Lab 08 — The Dig

List comprehensions; grids as lists of lists

DURATION ~3 hours **COVERS** Lectures 13 & 14 **TASKS** 7 files **CHECKER** `check.py`

How this lab works

- Every section opens with a worked example. It is a file in your folder, `example1.py` to `example6.py`: open it in Thonny, run it, and see the run printed on the sheet.
- Work top to bottom. Read a task, then its five keys. Four are practical: type a line in Thonny's Shell, or change one thing in the example file and run it again, and look at what it shows. The fifth is worked out by hand, Shell closed, with the small numbers it gives.
- Type the five answers into `check.py` when it asks. All five right, and the task's file opens. A wrong answer costs nothing: type the line again and look.
- Tasks are the deliverable: `task01.py` ... `task07.py` in your `lab08` folder. Aim to pass them all. A failed check costs nothing: read its reason and try again.
- Run `check.py` as often as you like.
- Ask. Talk to your neighbour, the TA or the instructor about the idea. Type your own code: no copy-paste.
- The minutes on a task are a guide for pacing, not a limit.
- This lab assumes Lab 07: building a list with `append`, `split` and `int()`, a function that returns a new list, and an alias against a clone (`[:]`).
- Tasks 8 and 9 are a bonus: optional, and not counted.

Before you start: get the files

Download the `lab08` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task09.py`, the worked examples `example1.py` ... `example6.py`, and `check.py`. Keep everything in that **one folder**, and do not rename any file — the checker finds your work by its file name. The example files are yours to change; nothing in them is handed in.

The story of this lab

Mohenjo-daro is a city of brick in Sindh. People lived there about 4500 years ago. Diggers found it in 1922. They mark the ground as a grid of squares. A row of squares is a trench. They write down every find: a bead, a seal, a wall. All the code in this lab is about their work.

1 A New List From an Old One (~14 min)

Ref: Lecture 13 — List Comprehensions, 2D Lists & Grids

A comprehension builds a new list in one line: an expression, then `for` a name `in` a list. It makes one new item for every old one, and the old list stays as it was. You will build lists this way, and read a line of numbers with one.

Worked example: finds and their layers

The ground is dug in layers 50 cm thick. Layer 0 is the top one. A find at 125 cm lies in layer 2, because `125 // 50` is 2. This program reads the depths of some finds and gives each one its layer. It is `example1.py` in your folder: open it, run it, type the four lines, and see this run.

```
1 n = int(input())
2 depths = []
3 for i in range(n):
4     depths.append(int(input()))
5 layers = [d // 50 for d in depths]
6 print("Depths:", depths)
7 print("Layers:", layers)
8 print("Deepest layer:", max(layers))
```

Input

```
3
40
125
260
```

Output

```
Depths: [40, 125, 260]
Layers: [0, 2, 5]
Deepest layer: 5
```

Notice: line 5 builds the whole new list in one line: the expression `d // 50`, then `for d in depths`. The list `depths` is printed unchanged on line 6. Key 1.1 changes the 50.

Three things check.py is strict about

- Read each line with `input()` and nothing inside the brackets, as in the CCC problems. A prompt such as `input("n: ")` is printed too, so the task fails.
- A key's answer is what the Shell showed, typed as it showed it: a number, `True`, `False`, `None`, an error's name such as `IndexError`, or the one word a `print` showed. No units, no quotes. A by-hand key's answer has the same shape.
- `check.py` also calls your function by itself, as you can in the Shell after F5. This week it first types a grid into the Shell, calls your function on it, and then looks at that grid again. So keep the exact name and parameters the task gives, and hand the answer back with `return`. Print nothing inside the function, and change the grid only when the task says so.

TASK 1 Layers — EASY — task01.py — ~11 min

⌨ The diggers write down how deep each find lay, in centimetres. On this dig the layers are not always 50 cm thick: the team sets the thickness. The depths of a day's finds come on one line, with spaces between them.

Part 1, the function: `layer_of(depths, thick)` returns a **new** list: the layer of each depth, in the same order, which is `depth // thick`. The list it is given does not change. It prints nothing. In the Shell, `layer_of([40, 125, 260], 50)` gives `[0, 2, 5]`.

