

Lists in Practice

Part A is six problems on list operations, slicing and copying, walking into three classic traps deliberately. Part B takes one class roster from parsing to ranking to dirty data, and ends by showing where lists run out. Part C is git on your own machine.

Due: Homework 2 · Tue 27 Oct, 23:59

What this session is for

- Get comfortable with list operations and slicing
- Reproduce and then fix three classic traps: `[[0]*n]*m`, shallow copies, and deleting while iterating
- Build the habit of settling an argument with `id()` instead of guessing
- Handle a realistically shaped dataset with nested lists, and find out where they stop working
- Start recording your own changes with git

AI policy

This week is **Level 0**: no AI-generated or AI-completed code. Turn off AI completion in your editor. When you get stuck, debug it yourself first, then ask a classmate or a TA.

Keep annotating types

Type annotations became a requirement last week. Lists are written `list[int]`, `list[str]`; a nested list is `list[list[int]]`. Part B has records with an awkward shape, and writing the type down is mostly for your own benefit.

5-1 Class statistics (warm-up)

Given a list of marks, print the count, the mean, the highest, the lowest and the pass rate. Round the mean to two decimal places.

```
input:  [87, 92, 45, 78, 55, 95]
output:
count 6
mean 75.33
high 95
low 45
pass rate 66.67%
```

5-2 Write your own max

Without using `max()`, implement `my_max(nums: list[int]) -> int | None` returning the largest value. Return `None` for an empty list.

Think before you type

What do you initialize the running maximum to? Zero seems natural — but what if every value is negative?

5-3 Deduplicate, order preserved

Implement `dedup(nums: list[int]) -> list[int]`, which drops repeats while keeping the original order.

```
input:  [3, 1, 3, 5, 1, 7]
output: [3, 1, 5, 7]
```

`set()` is not allowed here — it throws the ordering away.

5-4 Transpose a matrix

Implement `transpose(matrix: list[list[int]]) -> list[list[int]]`.

```
input:  [[1, 2, 3], [4, 5, 6]]
output: [[1, 4], [2, 5], [3, 6]]
```

5-5 Find the bug

The code below is supposed to double every even number, giving `[1, 4, 3, 8, 5, 12]`, but the result is wrong. Run it first and see what you actually get. Then work out why, write a corrected version, and explain the cause in a sentence or two in your submission notes.

```
a = [1, 2, 3, 4, 5, 6]
for i in range(len(a)):
    if a[i] % 2 == 0:
        a.insert(i, a[i] * 2)
```

5-6 A copying experiment

Write code that uses `id()` to demonstrate all three of the following, and paste the output into your submission notes:

1. After `b = a`, `a` and `b` are the same object
2. After `b = a[:]`, `a` and `b` are different objects
3. But for a nested list, after `b = a[:]`, `a[0]` and `b[0]` are **still the same object**

Part B — going deeper: one class roster, four times over

The six problems in Part A have nothing to do with each other. This part has a single problem — **a class roster** — but you work it four times: read it in, rank it, survive the dirty version, and then find the scale at which the whole approach falls apart.

The data is thirty lines of text. Each line is `student-id name quiz1 quiz2 quiz3`:

```
RAW = [  
    "2026001 zhangsan 91 57 59",  
    "2026002 lisi 70 81 55",  
    "2026003 wangwu 43 65 66",  
    "2026004 zhaoliu - 78 90",  
    "2026005 qianqi 90 90 61",  
    "2026006 sunba 73 91 73",  
    "2026007 zhoujiu 60 53 75",  
    "2026008 wushi 105 60 70",  
    "2026008 wushi 60 70 80",  
    "2026009 zhengyi 60 61 69",  
    "2026010 chener 57 48 70",  
    "2026011 chusan 59 99 46",  
    "2026002 lisi 10 10 10",  
    "2026012 weisi 43 59 89",  
    "2026013 jiangwu 95 96 48",  
    "2026014 shenliu 53 73 62",  
    "2026015 hanqi 64 73 96",  
    "2026016 yangba 59 71 85",  
    "2026017 zhujiu 64 41",  
    "2026018 qinshi 59 81 93",  
    "2026011 chusan 88 88 88",  
    "2026019 youyi 97 73 62",  
    " 2026020 xuer 70 80 90  ",  
    "2026021 hesan 82 90 80",  
    "2026022 lvsi 61 55 60",  
    "2026023 shiwu 58 x 88",  
    "2026024 zhangliu 63 93 95",  
    "2026019 youyi 40 40 40",  
    "2026025 kongqi 41 100 67",  
    "2026026 caoba 99 50 48",  
]
```

Several kinds of mess are hiding in those thirty lines; B-3 deals with them. B-1 and B-2 may assume every line is clean.

