

Word Counting, Container Choice and Git Remotes

Part A combines dictionaries, sets and strings in six function tasks and one design question. Part B compares list and set lookup costs. Part C continues last week's Git work with a private remote repository.

Due: Homework 2 · Tue 27 Oct, 23:59

Goals

- Use dictionaries for mappings, groups and counts
- Let a list preserve order while a set handles membership
- Split text into fields, trim it and convert the types
- Support a container choice with counts and measurements
- Distinguish a local Git commit from a remote push

Last week a list held a batch of values. Now decide what each task actually needs: order, a mapping, or membership alone. Apply those choices in Part A, compare their costs in Part B, then synchronize your own code with a private remote repository.

AI policy

This is still **Level 0**: do not use AI to generate or complete code. Run a small example and inspect intermediate values before asking classmates or teaching assistants for help.

Types and submissions

Keep the six function names and parameters exactly as specified, and annotate every function. Submit code under the matching problem in **Homework 2. Return the result**; the function tasks do not ask you to print it. Problem 6-7 and Parts B and C have separate answer cards. Do not mix those responses into the code files.

Container annotations

Annotation	Meaning
<code>list[int]</code>	a list of integers
<code>set[str]</code>	a set of strings
<code>tuple[str, int]</code>	a two-item tuple: string first, integer second
<code>dict[str, list[str]]</code>	strings mapped to lists of strings
<code>dict[str, dict[str, int]]</code>	student names mapped to subject-to-score dictionaries

Describe both the outer and inner types. An annotation does not check data at runtime; the implementation still has to follow the input contract.

Part A — basic operations in combination

6-1 Invert a dictionary

Implement `invert(d: dict[str, int]) -> dict[int, str]`, returning a **new dictionary** with keys and values swapped. Leave the input dictionary unchanged.

```
invert({"a": 1, "b": 2}) → {1: "a", 2: "b"}
invert({}) → {}
```

Process the input in its iteration order. If several entries produce the same result key, a later assignment replaces the earlier value.

In this problem's submission notes, answer: **if two original keys share one value, what does inversion do?** Give a concrete input and output, and explain what information cannot be kept.

6-2 Deduplicate while keeping order

Implement `unique_ordered(items: list[int]) -> list[int]`, removing repeated integers while preserving the **order of first appearance**. Return a list.

```
unique_ordered([3, 1, 3, 5, 1, 7]) -> [3, 1, 5, 7]
unique_ordered([9, 1, 9, 0])       -> [9, 1, 0]
unique_ordered([])                 -> []
```

Use a set for the membership test. Searching the list of retained items with `in` is not allowed. Converting to a set and then sorting is not a substitute: it loses first-appearance order.

Give each container one job

The set answers "have I seen this value?" The list stores retained values in the required order. A set need not do both.

Explain in this problem's notes why a set fits repeated membership checks. Average fast lookup does not mean exactly one comparison every time. Part B investigates the distinction.

6-3 Names in common

Implement `common(a: list[str], b: list[str]) -> list[str]`. Return names present in both inputs, with **each name appearing once**, sorted by Python's default string order. Return a list, not a set.

```
common(["Alex", "Blake", "Chen"], ["Blake", "Chen", "Dana"])
→ ["Blake", "Chen"]
common(["c", "a", "a"], ["a", "c", "c"])
→ ["a", "c"]
common([], ["a"]) → []
```

Use the default order of `sorted()`. Chinese names follow the same string-comparison rules; do not sort by pronunciation or stroke count. Names differing in case remain distinct.

Membership first, order second

The common members form an intersection, but the requested result is an ordered list.

6-4 Group by first character

Implement `group_by_first(words: list[str]) -> dict[str, list[str]]`. Each word's first character is its group key. Preserve input order **within each group**, including repeated words.