Part 2, the program: Read the thickness, then the depths from one line. Print the layers and the deepest one.

Input: Line 1: the thickness of a layer in cm, a whole number, 1 or more. Line 2: the depths in cm, whole numbers with single spaces between them, at least one.

Output: `Layers:` and the list, as `print("Layers:", layers)` prints it; then `Deepest layer: L`, the largest layer number.

Sample 1 (the checker uses this one)

Input

```
50
40 125 260 95
```

Output

```
Layers: [0, 2, 5, 1]
Deepest layer: 5
```

Why: 40 // 50 is 0, 125 // 50 is 2, 260 // 50 is 5 and 95 // 50 is 1. The largest of the four is 5.

Hint: the second line is text until you split it and turn each piece into a number: `[int(x) for x in input().split()]`. Keys 1.3 and 1.4 show the two halves of that line. Then one more comprehension, inside the function, gives the layers.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task01.py` opens.

Key 1.1 Run `example1.py` and type `3`, then `40`, `125`, `260`. Then change the `50` in `d // 50` to `100` and run it again on the same input. What number is on the `Deepest layer:` line now?

Answer: .....

Key 1.2 Type `len([d * 2 for d in [40, 125, 260]])`. What came back?

Answer: .....

Key 1.3 Type `"40 125 260".split()` and look at the pieces.
Now type `len("40 125 260".split())`. What came back?

Answer: .....

Key 1.4 Type `sum([int(x) for x in "40 125".split()])`. What came back?

Answer: .....

Key 1.5 **By hand, Shell closed.** `layer_of([99, 100, 149], 50)` returns a new list. How many of its items are `1`?

Answer: .....

2 Keep Some, Drop Some (~26 min)

Ref: Lecture 13 — List Comprehensions, 2D Lists & Grids

A comprehension can have an `if` **after** the `for`. It keeps the items that pass the test and drops the rest. A different `if`, with an `else`, goes **before** the `for`: it picks a value for every item and drops none. Task 2 is a program with two bugs. Task 3 changes it.

Worked example: micro beads

Thousands of beads were found at Mohenjo-daro. Some are tiny: a bead of 5 mm or less is a micro bead. This program keeps the micro beads, counts them, and then gives every bead a

word. It is `example2.py`: run it; nothing is typed.

```
1 sizes = [1, 7, 2, 12, 3, 5]
2 micro = [s for s in sizes if s <= 5]
3 print("Micro beads:", micro)
4 print("Count:", len(micro))
5 kinds = ["micro" if s <= 5 else "large" for s in sizes]
6 print(kinds)
```

Output

```
Micro beads: [1, 2, 3, 5]
Count: 4
['micro', 'large', 'micro', 'large', 'micro', 'micro']
```

Notice: on line 2 the `if` comes **after** the `for` and drops two beads. On line 5 the `if ... else` comes **before** the `for`, and every bead gets a word: six beads, six words. The 5 mm bead passes `<= 5`. Key 2.1 changes that test.

TASK 2 Bead Sort

EASY `task02.py` ~12 min

⌨ The team sorts beads by size. A bead of the limit or smaller is a small bead. `task02.py` reads the limit and the sizes, and `small(sizes, limit)` should return the small beads as a new list. Right now the program does not run at all, and once that is fixed it prints every bead.

Your task: It has exactly **two** bugs. Find and fix them. Do not rewrite the program. When both are fixed, in the Shell `small([6, 1, 4, 9], 4)` gives `[1, 4]`.

Input: Line 1: the limit in mm, a whole number, 0 or more. Line 2: the sizes in mm, whole numbers with single spaces between them, at least one.

Output: `Small beads:` and the list, as `print` prints it; then `Small: K`, how many it holds.

Sample 1 (the checker uses this one)

Input

```
4
6 1 4 9 2 4
```

Output

```
Small beads: [1, 4, 2, 4]
Small: 4
```

Why: 1, 4, 2 and 4 are 4 or less, so they are small. 6 and 9 are not. The order is the order typed.

