	2
0	7	9	4
1	8	3	10

Which row of the table on the right is quiz 2's total? _____

	Expression	Value
1	`sum(T[1])`	
2	`[sum(row) for row in T]`	
3	`sum([sum(row) for row in T])`	
4	`[row[1] for row in T]`	
5	`[T[r][0] for r in range(len(T))]`	
6	`sum([T[r][2] for r in range(len(T))])`	
7	`[max(row) for row in T]`	

H3. The board from the slides. Each expression reads three of its cells. **Ring** those three cells on the small board, and write the list.

```
1 B = [['X', 'O', 'X'],
2      ['O', 'X', 'O'],
3      ['X', ' ', 'O']]
```

(a)

	0	1	2
0	X	O	X
1	O	X	O
2	X		O

(b)

	0	1	2
0	X	O	X
1	O	X	O
2	X		O

(c)

	0	1	2
0	X	O	X
1	O	X	O
2	X		O

(d)

	0	1	2
0	X	O	X
1	O	X	O
2	X		O

Which of the four is a winning line, and whose?

_____

(a) `B[1]` _____

(b) `[B[r][2] for r in range(3)]` _____

(c) `[B[i][i] for i in range(3)]` _____

(d) `[B[i][2 - i] for i in range(3)]` _____

Part I • Build a Grid

DRAW ≈ 8 min

I1. All five grids come from one line, `grid = [[E for c in range(C)] for r in range(R)]`, with a different expression `E` and different numbers `C` and `R`. Fill in each grid. Every empty grid is drawn 3 rows by 4 columns: **cross out** the rows and columns that a grid does not have.

(a) `E` is `r * c`, `C` is 4, `R` is 3

(b) `E` is `r`, `C` is 3, `R` is 3

(c) `E` is `c`, `C` is 3, `R` is 2

(d) `E` is `r + c`, `C` is 2, `R` is 3

(e) `E` is `10*r + c`, `C` is 4, `R` is 2

I2. Now the grid is given. Each one was built by `[[E for c in range(3)] for r in range(3)]`. Write `E`.

	0	1	2
0	1	1	1
1	2	2	2
2	3	3	3

(a) `E`: _____

	0	1	2
0	1	2	3
1	2	4	6
2	3	6	9

(b) `E`: _____

	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1

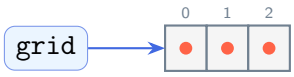
(c) `E`: _____

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

(d) `E`: _____

I3. The picture from the slides: the name `grid`, an outer list with one dot for each position, and the rows. Draw an arrow from each dot to the row it points at. **Cross out** any row box that the line never builds.

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



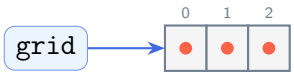
`[0, 0, 0]` `[0, 0, 0]` `[0, 0, 0]`

List objects: _____

After `grid[1][2] = 7`, `print(grid)`:

_____

(b) `grid = [[0] * 3] * 3`



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

List objects: _____

After `grid[1][2] = 7`, `print(grid)`:

_____

I4. One `5`, or three? Each row of the table is a program of its own. It builds `g` as shown, then `g[0][0] = 5` runs, then `print(g)`. How many 5s are printed?

	Line 1	Line 2	5s printed
1	`g = [[0] * 3 for r in range(3)]`		_____
2	`g = [[0] * 3] * 3`		_____
3	`row = [0] * 3`	`g = [row, row, row]`	_____
4	`row = [0] * 3`	`g = [row for r in range(3)]`	_____
5	`row = [0] * 3`	`g = [row[:] for r in range(3)]`	_____
6	`g = [[0] * 3] * 3`	`g[0] = [0, 0, 0]`	_____

Part J • Circle the Bug, Then Fix It

FIX ≈ 8 min

Each snippet has exactly one bug of a kind we met in class. From J3 on, say what actually happens (the wrong output, or the name of the error), then write the fix.

J1. We wanted a 3×3 grid of zeros and then to change only the top-left cell.

```
1 grid = [[0] * 3] * 3
2 grid[0][0] = 5
3 print(grid)
```

It actually prints:

_____

Corrected line 1:

_____

J2. We wanted the squares of the even numbers below 6.

```
1 L = [if e%2==0 e*e for e in range(6)]
```

Corrected line:

_____

J3. We wanted every mark below 50 replaced by `0`, and the list kept at its full length.

```
1 L = [m for m in marks if m >= 50 else 0]
```

Error: _____

Corrected line: _____

J4. Two loops over the marks table of Part H. The bug is in a different place in each.

