Lab 07 — Thin Air

Tuples and lists; aliases and clones

DURATION ~3 hours **COVERS** Lectures 11 & 12 **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 `lab07` 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 06: `def` and `return`, keeping a call's value in a name, a loop with a running total, and `if/elif/else`.
- Tasks 8 to 10 are a bonus: optional, and not counted.

Before you start: get the files

Download the `lab07` folder (from the course LMS page) onto your machine. It contains `task01.py` … `task10.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.

1 Tuples: One Parcel, Several Values (~15 min)

Ref: Lecture 11 — Tuples and Lists

A tuple holds several values in one object. So a function can hand back two answers with one `return`. The program that calls it opens the parcel into two names on one line. You will build a tuple, return it, and unpack it.

Worked example: a climb in two numbers

Above Base Camp a climber gains at most 300 m of sleeping height a day. This program takes two camp heights. It gives back the climb between them and the days it needs, as one tuple. It is `example1.py` in your folder: open it, run it, type the two heights, and see this run.

```
1 def camp_gap(low, high):
2     rise = high - low
3     return (rise, rise // 300)
4
5 low = int(input())
6 high = int(input())
7 metres, days = camp_gap(low, high)
8 print("Climb:", metres, "m")
9 print("Days:", days)
```

Input

```
5300
6100
```

Output

```
Climb: 800 m
Days: 2
```

Notice: line 3 builds one tuple with round brackets and returns it. Line 7 opens it: two names, a comma, `=`, the call. Keys 1.1 and 1.2 have you change line 3 and run it again.

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 list into the Shell, calls your function on it, and then looks at that list 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 list only when the task says so.

TASK 1 Loads

EASY `task01.py` ~11 min

⌨ On the Baltoro glacier, porters carry an expedition's loads up to Base Camp. The load is shared out in whole kilograms. What will not divide stays with the sirdar, the head porter.

Part 1, the function: `split_load(kg, porters)` returns the tuple `(each, left)`: the whole kilograms each porter carries, and the kilograms left over. It prints nothing. In the Shell, `split_load(100, 3)` gives `(33, 1)`.

Part 2, the program: Read the load and the number of porters. Call the function once, unpack the tuple into two names, and print both.

Input: Line 1: the load in kg, a whole number, 0 or more. Line 2: the number of porters, 1 or more.

Output: `Each porter: E kg`, then `Left over: L kg`.

Sample 1 (the checker uses this one)

Input	Output
`100` `3`	`Each porter: 33 kg` `Left over: 1 kg`

Why: Three porters take 33 kg each, which is 99 kg. One kilogram is left.

Hint: the `return` line builds the tuple with round brackets. One line in the program opens it: two names, a comma, `=`, the call, as in the worked example.

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 `5300`, then `7100`. What number is on the `Days:` line?

Answer: .....

Key 1.2 In `example1.py` change the `300` in `return (rise, rise // 300)` to `400`. Run it again with the same two heights. What is on the `Days:` line now?

Answer: .....

Key 1.3 Type `100 // 3`, then `100 % 3`. What did the second line show?

Answer: .....

Key 1.4 Type `2 // 5`: two kilograms for five porters. What came back?

Answer: .....

Key 1.5 **By hand, Shell closed.** `split_load(10, 4)` returns the tuple `(each, left)`. What is `left`?

Answer: .....

2 Lists: Build It, Index It, Slice It (~26 min)

Ref: Lecture 11 — Tuples and Lists

A list starts empty and grows with `append`. One index picks one item; a slice picks a run of items and gives them back as a new list. Task 2 is a program with two bugs. Task 3 builds a new list out of an old one.

Worked example: the loads in a list

The sirdar lists the loads in walking order, the heaviest at the front. This program reads them into a list, then looks at the list three ways. It is `example2.py`: run it and type the four lines.

```
n = int(input())
loads = []
for i in range(n):
    loads.append(int(input()))
print(loads)
print("Front:", loads[0])
print("First two:", loads[0:2])
```

Input

```
3
25
20
20
```

Output

```
[25, 20, 20]
Front: 25
First two: [25, 20]
```

Notice: `loads = []` sits above the loop and `append` inside it. The slice `[0:2]` gives two items, at indexes 0 and 1: it stops before index 2. Key 2.1 changes the index on line 6.