No file reading — that is L12. Copy the block above into your own file.

Why the names have no space in them

Because the line is split on **whitespace**. One space inside a name changes the field count and every index after it is wrong. In any real data format the first question to ask is whether the separator can appear inside the data itself — it is exactly why CSV has to quote a field containing a comma.

B-1 Read it in

Implement `parse_roster(lines: list[str]) -> list[list]`, turning each line into one record:

```
["2026001", "zhangsan", [87, 92, 45]]
```

Keep the student ID as a string — you never do arithmetic on it. The three quiz marks go in a list of their own.

```
in: ["2026001 zhangsan 87 92 45", "2026002 lisi 78 65 90"]
out: [["2026001", "zhangsan", [87, 92, 45]],
      ["2026002", "lisi", [78, 65, 90]]]
```

Use `split()` ; do not count spaces

`"2026001 zhangsan 87 92 45".split()` hands you the fields directly, and it does not care how many spaces separate them. That was L3. Remember that `split()` returns strings — the marks need `int()`.

B-2 Rank them

Implement `rank(records: list[list]) -> list[list]`, sorting by total mark **descending**, breaking ties by student ID **ascending**:

```
[[1, "2026003", "wangwu", 274],  
 [2, "2026002", "lisi", 233],  
 [3, "2026001", "zhangsan", 224]]
```

Positions are numbered 1, 2, 3, ... with no special handling for ties.

`sort()` does not hand you a list

```
ranked = records.sort()    # wrong: sort() returns None
```

Either call `records.sort()` and then use `records`, or use `sorted(records)` for a new list. The lecture said this. It is still the most common way to get this problem wrong.

How lists compare to each other

Python compares two lists **element by element**, stopping as soon as one differs. So if you put your sort keys at the front of the list in priority order, plain `sort()` with no arguments does the right thing. For “total descending, ID ascending”, try sorting on `[-total, student_id]`.

B-3 The dirty version

Real data is not that tidy. The full `RAW` contains all of these:

Case	Where it is in RAW	What to do
absent	2026004 zhaoliu - 78 90	- counts as 0; keep the record
wrong number of fields	2026017 zhujia 64 41 (one quiz short)	drop the whole record
invalid mark	2026008 wushi 105 60 70 , 2026023 shiwu 58 x 88	not a whole number in 0..100: drop the record
duplicate ID	2026002 , 2026011 and 2026019 each appear twice	keep the first one
extra whitespace	2026020 xuer 70 80 90	keep it; it is fine

Implement `clean(lines: list[str]) -> list[list]`, returning cleaned records in the same shape `parse_roster` produces. **Apply the checks in the order of the table:** field count, then mark validity, and de-duplicate **last**, among the records that survived.

In your **submission notes**, say how many records you dropped and why.

Deciding whether a string is a number

`"105".isdigit()` is `True`; `"x".isdigit()` is `False`. You need something like it to ask the question *before* calling `int()` — `int("x")` raises, and your program stops there.

It is a string method, and L6 goes through them properly; for now you need only this one. Note that `"-5".isdigit()` is also `False`, since the minus sign is not a digit — which happens to be exactly what this problem wants.

That order is not arbitrary

The problem fixes it as “validate, then de-duplicate”. The other way round misbehaves:

```
2026008 wushi 105 60 70      ← mark out of range
2026008 wushi  60 70 80      ← same ID, and this one is fine
```

Those two lines are in `RAW`, one after the other.

De-duplicate first and the bad line claims the ID, so the good one is discarded as a duplicate — **you have thrown away the good data and kept the bad**, which then gets dropped for being out of range, and the student vanishes entirely.

In your notes, give one more example of your own where the two orders disagree. Most real arguments about data cleaning are arguments about exactly this.