The input list may be empty. Every word in it is guaranteed to be **nonempty**. Do not normalize case.

```
group_by_first(["banana", "apple", "avocado", "apple"])
→ {"b": ["banana"], "a": ["apple", "avocado", "apple"]}
group_by_first(["a", "b", "a"])
→ {"a": ["a", "a"], "b": ["b"]}
group_by_first([]) → {}
```

Create a group on its first occurrence

`groups[w[0]].append(w)` raises `KeyError` when that key has not been inserted. Check for the key first, or use `setdefault()` to obtain the stored list and insert a new one if needed. Each group needs its own list. `dict.fromkeys(keys, [])` shares one list across all keys.

6-5 Parse a transcript

Implement `parse(lines: list[str]) -> dict[str, dict[str, int]]`. Parse each `name,subject,score` line into `{name: {subject: score}}`.

Each line is guaranteed to contain exactly three comma-separated fields. Names and subjects are nonempty and contain no commas; scores convert to integers in 0–100. **Any field may have surrounding whitespace**, which must be trimmed separately. Missing fields and invalid scores are outside this task's input domain.

Keep all of a student's different subjects. If the same name and subject occur again, **the later score wins**. Empty input gives an empty dictionary.

```
parse(["Alex,Math,87", "Alex,Physics,92", "Blake,Math,65"])
→ {"Alex": {"Math": 87, "Physics": 92}, "Blake": {"Math": 65}}
parse([" Alex , Math , 60 ", "Alex,Math,90"])
→ {"Alex": {"Math": 90}}
parse([]) → {}
```

Check the stages separately

`split(",")` separates fields, `strip()` trims their edges, and `int()` converts the score. A student's first line creates the inner dictionary. Later subjects must update it rather than replace it.

6-6 Word frequencies

Implement `top_words(text: str, n: int) -> list[tuple[str, int]]`, returning up to `n` frequent words and their counts. Each item is a `(word, count)` tuple.

Use these deliberately simple token rules:

1. Split on runs of whitespace with `split()` and no argument.
2. Ignore case.
3. Strip punctuation from token edges, handling at least `.,!?:;`. Internal punctuation is not a separator.
4. Skip tokens that become empty after cleaning; do not count an empty word.
5. Sort by decreasing frequency; break ties by increasing default string order.

`n` is a non-negative integer. `n = 0` gives an empty list. If `n` exceeds the number of distinct words, return them all without padding. Empty text, or text containing only the specified punctuation and whitespace, gives an empty list.

```
top_words("The cat. The DOG! the cat", 2)
→ [("the", 3), ("cat", 2)]
top_words("dog cat dog cat", 2)
→ [("cat", 2), ("dog", 2)]
top_words("... cat! CAT ;", 10)
→ [("cat", 2)]
top_words("cat", 0) → []
top_words("", 3)   → []
```

Inspect each stage before combining it

Check tokenization, cleaning, counting and ordering separately. Looking only at the final result makes it harder to locate a mistake. A named key function can return `(-count, word)`: compare the negative count first, then the word. `Counter.most_common()` breaks ties by first appearance, **not by the order required here**.

6-7 Choose a container (no code)

Use the **L06 · Container choice** answer card. For each scenario, name a container and justify it. A combination is fine if you explain what each part does:

1. Map the student numbers of 116 students to their names.
2. Keep every score from one exam, then compute the mean and maximum.
3. Repeatedly check whether a word belongs to a vocabulary of ten thousand common words.
4. Represent a point in a plane, to be used as a dictionary key.
5. Store the courses taken by a class, with no repeated course.
6. Preserve a student's successive submissions in arrival order.

Justify the actual operations

"Dictionaries are fast" is too vague. Say what will be looked up, whether order matters, whether duplicates are allowed, and whether entries must be added or removed. A tuple key also requires hashable elements; saying only "tuples are immutable" is not enough.

Part B — deeper work: list and set lookup costs

Lists and sets cooperate in Part A. Now compare them on **the same data and the same queries**, rather than declaring a winner from one timing.

B-1 Count comparisons first

This linear search returns a found flag and the number of element comparisons:

```
def contains_counted(items: list[int], target: int) -> tuple[bool, int]:
    checks = 0
    for x in items:
        checks += 1
        if x == target:
            return True, checks
    return False, checks
```

Use $n = 1000, 2000$ and 4000 , with `items = list(range(n))`. Query `0`, `$n // 2$` , `$n - 1$` and `$n$` ; record both results and comparison counts.