(a) should print the three quiz totals, `15`, `12` and `14`

```
1 T = [[7, 9, 4], [8, 3, 10]]
2 for c in range(len(T)):
3     total = 0
4     for r in range(len(T)):
5         total += T[r][c]
6     print(total)
```

It prints: _____

Fix: _____

(b) should print each student's total, `20` and then `21`

```
1 T = [[7, 9, 4], [8, 3, 10]]
2 total = 0
3 for row in T:
4     for m in row:
5         total += m
6     print(total)
```

It prints: _____

Fix: _____

J5. Two more.

(a) `has_zero([[1, 0], [2, 3]])` should be `True`

```
1 def has_zero(grid):
2     for row in grid:
3         for x in row:
4             if x == 0:
5                 return True
6             else:
7                 return False
```

It returns: _____

Fix: _____

(b) should print `[[0, 0], [0, 0]]`

```
1 g = [[1, 2], [3, 4]]
2 for row in g:
3     for x in row:
4         x = 0
5 print(g)
```

It prints: _____

Rewrite lines 2 to 4:

_____

_____

_____

Part K • One Sentence Each

EXPLAIN ≈ 3 min

K1. Why is `[[0] * 3] * 3` a bad way to build a 3×3 grid?

_____

K2. `nums = [1, 2, 3, 4, 5]` and `evens = [n for n in nums if n%2==0]`. Why is `evens` shorter than `nums`?

_____

K3. `[0] * 3` is a safe way to build one row, and `[[0] * 3] * 3` is not a safe way to build a grid. What makes the difference?

_____

K4. The Bug Hunt program that wrote `range(len(grid))` for the columns printed every 3×3 grid correctly. Why did it go wrong only on the grid of 2 rows and 4 columns?

_____

Part L • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full. Run the sample call on paper before you call it done.

L1. `count_in_grid(grid, target)`: return how many cells of `grid` hold `target`. `count_in_grid([[1, 0], [0, 0]], 0)` is 3.

L2. `make_grid(rows, cols, value)`: return a new grid with `rows` rows and `cols` columns, every cell holding `value`. **Changing one cell afterwards must change that cell only.** `make_grid(2, 3, 0)` is `[[0, 0, 0], [0, 0, 0]]`.

L3. `row_win(board, player)`: `board` is a tic-tac-toe board like `B` in H3. Return `True` if some **row** holds `player` in all three cells, and `False` otherwise. This is the first third of `check_winner`.

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. Open `pythontutor.com`, paste the two lines `grid = [[0] * 3] * 3` and `grid[0][0] = 5`, and press *Next*. How many arrows end at the one row? Then change line 1 to the comprehension of I3 (a) and run it again. Do both pictures match the ones you drew?
2. Type `G = [[3, 1, 4], [1, 5, 9], [2, 6, 5]]`. Read the last line of the error from each of `G[3][0]`, `G[0][3]` and `G[1, 2]`. Two of them have the same name. The third is what Python says when one pair of brackets is written where two are needed.
3. Build the slides' table: `table = [[r*c for c in range(1, 6)] for r in range(1, 6)]`. Predict `table[3][4]`, `len(table)`, `sum(table[0])` and `[row[-1] for row in table]`, then check each. Last, the loop `for row in table:` with `print(row)` inside it: the quickest way to see a grid as a grid.

Stuck on more than two items? Re-read the Lecture 13 slides "List Comprehension Examples", "Two-Step Indexing", "Row vs Column Traversal" and "The Aliasing Trap", and the four Bug Hunts, before the next class — then redo Parts C, G and I cold. For a comprehension, say the three steps aloud: *values in, values kept, list out.* For a grid, say *row first, then column*, and you will land on the right cell.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. a new **list**; an **expression** (a function applied to each element); a **test** (condition). The test is the only optional part.

A2. a **list** of **lists**; `G[2][0]` — **row first**, then column.

A3. `len(G)` = number of **rows** (the outer list); `len(G[0])` = number of **columns** (how long one row is).

A4. **drops** (filters out); **shorter**; a **value**; its `else`. With `if ... else` nothing is dropped, so the new list is as long as the sequence.