TASK 2 First Push

EASY `task02.py` ~12 min

⌨ In 1954 the Italian expedition that first climbed K2 placed its camps one above the other up the Abruzzi Spur. The first team's push reached the first k camps, counted from the bottom. `task02.py` reads the camps into a list of heights, and `first_camps(heights, k)` should return the first k as a new list. Right now the program crashes, and once the crash is fixed it prints one camp too many.

Your task: It has exactly **two** bugs. Find and fix them. Do not rewrite the program. When both are fixed, in the Shell `first_camps([5300, 6100, 6700, 7300], 2)` gives `[5300, 6100]`.

Input: Line 1: n, the number of camps, 1 or more. Line 2: k, how many camps the push reaches, 1 to n. Then n lines: the heights in metres, lowest camp first.

Output: the list of the first k heights, printed with `print`, then `Top of the push: H`, the last height in that list.

Sample 1 (the checker uses this one)

Input

```
6
3
5300
6100
6700
7300
7700
8000
```

Output

```
[5300, 6100, 6700]
Top of the push: 6700
```

Why: The first three camps are at indexes 0, 1 and 2. The slice stops before index 3.

Hint: find the bugs by **shrinking the failing input**. Key 2.2's error names the type `None`: which line put `None` into a name? Key 2.4 says what `append` hands back. With that fixed, try 2 camps and $k = 1$, the smallest input still wrong, and follow the code by hand: Key 2.5 is that count.

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` and type 3, 25, 20, 15. Then change `loads[0]` on the `Front:` line to `loads[2]` and run it again on the same input. What came after `Front:`?

Answer: .....

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

Answer: .....

Key 2.3 ⌨ Type `c = [5300, 6100, 6700]`, then `len(c[0:2])`. What came back?

Answer: .....

Key 2.4 Type `x = c.append(7300)`, then `print(x)`. What printed?

Answer: .....

Key 2.5 **By hand, Shell closed.** The list is `[5300, 6100, 6700, 7300]` and k is 2. How many items does the slice `[0:k + 1]` give?

Answer: .....

TASK 3 Giants

EASY `task03.py` ~11 min

Fourteen mountains on Earth rise above 8000 m. Five stand in Pakistan: K2 (8611 m), Nanga Parbat (8126), Gasherbrum I (8080), Broad Peak (8051) and Gasherbrum II (8035). Masherbrum (7821) and Rakaposhi (7788) stop just short.

Part 1, the function: `above(heights, limit)` returns a **new** list holding every height in `heights` that is more than `limit`, in the same order. The list it is given does not change. It prints nothing. In the Shell, `above([8611, 7821, 8126], 8000)` gives `[8611, 8126]`.

Part 2, the program: Read a limit and some heights. Print the heights above the limit, then how many there are.

Input: Line 1: the limit in metres. Line 2: n, the number of peaks, 0 or more. Then n lines: one height each, a whole number.

Output: the list of heights above the limit, printed with `print`, then `Count: C`, how many it holds.

Sample 1 (the checker uses this one)

Input

```
8000
5
8611
7821
8126
8051
7788
```

Output

```
[8611, 8126, 8051]
Count: 3
```

Why: K2, Nanga Parbat and Broad Peak are above 8000; the other two are not. The order is the order typed.

Hint: start an empty list above the loop, append inside an `if`, and return it after the loop. `len` counts; with $n = 0$ the list is empty and it counts 0.

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 Type `8000 > 8000`. What came back?

Answer: .....

Key 3.2 Type `len([])`. What came back?

Answer: .....

Key 3.3 Type `h = [8611, 7821, 8126]`, then `h[1] > 8000`. What came back?

Answer: .....

Key 3.4 Type `out = []`, then `out.append(8611)`, then `len(out)`. What came back?

Answer: .....