Hint: find the bugs by **bisecting the program**: put `print(kept)` on its own line inside the function, run it, and see whether the list is right there. If it is, the bug is after that line. Key 2.2 names the first bug's error; Key 2.4 is the fact behind the second: a comprehension builds a new list and leaves the old one alone, so look at what the function hands back.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task02.py` opens.

Key 2.1 ⌨ Run `example2.py`. Then change the `<= 5` in `micro = [s for s in sizes if s <= 5]` to `< 5` and run it again. What number is on the `Count:` line now?

Answer: .....

Key 2.2 ⌨ Open `task02.py` and run it as it is. The last line of the error starts with an error's name. Which?

Answer: .....

Key 2.3 Type `len([s for s in [1, 7, 5] if s <= 5])`. What came back?

Answer: .....

Key 2.4 Type `b = [1, 7, 5]` and then `k = [s for s in b if s <= 5]`. Now type `len(b)`. What came back?

Answer: .....

Key 2.5 **By hand, Shell closed.** `small([5, 6, 4, 9], 5)` returns a new list. How many items does it hold?

Answer: .....

TASK 3 Micro or Large

EASY `task03.py` ~11 min

The museum label needs a word for every bead, not a shorter list. A bead of the limit or smaller is `micro`; any other is `large`.

Part 1, the function: `kind(sizes, limit)` returns a **new** list of words, one for each size and in the same order: `"micro"` for a size of at most `limit`, `"large"` for any other. It prints nothing. In the Shell, `kind([6, 1, 4], 4)` gives `['large', 'micro', 'micro']`.

Part 2, the program: Start from your fixed `task02.py`: copy it into this file, then change it. Rename `small` to `kind` and put the `if ... else` **before** the `for` in its comprehension. Print the words, then how many are `micro`.

Input: Two lines, as in Task 2: the limit, then the sizes.

Output: `Kinds:` and the list of words, as `print` prints it; then `Micro: K`, how many words are `micro`.

Sample 1 (the checker uses this one)

Input

```
4
6 1 4 9 2 4
```

Output

```
Kinds: ['large', 'micro', 'micro', 'large', 'micro', '
    micro']
Micro: 4
```

Why: Six beads, six words: 6 and 9 are large, the other four are micro. The count is how many words are micro.

Hint: the count is the length of a filtered list, as in Task 2: `[k for k in kinds if k == "micro"]`. Key 3.4 is that line on three words.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task03.py` opens.

Key 3.1 In `example2.py` change the word `"large"` to `"big"` and run it again. On the last line, what is the second word?

Answer: .....

Key 3.2 Type `("x" if 7 <= 5 else "y") == "y"`. What came back?

Answer: .....

Key 3.3 Type `len(["micro" if s <= 5 else "large" for s in [1, 7, 5]])`. What came back?

Answer: .....

Key 3.4 Type `len([w for w in ["micro", "large", "micro"] if w == "micro"])`. What came back?

Answer: .....

Key 3.5 **By hand, Shell closed.** `kind([5, 10, 2, 8], 5)` returns a new list of words. How many of them are `large`?

Answer: .....

3 The Site Grid: Rows and Columns (~34 min)

Ref: Lecture 13 — List Comprehensions, 2D Lists & Grids

A grid is a list of lists: each inner list is one row. Two indexes reach one square, `site[r][c]`, the row first. `len(site)` is the number of rows, and `len(site[0])` the number of columns. A column is one position taken from every row.

Worked example: the site grid

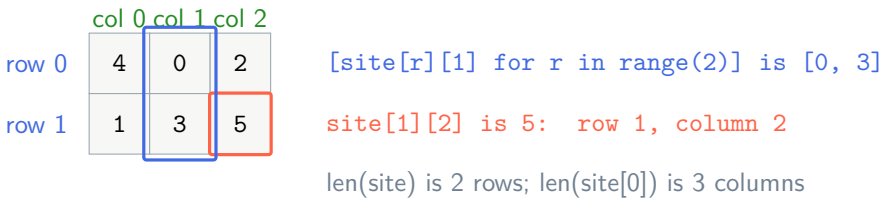
The team writes the number of finds in each square into a grid, one row per trench. This program looks at one square, one trench and one column. It is `example3.py`: run it; nothing is typed.

```
1 site = [[4, 0, 2],
2         [1, 3, 5]]
3 print("Square (1, 2):", site[1][2])
4 print("Trench 0:", sum(site[0]))
5 column = [site[r][1] for r in range(len(site))]
6 print("Column 1:", sum(column))
7 print("Rows:", len(site), "Columns:", len(site[0]))
```