A5. a whole **row**, which is a **list**; then one **element** (the one at index 2 of that row).

A6. (a) **False** — it *creates a new list* and leaves the original alone. (b) **True** — one index gets a row; you need two indices to get a number. (c) **False** — it builds one row and points at it three times (the aliasing trap). (d) **True**. (e) **False** — they hold the same two digits in the opposite order: row 1 column 2, and row 2 column 1. (f) **True** — when no element passes the test. (g) **False** — `row` is a whole row, a list; it takes a second loop to reach the numbers. (h) **False** — `len(G)` counts rows, so it is `2`; `len(G[0])` is `5`.

B1. `[0, 2, 4, 6]` — `range(4)` gives `0,1,2,3` and each one is doubled.

B2. `[2, 6]` — only `1` and `3` pass the test, so the new list has **2** items, not 4. A test makes the new list *shorter* than the sequence.

B3. `[[0, 0], [1, 1], [2, 4]]` — the expression is itself a list, so we get a list of lists (a 2D list).

B4. `[3, 4]`, then `[3, 0, 4, 0]`, then `[3, -1, 4, -5]`. The `if` at the end *drops* the two negatives. The `if ... else` in front keeps all four places and only chooses what goes into each. Neither line changes `nums`: a comprehension builds a new list.

B5. `[3, 4, 2, 4]`, then `['g', 'l']`, then `['b', 'n', 'n']`. In `C` the sequence is a string, so `ch` is one character at a time, and the result is still a *list*.

B6. `[1, 2, 3, 9] [2, 3, 4]`, and **2** list objects. The comprehension on line 3 builds a second list and `L =` points the name `L` at it. `K` still names the first list, and that is the one the `9` goes into (Lecture 12).

C1. Row 2: `9 4 7 10`; `9 7 10`; `[4, 3, 5]`. Row 3: `'list' 'of' 'rows'`; `'list' 'rows'`; `['t', 's']`. Row 4: `3 2 1`; all three (there is no test); `[7, 8, 9]`. Row 5: `9 10 11 12 13`; `9 12`; `[9, 12]`.
The test looks at the value that came *in*, never at the value that goes out: in row 2 the `9` is kept because `9 > 5`, and what is stored is `4`. In row 5 the expression is just `n`, so the kept values go in unchanged.

C2. Left column: **7, 4, 3, 3**. Right column: **2, 7, 1, 0**.
With no test the new list has one item for each value of the sequence, whatever the expression is: seven zeros, and three 25s. When the expression is a list, each item is a whole list: `[[e, e] ...]` has 4 items, not 8, and `[[0] * 5 ...]` has 3 rows, not 15 numbers. On the right, a test at the end can only shorten (2, 1 and 0 items; a test that nothing passes gives an empty list, not an error); `if ... else` in front never does (7).

D1. (a) `big = [m for m in marks if m >= 50]`
(b) `halves = [n / 2 for n in range(10)]`
Read the loop from the inside out: what is appended goes first, the `for` line is copied as it stands, and an `if` around the `append` goes last.

D2. (a) `tens = []` / `for n in nums:` / `if n > 2:` / `tens.append(n * 10)`
(b) `col = []` / `for row in G:` / `col.append(row[1])`. For a grid `G` this collects the item at index 1 of every row: column 1.

E1. `4` and `2`. Same two digits, opposite order, different cell — **row comes first**, so `G[0][2]` is row 0 column 2.

E2. `[1, 5, 9]` — one index selects a whole **row**, and a row is a list.

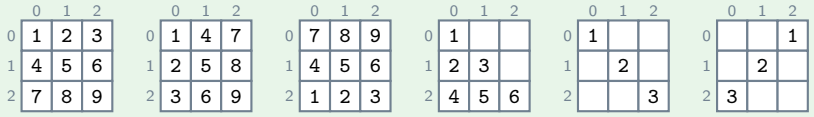
E3. `15` (row 1: $1+5+9$) and `6` (column 0: $3+1+2$). Summing a row is easy because a row already is a list; a column has to be collected with a comprehension.

E4. `5`, `9` and `[2, 6, 5]`. A negative index counts from the end, in either step: `G[-1]` is the last row, and `G[1][-1]` is the last item of row 1.