Key 3.5 **By hand, Shell closed.** `above([8051, 7788, 8035], 8040)` returns a new list. How

many items are in it?

Answer: .....

3 Strings to Lists and Back (~19 min)

Ref: Lecture 11 — Tuples and Lists

`split` cuts a string into a list of strings, and `join` glues a list of strings into one string. Watch the type at the border. A number read from a line is still text until `int()` reads it, and `join` will not take a number.

Worked example: the summit register

Base Camp on the Baltoro is at about 5000 m. The register holds one line per peak: its name, a comma, its height. This program prints how far each peak rises above Base Camp, then all the names on one line. It is `example3.py`: run it and type the three lines.

```
1 n = int(input())
2 names = []
3 for i in range(n):
4     parts = input().split(",")
5     names.append(parts[0])
6     print(parts[0], "rises", int(parts[1]) - 5000, "m")
7 print(", ".join(names))
```

Input

```
2
K2,8611
Broad Peak,8051
```

Output

```
K2 rises 3611 m
Broad Peak rises 3051 m
K2, Broad Peak
```

Notice: `split` gives strings, so line 6 reads the height with `int()` before subtracting. `join` on line 7 puts `", "` between the names and nowhere else. Key 4.1 moves Base Camp.

TASK 4 Route Log

MEDIUM task04.py ~17 min

An expedition keeps a log of its camps, one line per camp: the name, the height in metres and the load in kg, separated by commas with no spaces, as `Camp 1,6100,40`.

Part 1, the functions: `parse_camp(line)` returns the tuple `(name, height, load)`: the name as a string, the height and the load as whole numbers. `route_line(names)` returns one string: the names in the list, joined with `>` between them. Neither prints. In the Shell, `parse_camp("Camp 1,6100,40")` gives `('Camp 1', 6100, 40)`, and `route_line(['Base', 'Camp 1'])` gives `'Base > Camp 1'`.

Part 2, the program: Read the log. Print one line per camp, then the route, then the total load:

Step 1: Define `parse_camp`. Read n, then each line. Unpack the tuple and print `NAME: H m, L kg` for each camp. Run it on the sample: you should see the three camp lines.

Step 2: Append each name to a list as you go. Define `route_line`, and print `Route:` and its value after the loop. Run it: the `Route:` line appears.

Step 3: Add each load to a running total; print `Total load:` last. Run it: you should see the sample.

Input: Line 1: n, the number of camps, 1 or more. Then n lines: `name,height,load`, lowest camp first.

Output: one line per camp, `NAME: H m, L kg`; then `Route: A > B > C`, the names in order; then `Total load: T kg`.

Sample 1 (the checker uses this one)

Input

```
3
Base,5300,120
Camp 1,6100,40
Camp 2,6700,25
```

Output

```
Base: 5300 m, 120 kg
Camp 1: 6100 m, 40 kg
Camp 2: 6700 m, 25 kg
Route: Base > Camp 1 > Camp 2
Total load: 185 kg
```

Why: Each line splits into three pieces. $120 + 40 + 25$ is 185. Three names need two arrows; one name needs none.

Hint: `line.split(",")` gives three strings; `int()` the second and the third before they go into the tuple.

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 ⌨ In `example3.py` change the `5000` in `int(parts[1]) - 5000` to `4000`: a lower Base Camp. Run it again with the same three lines. What number is on the `K2` line?

Answer: .....

Key 4.2 ⌨ Type `"Camp 1,6100,40".split(",")` and look at the pieces. Then type `len("Camp 1,6100,40".split(","))`. What came back?

Answer: .....

Key 4.3 ⌨ Type `"40" + "25"` and look. Then type `len("40" + "25")`. What came back?

Answer: .....

Key 4.4 ⌨ Type `" > ".join(["Base"]) == "Base"`. What came back?

Answer: .....

Key 4.5 ✎ **By hand, Shell closed.** Key 4.3 glued two strings. What is `int("40") + int("25")`?

Answer: .....