B-4 Where this approach gives out

Now answer some queries, using **nothing but lists**:

1. Write `find(records, sid)`, locating one record by student ID.
2. Add a counter to `find` recording how many comparisons it took. Look up the 1st, the 15th and the 30th student, and note the count for each.
3. Write `has_duplicate(records)` — no `set` — and count its comparisons too.

Then answer:

1. How does `find`'s comparison count relate to the number of records?
2. How does `has_duplicate`'s? What makes it worse than `find`?
3. This class has 75 students and the approach is entirely adequate. What happens with a university-wide table of thirty thousand, queried tens of thousands of times a day?
4. What would you want a container to do, so that “find one record by ID” did not mean comparing against every record in turn?

Question 4 is next week's lecture

L6 introduces dictionaries. A dictionary lookup barely notices how much data there is — ten thousand records and a million records take about the same time. The price is that its keys must be immutable. **Hold on to the discomfort from this section**; next week you will know what dictionaries are for.

Part C — fundamentals: git on your own machine

Your code now has some weight to it: break one of those Part B functions and you may not get back. That is the problem git solves — **it remembers every version for you, so you can always go back to one.**

This week is local only. Pushing to GitHub is next week.

Tell git who you are, once

```
git config --global user.name "Your Name"  
git config --global user.email "you@example.com"
```

Every commit records who made it, so git has to be told. **You do this once, ever.**

Six commands

Command	What it does
<code>git init</code>	make the current directory a repository. Once, at the start of a project
<code>git status</code>	what has changed, and what is staged. Type it whenever you are unsure
<code>git add file</code>	put changes in the staging area : "these are the ones I mean to commit"
<code>git commit -m "message"</code>	record everything staged as one version
<code>git log --oneline</code>	the history, one line per commit
<code>git diff</code>	changes not yet staged ; <code>git diff --staged</code> for the staged ones

`commit` without `-m` drops you into an editor

Leave off `-m` and git opens an editor for the commit message. Plenty of people meet this for the first time and conclude the terminal has hung. If you land in one: it is probably vim, so press `Esc`, type `:q!` and hit Enter; if it is nano, `Ctrl+X`. Then run it again with `-m`.

`git add .` is not as innocent as it looks

`.` means everything under the current directory — `__pycache__`/, `.DS_Store`, that 300 MB data file — and **once something has been committed it is genuinely hard to purge**. Run `git status` first and decide deliberately.

To exclude things permanently, put a `.gitignore` in the project root, one pattern per line:

```
__pycache__/  
.DS_Store  
*.pyc
```

A commit is not a backup

Committing stores a version in a repository **on this machine**. When the disk dies it all goes with it. Next week you push to GitHub; that is the backup.

C-1 Walk through it

Work in this week's lab directory, and record each command and its output in your submission notes:

1. `git init`, then `git status` — what does it tell you?
2. Create a `.gitignore` with at least `__pycache__`/ in it
3. `git add` your lab code, run `git status` again, and see what changed in the output
4. `git commit -m "lab05 part A"` for your first version
5. Change a function, then `git diff`. **Work out what the leading `-` and `+` mean**
6. `git add` it and run `git diff` again — why is there nothing now? Try `git diff --staged`
7. Commit the second version and look at `git log --oneline`

C-2 Think it through

1. If everything ends up committed anyway, why does git bother with a staging area? When would you want to commit only some of your changes?
 2. What belongs in a commit message? Compare these two and ask which helps you more three months from now: `"fixed stuff"` versus `"B-2: sort on [-total, id] so ties are no longer order-dependent"`
-

Submitting

The six Part A problems are marked independently on the platform, and so are Part B's `parse_roster`, `rank` and `clean`. The answers for B-4 and Part C go in your **submission notes**.

How the marking works

The platform calls your function with a set of test cases and compares what comes back. So: **your function must return its result, not print it** — unless the problem explicitly asks for printed output. This was covered in L3, and it is the most common way people lose marks here.

Before you submit

- ☐ Every function has type annotations
- ☐ 5-5's bug is explained
- ☐ 5-6's three `id()` outputs are pasted in
- ☐ B-3 says how many records were dropped and why, with your own example of the orders disagreeing
- ☐ All four B-4 questions are answered
- ☐ Part C's command output and both questions are written up