E5. `G[1][2]`, `G[0][2]`, `G[1][1]` and `G[2][2]`. Equal values in different cells have different addresses.

F. `G[1][1]` — **one number** (`5`); two indices always land on a single cell.
`G[1]` — **a flat list** (`[1, 5, 9]`); one index gives a whole row.
`sum(G[0])` — **one number** (`8`).
`[G[r][0] for r in range(3)]` — **a flat list** (`[3, 1, 2]`); each item is one cell, so nothing nests.
`[[0] * 3 for r in range(3)]` — **a list of lists**; the expression `[0] * 3` is itself a row.
`G[0] + G[1]` — **a flat list** (`[3, 1, 4, 1, 5, 9]`); `+` joins two rows end to end into one list of six numbers.
`[sum(row) for row in G]` — **a flat list** (`[8, 15, 13]`); each item is one number, a row's total.
`G[1:]` — **a list of lists** (`[[1, 5, 9], [2, 6, 5]]`); a slice of a grid is a shorter list of rows.
`[[x * 2 for x in row] for row in G]` — **a list of lists**; the expression is a comprehension, so each item is a new row.

G1. From (a) to (f):



The *inner* loop is the index that changes fastest. In (a) that is `c`, so the visit runs along a row (row-major); in (b) it is `r`, so it runs down a column (column-major). (c) is row-major with the rows taken from the bottom. In (d) row `r` is given only `r + 1` columns, so the loop stops at the diagonal. (e) and (f) need one loop, because each row has exactly one cell on a diagonal: `[i][i]` and `[i][2 - i]`.

G2. (a) **2** lines, the first is `1 2`. (b) **4** lines, the first is `1`. (c) **1** line, `1 2 3 4`.
`end=' '` keeps the numbers of a row on one line; the bare `print()` ends that line. It belongs to the *outer* loop, so that it runs once for each row. Inside the inner loop it runs after every number, and without it no line is ever ended.

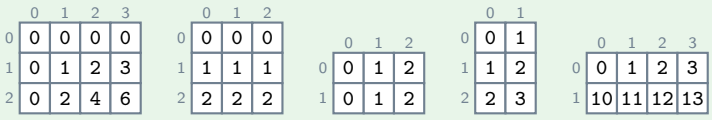
G3. `(0, 0)` after 1; `(0, 2)` after 3; `(1, 1)` after 5; `None` after 6. A value that is not there costs a look at every cell, and only then does line 6 run. Column by column the visit goes 4, 8: `find(g, 8)` returns `(1, 0)` after **2**. Both answers hold an 8; "first" depends on the order of the two loops.

H1. 2, 3, 10; then *error*, *error*, 8. Two rows and three columns. `T[2][1]` fails at the first step: there is no row 2. `T[0][3]` gets row 0 and then fails, because that row has no index 3. Both are an `IndexError`. `T[1][2]` and `T[2][1]` hold the same two digits, and only one of them exists.

H2. 1: 21. 2: `[20, 21]`. 3: 41. 4: `[9, 3]`. 5: `[7, 8]`. 6: 14. 7: `[9, 10]`. Quiz 2's total is row **6**.
A row is already a list, so `sum` and `max` work on it directly (rows 1, 2 and 7). A column is not a list anywhere in memory: it has to be collected, one cell from each row (rows 4, 5 and 6). Rows 4 and 5 are two spellings of the same idea: `row[1]` is `T[r][1]`.

H3. (a) row 1: `['O', 'X', 'O']`. (b) column 2, the right-hand one: `['X', 'O', 'O']`. (c) the diagonal from the top left: `['X', 'X', 'O']`. (d) the diagonal from the top right: `['X', 'X', 'X']`. Line (d) wins, for **X**. These four shapes are all that `check_winner` has to look at: 3 rows, 3 columns and 2 diagonals.

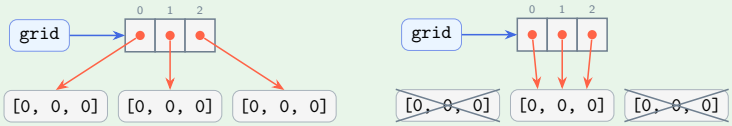
I1. From (a) to (e):



The inner comprehension builds *one row*, so its range, `C`, is the number of columns. The outer `for` repeats it, so `R` is the number of rows. `r` stays fixed along a row (b) and `c` counts along it (c).