4 Alias or Clone? (~14 min)

Ref: Lecture 12 — Aliasing and Cloning

Two names can point at one list. Then a change made through either name shows through both. `[:]` makes a second list with the same values, a clone. A function that must leave its argument alone works on the clone.

Worked example: two names, one route

The leader writes the route up as a list. A second name for the same list is an alias. A copy made with `[:]` is a clone. It is `example4.py`: run it; nothing is typed.

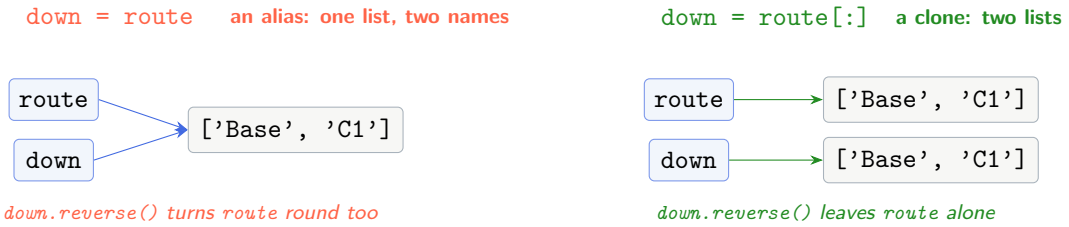
```
1 route = ["Base", "Camp 1"]
```

```
same = route
same.append("Camp 2")
print("route:", route)
plan = route[:]
plan.append("Summit")
print("route:", route)
print("plan:", plan)
print(plan is route)
```

Output

```
route: ['Base', 'Camp 1', 'Camp 2']
route: ['Base', 'Camp 1', 'Camp 2']
plan: ['Base', 'Camp 1', 'Camp 2', 'Summit']
False
```

Notice: line 3 appends through `same`, and `route` grows: one list, two names. Line 6 appends to the clone, and `route` stays as it was. `is` on line 9 asks whether two names point at one object. Key 5.1 takes the `[:]` off line 5.



TASK 5 The Way Down

EASY task05.py ~11 min

The route up is the route down: on the descent the same camps are passed in the opposite order. The leader keeps the list of the way up. The way down is a second list, and the first must not change.

Part 1, the function: `descent(route)` returns a **new** list: the camps of `route` in reverse order. It does not change `route`, and it prints nothing. In the Shell, `descent(['Base', 'Camp 1', 'Camp 2'])` gives `['Camp 2', 'Camp 1', 'Base']`.

Part 2, the program: `task05.py` holds the whole program already, with one gap: the body of `descent`, marked `# TODO`. Fill it with three lines: make a clone of `route`, reverse the clone, return it.

Input: Line 1: n, the number of camps, 1 or more. Then n lines: one camp name each, lowest first.

Output: `Up:` and the list, as `print("Up:", route)` prints it; `Down:` and the returned list, the same way; `Same list:` and what `route is down` gives.

Sample 1 (the checker uses this one)

Input

```
3
Base
Camp 1
Camp 2
```

Output

```
Up: ['Base', 'Camp 1', 'Camp 2']
Down: ['Camp 2', 'Camp 1', 'Base']
Same list: False
```

Why: The way down starts at the top camp. The two names point at two lists, so `is` gives False.

Hint: `reverse()` turns a list round in place and hands back `None`, as Key 5.4 showed. So reverse the clone on its own line, then return the clone's name.

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 Run `example4.py`. Then change `plan = route[:]` to `plan = route` and run it again. What does the last line show now?

Answer: .....

Key 5.2 Type `a = [3, 1, 2]`, then `b = a`, then `b.reverse()`, then `a[0]`. What came back?

Answer: .....

Key 5.3 Type `c = a[:]`, then `c is a`. What came back?

Answer: .....

Key 5.4 Type `print([3, 1].reverse())`. What printed?

Answer: .....

Key 5.5 **By hand, Shell closed.** `up = [1, 2, 3]`, then `down = up`, then `down.reverse()`. What is `up[0]` afterwards?

Answer: .....