Output

```
Square (1, 2): 5
Trench 0: 6
Column 1: 3
Rows: 2 Columns: 3
```

Notice: `site[1][2]` on line 3 is row 1, then column 2. Line 5 builds a column: the same column 1, taken from each row `r`. This grid has 2 rows and 3 columns, so `site[2]` does not exist: Key 4.1 asks for it.



TASK 4 Trench Totals

MEDIUM task04.py ~16 min

The dig's grid has r trenches of c squares each. The finds of one trench are the numbers in its row. The team wants the total of every trench, and the total for the whole site.

Part 1, the function: `trench_totals(site)` returns a **new** list: the sum of each row of `site`, in order. The grid it is given does not change. It prints nothing. In the Shell, `trench_totals([[4, 0, 2], [1, 3, 5]])` gives `[6, 9]`.

Part 2, the program: Read the grid, then print the totals:

Step 1: Read r and c from the first line with `r, c = [int(x) for x in input().split()]`. Then read r lines, each into a list of numbers the same way, and append each list to `site`. Print `site`. Run it on the sample: you should see the three rows inside one pair of brackets.

Step 2: Define `trench_totals` above, with one comprehension that uses `sum(row)`. Replace the print with `Trenches:` and what the function returns. Run it: you should see `Trenches: [7, 9, 4]`.

Step 3: Print `Total finds:` and the sum of the totals. Run it: you should see the sample.

Input: Line 1: r and c, the rows and columns, with a space between, each 1 or more. Then r lines: one row each, c whole numbers with single spaces between them.

Output: `Trenches:` and the list of totals, as `print` prints it; then `Total finds: T`, the sum of every square.

Sample 1 (the checker uses this one)

Input

```
3 4
4 0 2 1
1 3 5 0
0 0 2 2
```

Output

```
Trenches: [7, 9, 4]
Total finds: 20
```

Why: $4 + 0 + 2 + 1$ is 7, $1 + 3 + 5 + 0$ is 9, $0 + 0 + 2 + 2$ is 4. The three totals add up to 20.

Hint: `sum` adds up one row, as Key 4.2 showed. A comprehension over the rows, `for row in site`, gives one total per row. The whole site is `sum` of that list: Key 4.3.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task04.py` opens.

Key 4.1 Run `example3.py`. Then change `site[1][2]` on the `Square` line to `site[2][1]` and run it again. The last line of the error starts with an error's name. Which?

Answer: .....

Key 4.2 Type `g = [[4, 0, 2], [1, 3, 5]]`, then `sum(g[1])`. What came back?

Answer: .....

Key 4.3 Type `sum([sum(row) for row in g])`. What came back?

Answer: .....

Key 4.4 Type `len("4 0 2 1".split())`. What came back?

Answer: .....

Key 4.5 By hand, Shell closed. `trench_totals([[2, 2], [0, 5], [1, 0]])` returns a new list. What is its last item?

Answer: .....

TASK 5 Column Count

MEDIUM `task05.py` ~15 min

⌨ The same grid as Task 4. A column of squares runs across every trench, from the top row to the bottom one. `task05.py` holds the whole program already, with one gap: the body of `column_total`, marked `# TODO`.

Part 1, the function: `column_total(site, c)` returns the sum of column `c`: `site[r][c]` for every row `r`. It prints nothing. In the Shell, `column_total([[4, 0, 2], [1, 3, 5]], 2)` gives 7.

Part 2, the program: It is written: it reads the grid and prints `Column j: T` for every column `j`. Fill the gap:

Step 1: Run the file as it is on the sample. Every line ends in `None`: the function has no `return` yet.