I2. (a) `r + 1` (b) `(r + 1) * (c + 1)`, the slides' multiplication table (c) `1 if r == c else 0`: this is the `if ... else` that picks a value (d) `(r + c) % 2`. Check any answer on two cells: the top-left one, where `r` and `c` are both 0, and one more.

I3. (a) on the left, (b) on the right:



(a) One arrow from each dot to a row of its own: **4** list objects (one outer, three rows), and `[[0, 0, 0], [0, 0, 7], [0, 0, 0]]`. (b) All three arrows end at *one* row, and the other two boxes are

crossed out: **2** list objects, and `[[0, 0, 7], [0, 0, 7], [0, 0, 7]]`. `[0] * 3` is worked out once, and `* 3` then copies the pointer to that one row. In (a) the comprehension works out `[0] * 3` again on every round, so each round builds a new row.

I4. **1, 3, 3, 3, 1, 1.** Ask of each program: how many times is a row *built*? In rows 3 and 4 it is built once, on line 1, and `g` holds three pointers to it. Row 4 shows that a comprehension is no cure by itself: its expression is the name `row`, and a name builds nothing. Row 5 clones the row on every round. In row 6 the grid starts as the trap, but `g[0] = ...` puts a new list into position 0, so row 0 is on its own (rows 1 and 2 are still one list).

J1. It prints `[[5, 0, 0], [5, 0, 0], [5, 0, 0]]`. `* 3` copies the *same* row three times, so all three rows are one list. Fix: `grid = [[0] * 3 for r in range(3)]`.

J2. `L = [e*e for e in range(6) if e%2==0]`. The test goes **last**: expression, then `for`, then `if`. (Result: `[0, 4, 16]`.)

J3. `SyntaxError`. This is J2 turned round. An `if` with an `else` chooses a value, so it is part of the expression and goes **first**: `L = [m if m >= 50 else 0 for m in marks]`. Only an `if` with no `else` may stand at the end.

J4. (a) It prints `15` and `12`, and never reaches the third quiz. `len(T)` is the number of *rows* (2); the columns are counted by `len(T[0])` (3). Fix line 2: `for c in range(len(T[0])):`. Line 4 is right as it is: there `r` does count rows.
(b) It prints `20`, then `41`. `total` is set to 0 once, so the second student starts from the first student's 20. Fix: move `total = 0` inside the outer loop, as its first line.

J5. (a) It returns `False`. The very first cell is a `1`, and the `else` returns at once, so no other cell is ever looked at. Fix: delete lines 6 and 7, and put `return False` after both loops, in line with the outer `for`. Only when every cell has been seen can the answer be "no". (It is Bug Hunt 4 again, one level deeper.)
(b) It prints `[[1, 2], [3, 4]]`. `x = 0` points the name `x` at a zero and leaves the grid alone (Lecture 11). To change a cell, assign to the cell: `for r in range(len(g)):` / `for c in range(len(g[r])):` / `g[r][c] = 0`.

K1. Because it stores three pointers to the *same* inner list, so changing one cell changes that column position in every row.

K2. Because the `if` test filters: only the elements that pass it are put into the new list, so a comprehension with a test can produce fewer items than it looked at.

K3. `*` copies pointers in both. Three pointers to one `0` do no harm, because a number cannot be changed: `row[0] = 5` puts a new number into one position. Three pointers to one *list* do harm, because a list can be changed in place, and that change shows through all three.

K4. On a square grid the number of rows equals the number of columns, so the wrong count happens to be the right number. Only a grid that is wider than it is tall (or taller than wide) makes the two counts differ.

L1. `count = 0` / `for row in grid:` / `for x in row:` / `if x == target:` / `count += 1` / then `return count` after both loops. The search of G3 may stop at the first match; a count may not, so there is no `return` inside the loops.

L2. `return [[value] * cols for r in range(rows)]`. The inner `[value] * cols` is one row, and the comprehension builds it afresh for every row. `[[value] * cols] * rows` gives the same printout and fails the test in bold.

L3. `for row in board:` / `if row == [player] * 3:` / `return True` / then `return False` after the loop. (Comparing the three cells one by one, or the slides' `all(...)`, is as good.) The `return False` stands outside the loop for the reason of J5: one row that fails says nothing about the rows below it.