5 Changing a List on Purpose (~36 min)

Ref: Lecture 12 — Aliasing and Cloning

Some functions are meant to change the list they are given. `pop(0)` takes the front item out and hands it back; `remove` takes out the first item that matches. A loop that removes from the list it is walking skips items, so it walks a clone.

Worked example: the fixed rope

At the fixed rope, climbers clip in one at a time, the front of the queue first. Whoever is unwell is taken out by name. It is `example5.py`: run it; nothing is typed.

```
1 queue = ["Ashraf", "Nazir", "Samina", "Sajid"]
2 first = queue.pop(0)
3 print(first, "clips in")
4 print("Waiting:", queue)
5 queue.remove("Samina")
6 print("Waiting:", queue)
7 for name in queue[:]:
8     queue.remove(name)
9 print("Waiting:", queue)
```

Output

```
Ashraf clips in
Waiting: ['Nazir', 'Samina', 'Sajid']
Waiting: ['Nazir', 'Sajid']
Waiting: []
```

Notice: `pop(0)` on line 2 takes the front name out and hands it back, so `first` keeps it. The loop on line 7 walks a clone, `queue[:]`, and removes from `queue`. A loop that removes from the list it walks skips names: Key 6.1 shows it.

TASK 6 Summit Push

MEDIUM `task06.py` ~16 min

⌨ On summit day the rope team at the top camp splits. The first k climbers on the rope go for the summit; the rest stay. If fewer than k are on the rope, everyone goes.

Part 1, the function: `summit_team(team, k)` takes the first k names out of the front of `team` and returns them as a new list, in order. `team` is shorter afterwards. With fewer than k names, it takes them all. It prints nothing. In the Shell, with `team = ['Ashraf', 'Nazir', 'Samina', 'Sajid']`, `summit_team(team, 2)` gives `['Ashraf', 'Nazir']`, and `team` is then `['Samina', 'Sajid']`.

Part 2, the program: Read the rope team and k. Print who goes up and who stays:

Step 1: Read n, k and the names into a list `team`. Print `Staying:` and `team`. Run it on the sample: you should see all four names.

Step 2: Above that, define `summit_team`: start `going` empty; while it holds fewer than k names and `team` is not empty, append `team.pop(0)` to it; return it. Run the file, then type `summit_team(['Ali', 'Hassan'], 5)` in the Shell: you should see `['Ali', 'Hassan']`.

Step 3: Call it before the print, keep its value, and print `Going up:` and that value first. Run it: you should see the sample.

Input: Line 1: n, the number of climbers, 1 or more. Line 2: k, how many go for the summit, 0 or more. Then n lines: one name each, front of the rope first.

Output: `Going up:` and the list the function returned, then `Staying:` and `team`, printed after the call.

Sample 1 (the checker uses this one)

Input

```
4
2
Ashraf
Nazir
Samina
Sajid
```

Output

```
Going up: ['Ashraf', 'Nazir']
Staying: ['Samina', 'Sajid']
```

Why: The first two on the rope go. The other two are what is left of the team.

Hint: the loop has two reasons to stop: k names taken, or nobody left. Key 6.4 is what happens without the second; Key 6.5 is the case that needs it.

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 `example5.py`. Then change `queue[:]` in the `for` line to `queue` and run it again. How many names are in the last `Waiting:` line now?

Answer: .....

Key 6.2 ⌨ Type `q = ['Ashraf', 'Nazir', 'Samina']`, then `x = q.pop(0)`, then `len(q)`. What came back?

Answer: .....

Key 6.3 ⌨ Now type `x == 'Ashraf'`. What came back?

Answer: .....

Key 6.4 ⌨ Type `[].pop(0)`. The last line of the error starts with an error's name. Which?

Answer: .....

Key 6.5 **By hand, Shell closed.** `team` holds three names and k is 5, so `summit_team(team, 5)` takes everyone. How many names are left in `team` afterwards?

Answer: .....

TASK 7 Two O'Clock Rule

MEDIUM `task07.py` ~16 min

On 10 May 1996 several teams were on the summit ridge of Everest. The guides' rule was the turn-back time: anyone not on the summit by 2 pm turns back. Many kept going past it, and the afternoon storm killed eight people. A log keeps each climber's summit time as an hour on the 24-hour clock: 14 is 2 pm.