Step 2: Replace `pass` with two lines: build the column with a comprehension over the rows, `site[r][c] for r in range(len(site))`, and return its sum. Run it: you should see the sample.

Input: Line 1: r and c, the rows and columns, with a space between, each 1 or more. Then r lines: one row each, c whole numbers with single spaces between them.

Output: one line per column, `Column j: T`, for `j` from 0.

Sample 1 (the checker uses this one)

Input

```
3 4
4 0 2 1
1 3 5 0
0 0 2 2
```

Output

```
Column 0: 5
Column 1: 3
Column 2: 9
Column 3: 3
```

Why: Column 0 is 4 + 1 + 0, read down the rows. The grid has 3 rows and 4 columns, so there are four lines.

Hint: row first, then column: `site[r][c]`, never `site[c][r]`. On this grid `site[3]` does not exist, so the swapped version crashes. The loop over the rows runs `range(len(site))`: Key 5.2 is that number, Key 5.3 the other one.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task05.py` opens.

Key 5.1 ⌨ In `example3.py` change `site[r][1]` on the `column` line to `site[r][2]` and run it again. What number is on the `Column 1:` line now?

Answer: .....

Key 5.2 ⌨ Type `g = [[4, 0, 2], [1, 3, 5]]`, then `len(g)`. What came back?

Answer: .....

Key 5.3 ⌨ Now type `len(g[0])`. What came back?

Answer: .....

Key 5.4 ⌨ Type `sum([g[r][0] for r in range(2)])`. What came back?

Answer: .....

Key 5.5 ✎ **By hand, Shell closed.** `column_total([[1, 2], [3, 4], [5, 6]], 1)`. What does it return?

Answer: .....

4 A Fresh Sheet (~15 min)

Ref: Lecture 13 — List Comprehensions, 2D Lists & Grids

A new grid of zeros is built with a comprehension: one new row for every `r` in `range(rows)`. `[[0] * cols] * rows` looks the same when printed, but it is one row repeated: a change in one square shows in every row.

Worked example: a fresh sheet, two ways

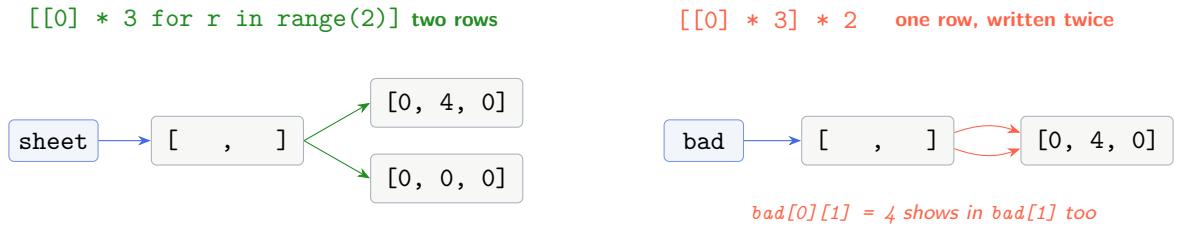
Each day the team starts a new tally sheet, a grid of zeros. This program builds one the right way and one the wrong way, and changes one square of each. It is `example4.py`: run it; nothing is typed.

```
1 sheet = [[0] * 3 for r in range(2)]
2 sheet[0][1] = 4
3 print("Good:", sheet)
4 bad = [[0] * 3] * 2
5 bad[0][1] = 4
6 print("Bad:", bad)
7 print(bad[0] is bad[1])
```

Output

```
Good: [[0, 4, 0], [0, 0, 0]]
Bad: [[0, 4, 0], [0, 4, 0]]
True
```

Notice: line 1 builds a new `[0] * 3` for each row, so line 2 changes one square. Line 4 makes one row and repeats it, so line 5 changes "both" rows: they are one list, and `is` on line 7 says so. Key 6.1 changes the `range` on line 1.



TASK 6 Tally Sheet

EASY `task06.py` ~12 min

Every find is one tally mark in its square: the number there goes up by one. The day's finds come as pairs, the row and the column of each.