Part 1, the function: `turn_back(times, deadline)` removes from `times` every hour later than `deadline`, and returns how many it removed. The list it is given changes; the hours kept stay in their order. It prints nothing. In the Shell, with `times = [11, 15, 16, 9]`, `turn_back(times, 14)` gives 2, and `times` is then `[11, 9]`.

Part 2, the program: Start from your `task03.py`: copy it into this file, then change it:

Step 1: Rename `above` to `turn_back` and change what it does: instead of building a new list, loop over a clone, `times[:]`, remove each late hour from `times`, and count. Return the count. Keep the count the call gives back, delete the two old print lines, and print `Turned back:` and the count. Run it on the sample: you should see `Turned back:  2`.

Step 2: Print `Summited:` and `times` after it. Run it: you should see the sample.

Input: Line 1: the deadline hour, 0 to 23. Line 2: n, the number of climbers, 0 or more. Then n lines: one summit hour each.

Output: `Turned back: C`, then `Summited:` and `times`, printed after the call.

Sample 1 (the checker uses this one)

Input

```
14
4
11
15
16
9
```

Output

```
Turned back: 2
Summited: [11, 9]
```

Why: 15 and 16 are later than 14, so two turn back. 14 itself would be in time. 11 and 9 stay, in their order.

Hint: never remove from the list you are looping over: Key 7.2 stepped past 16 and left it in, and Key 7.4 is the same loop over a clone. Loop over `times[:]`, remove from `times`, and count as you remove.

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 Type `L = [11, 15, 16, 9]`, then `L.remove(15)`, then `len(L)`. What came back?

Answer: .....

Key 7.2 Type `L = [15, 16, 9]`, then `for t in L: L.remove(t)` and press Enter twice, then `len(L)`. What came back?

Answer: .....

Key 7.3 Type `14 > 14`. What came back?

Answer: .....

Key 7.4 Type `L = [15, 16, 9]` again, then `for t in L[:]: L.remove(t)` and press Enter twice, then `len(L)`. What came back?

Answer: .....

Key 7.5 **By hand, Shell closed.** `times = [14, 15, 13, 17]`. What does `turn_back(times, 14)` return?

Answer: .....

6 Bonus: Lists Inside Lists (~41 min)

Optional. Not counted. From Lecture 12 — Aliasing and Cloning

A list can hold lists. Two indexes reach an item inside: `stores[2][0]` is the first item of the third inner list. A copy made with `[:]` is a new outer list whose inner lists are the old ones. A copy that shares nothing copies each inner list too. Task 8 first changes a plain list through a parameter.

Worked example: a store inside the store list

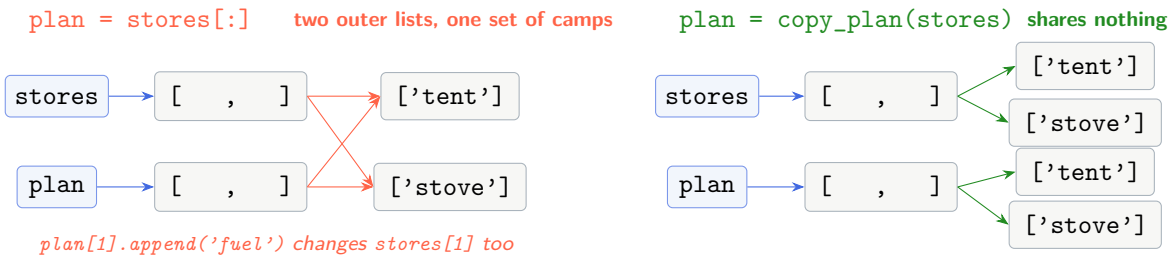
Each camp keeps a store, a list of gear. The mountain is a list of those lists. It is `example6.py`: run it; nothing is typed.

```
1 stores = [["tent", "rope"], ["stove"]]
2 print(stores[1][0])
3 stores[1].append("fuel")
4 print(stores)
5 plan = stores[:]
6 plan[0].append("flag")
7 print(stores[0])
8 print(plan is stores)
```

Output

```
stove
[['tent', 'rope'], ['stove', 'fuel']]
['tent', 'rope', 'flag']
False
```

Notice: two indexes reach inside on line 2: `stores[1]` is the second camp's list, and `[0]` its first item. Line 5 makes a new outer list, so `is` gives False. But `plan[0]` is the very list `stores[0]`, so the flag on line 6 shows in both. Key 8.1 changes the two indexes on line 2.



TASK 8 Sleep Limit

EASY `task08.py` ~11 min

Above about 7500 m the body cannot recover, so an expedition sets a sleep limit. Any planned camp above the limit is moved down to the limit.

Part 1, the function: `cap(camps, limit)` **changes** the list it is given: every height above `limit` becomes `limit`. It returns nothing (`None`) and prints nothing. In the Shell, with `c = [5300, 7800, 8000]`, after `cap(c, 7500)` the list `c` is `[5300, 7500, 7500]`.

Part 2, the program: Read the limit and the planned camps. Apply the limit, then print the plan and its highest camp.

Input: Line 1: the sleep limit in metres. Line 2: n, the number of camps, 1 or more. Then n lines: one planned height each.
Output: the list after the cap, printed with `print`, then `Highest camp: H`.

Sample 1 (the checker uses this one)

Input

```
7500
3
5300
7800
8000
```

Output

```
[5300, 7500, 7500]
Highest camp: 7500
```

Why: 7800 and 8000 are above the limit, so both become 7500. 5300 stays. A camp at exactly 7500 would stay too.

Hint: walk the indexes with `range(len(camps))` and assign into `camps[i]` where needed. Do not store what `cap` gives back: print `camps` itself after the call. `max` finds the highest.

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 `print(stores[1][0])` to `print(stores[0][1])` and run it again. What printed first?
Answer: .....

Key 8.2 Type `c = [5300, 7800]`, then `c[1] = 7500`, then `c[1]`. What came back?
Answer: .....

Key 8.3 Type `7500 > 7500`. What came back?
Answer: .....

Key 8.4 Type `max([5300, 7500, 6100])`. What came back?
Answer: .....

Key 8.5 **By hand, Shell closed.** `c = [7000, 7600, 7500]`, then `cap(c, 7500)`. How many items of `c` did the call change?
Answer: .....

TASK 9 Camp Stores

EASY task09.py ~12 min

⌨ Each camp has a store: a list of the kinds of gear kept there. The mountain is a list of those lists, lowest camp first: `[['tent', 'rope'], ['stove'], []]`. The top camp's store is empty until a load reaches it.

Part 1, the functions: `camps_with(stores, item)` returns how many camps have `item` in their store; a camp counts once, however many it has. `stock(stores, k, item)` adds `item` to the store of camp `k`, the list at index `k`, and returns nothing. Neither prints. In the Shell, with `s = [['tent', 'rope'], ['stove'], []]`, `camps_with(s, 'rope')` gives `1`. After `stock(s, 2, 'rope')`, the list `s[2]` is `['rope']`.

Part 2, the program: Read the stores and one item. Print how many camps have it. Stock the top camp with it. Print the top camp's store, then the count again.

Input: Line 1: n, the number of camps, 1 or more. Then n lines, lowest camp first: one camp's items, separated by single spaces; an empty line is an empty store. Last line: the item.

Output: `Camps with ITEM: C`; the top camp's store after the stocking, printed with `print`; `Camps with ITEM: C` again.

Sample 1 (the checker uses this one)

Input

```
3
tent rope
stove

rope
```

Output

```
Camps with rope: 1
['rope']
Camps with rope: 2
```

Why: The third camp's line is empty: its store is `[]`. Only the first camp has rope. After the stocking the top camp has it too.

Hint: `line.split()` with nothing in the brackets cuts at spaces. `stores[k]` is a list: append to it. `in` asks whether a list holds an item, so two ropes still count once.