Part 1, the function: `tally(sheet, r, c)` **changes** the grid it is given: the number in row `r`, column `c` goes up by 1. It returns nothing (`None`) and prints nothing. In the Shell, with `s = [[0, 0], [0, 0]]`, after `tally(s, 1, 0)` the grid `s` is `[[0, 0], [1, 0]]`.

Part 2, the program: Read the size of the sheet and the finds. Build a fresh sheet of zeros with a comprehension, tally every find, and print the sheet row by row.

Input: Line 1: `rows` and `cols`, with a space between, each 1 or more. Line 2: n, the number of finds, 0 or more. Then n lines: the row and the column of one find, with a space between.
Output: one line per row of the sheet, as `print(row)` prints it.

Sample 1 (the checker uses this one)

Input

```
2 3
4
0 1
1 2
0 1
1 0
```

Output

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

Why: Square (0, 1) is tallied twice, so it holds 2. Squares (1, 2) and (1, 0) hold 1 each. The other three squares stay 0.

Hint: build the sheet as line 1 of the worked example does, with `cols` and `rows` in place of the numbers. Key 6.2 is what the `*` version does to a tally. Do not store what `tally` gives back: print the sheet itself afterwards.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task06.py` opens.

Key 6.1 Run `example4.py`. Then change `range(2)` in the `sheet` line to `range(3)` and run it again. How many inner lists are on the `Good:` line now?

Answer: .....

Key 6.2 Type `b = [[0] * 2] * 2`, then `b[0][0] = 7`, then `b[1][0]`. What came back?

Answer: .....

Key 6.3 Type `g = [[0] * 2 for r in range(2)]`, then `g[0][0] = 7`, then `g[1][0]`. What came back?

Answer: .....

Key 6.4 Now type `g[0][0] += 1`, then `g[0][0]`. What came back?

Answer: .....

Key 6.5 **By hand, Shell closed.** `s` is a 2 by 2 sheet of zeros. Then `tally(s, 0, 0)` runs twice and `tally(s, 1, 1)` once. What is `sum(s[0])` afterwards?

Answer: .....

5 A Plan in Text (~19 min)

Ref: Lecture 14 — Problem Set 3

In the contest problems a grid comes in as lines of text, one line per row. A list of strings is a grid too: `plan[r][c]` is one character, the line first. Two loops, one over the lines and one over the characters, visit every square.

Worked example: a wall plan in text

The brick walls of Mohenjo-daro still stand. A plan of a wall is drawn in text, one line per course of bricks, top line first: `#` is a brick and `.` is a gap. This program counts the bricks and looks at one corner. It is `example5.py`: run it; nothing is typed.

```
plan = ["#.#", "###", "#.."]
bricks = 0
```

```
for line in plan:
    for ch in line:
        if ch == "#":
            bricks += 1
print("Bricks:", bricks)
print("Corner:", plan[2][0])
```

Output

```
Bricks: 6
Corner: #
```

Notice: the outer loop on line 3 takes one line, the inner loop on line 4 one character of it. `plan[2][0]` on line 8 is line 2, then its first character. Key 7.1 changes the last line of the plan.

TASK 7 Wall Height

MEDIUM task07.py ~16 min

A column of the plan is one position, read down every line. Its height is how many lines have a brick there. The team wants the height of every column, and the tallest.

Part 1, the function: `wall_height(plan, c)` returns how many lines of `plan` have a `#` at position `c`. It prints nothing. In the Shell, `wall_height(["#.#", "###", "#.."], 0)` gives 3.

Part 2, the program: Read the plan, then print the heights:

Step 1: Read n, then n lines into a list `plan` with `append`. Print `Width:` and the length of the first line. Run it on the sample: you should see `Width: 4`.

Step 2: Define `wall_height` above: a count, a loop over the lines, and an `if` on `line[c]`. Then loop over the columns with `range(len(plan[0]))` and print `Column c: H` for each. Run it: four more lines appear.

Step 3: Keep each height in a list as you go. After the loop print `Tallest:` and the largest. Run it: you should see the sample.

Input: Line 1: n, the number of lines in the plan, 1 or more. Then n lines: one line of the plan each, all the same length, made of `#` and `.` only.

Output: `Width: W`, the length of a line; then one line per column, `Column c: H`, for `c` from 0; then `Tallest: T`, the largest height.

Sample 1 (the checker uses this one)

Input

```
3
#.#.
###.
#...
```

Output

```
Width: 4
Column 0: 3
Column 1: 1
Column 2: 2
Column 3: 0
Tallest: 3
```

Why: Position 0 has a brick in all three lines. Position 3 has none. The plan is 3 lines by 4 columns, so there are four Column lines.

Hint: the columns run to the **width**, `len(plan[0])`, not to the number of lines: this plan is wider than it is tall. Key 7.4 is what happens one position past the width.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task07.py` opens.

Key 7.1 ⌨ In `example5.py` change the last line of the plan, `"#.."`, to `"###"` and run it again. What number is on the `Bricks:` line now?

Answer: .....

Key 7.2 ⌨ Type `p = ["#.#", "###"]`, then `p[1][2] == "#"`. What came back?

Answer: .....

Key 7.3 ⌨ Type `len(p[0])`. What came back?

Answer: .....

Key 7.4 ⌨ Type `p[0][3]`. The last line of the error starts with an error's name. Which?

Answer: .....

Key 7.5 ✎ **By hand, Shell closed.** `wall_height(["#..", "#.#", "..#"], 2)`. What does it return?

Answer: .....

6 Bonus: Every Square, and a Copy That Shares Nothing (~29 min)

Optional. Not counted. From Lecture 13 — List Comprehensions, 2D Lists & Grids

Two loops over the indexes visit every square, and a tuple `(r, c)` names one. A copy of a grid made with `[:]` is a new outer list whose rows are the old ones; a copy that shares nothing clones each row too. Task 8 collects squares. Task 9 builds the mirror of a grid.

Worked example: squares as tuples, rows as clones

This program lists the squares with more than 2 finds, then makes a copy of the site and turns one row of the copy round. It is `example6.py`: run it; nothing is typed.

```
1  site = [[4, 0], [1, 3]]
2  spots = []
3  for r in range(len(site)):
4      for c in range(len(site[r])):
5          if site[r][c] > 2:
6              spots.append((r, c))
7  print(spots)
8  copy = [row[:] for row in site]
9  copy[0].reverse()
10 print(copy)
11 print(site)
```

Output

```
[(0, 0), (1, 1)]
[[0, 4], [1, 3]]
[[4, 0], [1, 3]]
```

Notice: line 6 appends a tuple, round brackets, and there is no `return` in the loops: every square is looked at. Line 8 clones each row with `row[:]`, so reversing `copy[0]` on line 9 leaves `site` as it was on line 11. Key 8.1 changes the test on line 5; Key 9.1 takes the `[:]` off line 8.

TASK 8 Rich Squares

EASY `task08.py` ~12 min

Some squares hold many finds. A square is named by its row and column as a tuple, `(r, c)`. The team wants every square with more than k finds.

Part 1, the function: `rich_squares(site, k)` returns a **new** list of tuples `(r, c)`, one for every square whose number is more than `k`, row by row and left to right. The grid does not change. It prints nothing. In the Shell, `rich_squares([[4, 0], [1, 3]], 2)` gives `[(0, 0), (1, 1)]`.

Part 2, the program: Read the grid as in Task 4, then k. Print the rich squares and how many there are.

Input: Line 1: r and c, with a space between, each 1 or more. Then r lines: one row each, c whole numbers with single spaces between them. Last line: k, a whole number.

Output: `Rich squares:` and the list of tuples, as `print` prints it; then `Count: N`, how many it holds.

Sample 1 (the checker uses this one)

Input

```
2 3
4 0 2
1 3 5
2
```

Output

```
Rich squares: [(0, 0), (1, 1), (1, 2)]
Count: 3
```

Why: 4, 3 and 5 are more than 2. Square (0, 2) holds exactly 2, so it is not rich. Row 0 comes before row 1.