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 ⌨ Type `s = [['tent', 'rope'], ['stove'], []]`, then `s[1][0] == 'stove'`. What came back?

Answer: .....

Key 9.2 ⌨ Type `len("".split())`: an empty line, split at spaces. What came back?

Answer: .....

Key 9.3 ⌨ Type `'rope' in s[0]`. What came back?

Answer: .....

Key 9.4 ⌨ Type `s[2].append('rope')`, then `len(s[2])`. What came back?

Answer: .....

Key 9.5 ✎ **By hand, Shell closed.** Three camps: the first store has rope twice, the second has it once, the third is empty. What does `camps_with(s, 'rope')` return?

Answer: .....

TASK 10 The Plan on Paper

MEDIUM task10.py ~14 min

Before a load is carried, the leader tries the plan on paper: a copy of every camp's store, to change without touching the real stores. A copy made with `[:]` is not enough. It is a new outer list, but its camps are the very same inner lists.

Part 1, the function: `copy_plan(stores)` returns a copy of `stores` that shares nothing with it: a new outer list, holding a new copy of every camp's list. A change to a camp of the copy leaves the real stores as they were. Build it with a loop and `[:]`; the `copy` module is not allowed in this task. It prints nothing. In the Shell, with `s = [['tent'], []]` and `p = copy_plan(s)`, after `p[1].append('fuel')` the list `s` is still `[['tent'], []]`.

Part 2, the program: Read the stores. Make the plan, add `flag` to the top camp of the plan only, then print both top camps:

Step 1: Read the stores as in Task 9. Define `copy_plan`: start a new empty list, append `store[:]` for each camp, return it. Call it, then print `Real:` and the real top camp and `Plan:` and the plan's. Run it on the sample: both lines show `['stove']`.

Step 2: Between the call and the prints, append `flag` to the plan's top camp. Run it: you should see the sample.

Input: Line 1: n, the number of camps, 1 or more. Then n lines, lowest camp first: one camp's items, separated by single spaces; an empty line is an empty store.

Output: `Real:` and the real top camp's store, then `Plan:` and the plan's.

Sample 1 (the checker uses this one)

Input

```
2
tent rope
stove
```

Output

```
Real: ['stove']
Plan: ['stove', 'flag']
```

Why: The flag went into the plan's copy of the top camp. The real top camp did not change.

Hint: append `camp[:]`, not `camp`. With `camp`, the plan's top camp and the real one are one list, and the flag shows in both: Key 10.3. Key 10.4 is the line that fixes it.

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 `task10.py` opens.

Key 10.1 Type `s = [['tent'], []]`, then `p = s[:]`, then `p is s`. What came back?
Answer: .....

Key 10.2 Type `p[0] is s[0]`. What came back?
Answer: .....

Key 10.3 Type `p[1].append('fuel')`, then `len(s[1])`. What came back?
Answer: .....

Key 10.4 Type `q = []`, then `q.append(s[0][:])`, then `q[0] is s[0]`. What came back?
Answer: .....

Key 10.5 **By hand, Shell closed.** `s = [[1], [2]]`, then `p = copy_plan(s)`, then `p[0].append(9)`. What is `len(s[0])` afterwards?
Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

☐ I can return two values as one tuple, and unpack it into two names.

☐ I can build a list from input with `append`, and I know `append` hands back `None`.

☐ I can take the first k items with a slice, and I know the slice stops before its stop.

☐ I can shrink a failing input until the bug is easy to see.

☐ I can build a new list out of an old one with a loop and an `if`.

☐ I can split a line into pieces, turn the right ones into numbers, and join strings back together.

☐ I can change one line of a working program, predict what the run will show, and check it.

☐ I can tell an alias from a clone, and `==` from `is`.

☐ I can make a function leave the caller's list alone by working on a clone.

☐ I can take items out of a list with `pop(0)` and `remove`, and keep what `pop` hands back.

☐ I never remove from a list while looping over it: I loop over a clone.

☐ I can reach into a list of lists with two indexes, and copy one so that it shares nothing.