Hint: two loops over the indexes, as in the worked example, and an `append` of the tuple. More than k is a strict `>`: Key 8.4.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task08.py` opens.

Key 8.1 Run `example6.py`. Then change `> 2` in the `if` line to `> 0` and run it again. How many tuples are on the first line now?

Answer: .....

Key 8.2 Type `(1, 2)[0]`. What came back?

Answer: .....

Key 8.3 Type `L = []`, then `L.append((0, 1))`, then `len(L)`. What came back?

Answer: .....

Key 8.4 Type `3 > 3`. What came back?

Answer: .....

Key 8.5 **By hand, Shell closed.** `rich_squares([[3, 7], [8, 1]], 3)` returns a new list. How many tuples does it hold?

Answer: .....

TASK 9 Mirror the Plan

MEDIUM task09.py ~14 min

The plan of the site is drawn from the west. Seen from the east, every row runs the other way. The mirror must be a new grid: the original must not change, and no row may be shared between the two.

Part 1, the function: `mirror(site)` returns a **new** grid: every row of `site` reversed, as a new list. It shares no list with `site`, and `site` does not change. It prints nothing. In the Shell, with `s = [[4, 0, 2], [1, 3, 5]]`, `mirror(s)` gives `[[2, 0, 4], [5, 3, 1]]` and `s` is still `[[4, 0, 2], [1, 3, 5]]`.

Part 2, the program: Read the grid as in Task 4, then print the mirror:

Step 1: Define `mirror`: start a new empty list; for each row, clone it with `[:]`, reverse the clone, append it; return the list. Print the mirror row by row with `print(row)`. Run it on the sample: you should see the two reversed rows.

Step 2: Print `Original row 0:` and `site[0]` last. Run it: you should see the sample, with row 0 unchanged.

Input: Line 1: r and c, with a space between, each 1 or more. Then r lines: one row each, c whole numbers with single spaces between them.

Output: one line per row of the mirror, as `print(row)` prints it; then `Original row 0:` and `site[0]`, as `print` prints it.

Sample 1 (the checker uses this one)

Input

```
2 3
4 0 2
1 3 5
```

Output

```
[2, 0, 4]
[5, 3, 1]
Original row 0: [4, 0, 2]
```

Why: Each row is turned round. The original's first row still reads 4, 0, 2: the function reversed a clone, not the row.

Hint: `reverse()` turns a list round in place and hands back `None`, so reverse the clone on its own line and then append the clone's name. Key 9.1 is what happens without the `[:]`.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task09.py` opens.

Key 9.1 Run `example6.py`. Then change `row[:]` in the `copy` line to `row` and run it again. What is the first number on the last line now?

Answer: .....

Key 9.2 Type `r = [4, 0, 2]`, then `s = r[:]`, then `s.reverse()`, then `r[0]`. What came back?

Answer: .....

Key 9.3 Now type `s[0]`. What came back?

Answer: .....

Key 9.4 Type `len([row[:]  for row in [[1, 2], [3, 4]]])`. What came back?

Answer: .....

Key 9.5 **By hand, Shell closed.** `m = mirror([[1, 2, 3], [4, 5, 6]])`. What is `m[1][0]`?

Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

- ☐ I can build a new list from an old one with a comprehension, and I know the old list does not change.
- ☐ I can read a line of numbers with `[int(x) for x in input().split()]`.
- ☐ I can keep some items with an `if` after the `for`, and give every item a value with an `if ... else` before it.
- ☐ I can tell the two `if`s apart, and I know which one needs an `else`.
- ☐ I can bisect a program: print a value in the middle and see which half is wrong.
- ☐ I can reach one square of a grid with two indexes, row first.
- ☐ I know `len(site)` is the rows and `len(site[0])` the columns, and I use the right one in each loop.
- ☐ I can add up a row with `sum`, and a column with a comprehension over the rows.
- ☐ I can build a fresh grid of zeros with a comprehension, and I know why `[[0] * c] * r` is wrong.
- ☐ I can change one square of the caller's grid from inside a function.
- ☐ I can treat a list of strings as a grid and count down one column.
- ☐ I can visit every square with two loops and collect the ones that pass a test as tuples.