Paste the table into Part B's answer card. Explain how the target's position or absence changes the work, and why querying only the first item is not a fair assessment of list lookup.

B-2 Query the same values in a set

Construct `members = set(items)` once, then check the same four targets with `target in members`. Confirm that the found flags agree with B-1.

Answer:

1. Why can a set usually avoid comparing every element from the front?
2. Why is "exactly one comparison on every lookup" still not a valid claim?
3. How many input elements does constructing the set have to process? Is converting always worth it for one query?

Do not invent a count for the set's internal comparisons. B-1 counts a loop you wrote, not the interpreter's implementation of a hash table.

B-3 Separate preparation from query time

Measure with L5's `time.perf_counter()`. For example:

```
import time

start = time.perf_counter()
members = set(items)
build_seconds = time.perf_counter() - start
print("set construction:", build_seconds)
```

Time set construction separately from executing all queries. B-1's function is for counting only: time `target in items` and `target in members`, with the same query sequence, loop structure and repetition count. Do not print inside the timed section.

Run each measurement at least three times and record the values. If one query is too brief to measure reliably, repeat the same query sequence for multiple rounds; use the same number of rounds for both containers.

Explain which pattern the counts establish and what the timings add. A speed ratio measured once is not a universal constant for every machine and input.

B-4 Choose for a workload

Choose and justify a representation for each case, using B-1 through B-3:

- There are only a few items and the program makes one membership query.
- There are many items and many repeated membership queries, but the original order must also be retained.

The second may keep both a list and a set, at a memory cost. Explain the convenience gained and the cost paid. L11 will express these relationships with complexity notation.

Part C — fundamentals: Git remote synchronization

Lab 5's commits kept versions on this computer. A remote repository keeps another copy of the history, useful for backup and continuing in another working directory. **A commit does not upload anything; push sends commits to a remote.**

Six operations

Command	Effect
<code>git branch -M main</code>	rename the current branch to main when setting up its remote counterpart
<code>git remote add origin URL</code>	record a remote address as origin; no files are uploaded
<code>git remote -v</code>	inspect configured remote addresses
<code>git push -u origin main</code>	send main's commits and remember its upstream branch
<code>git clone URL new-directory</code>	copy the history and start a new working directory
<code>git pull --ff-only</code>	obtain remote commits; stop if a direct fast-forward is impossible

Private repositories and authentication

Create an **empty private repository** on GitHub. Do not initialize it with a README, license or `.gitignore`, which would create a separate history before your first push. Use the local repository and commits from Lab 5.

- Do not publish homework code in a public repository or upload a classmate's code.
- Follow GitHub's authentication prompts. HTTPS Git authentication does not accept the account's login password; use a credential manager or personal access token as appropriate.
- **Never put tokens, passwords or private keys in source files, remote URLs, submission notes or screenshots.**
- If GitHub is unreachable or authentication fails, ask a teaching assistant rather than trying a force push. The local alternative below exercises the same operations.

C-1 Push and obtain a new version

Record each command and its key output in Part C's answer card. Do not capture the authentication flow.

1. Run `git status` in the original lab directory. Identify what has already been committed. Commit only your own lab work and check that `.gitignore` excludes caches and files that must not be uploaded.
2. Name the current branch `main`, add your private repository as origin, and inspect it with `git remote -v`. If origin already exists, inspect it first rather than adding another or replacing it blindly.
3. Run `git push -u origin main`. Check that your commit appears in the remote repository.
4. Outside the original lab directory, clone into a new directory. Compare `git log --oneline` in the two copies.
5. In the original copy, add one line of lab notes to the README, commit it and push.
6. In the new copy, confirm that there are no uncommitted changes, then run `git pull --ff-only`. Check that the new commit and README line arrived.

Do not fix an error by forcing an overwrite

This exercise needs no `--force`, `reset --hard` or deletion of Git history. If `--ff-only` stops, inspect both histories with a teaching assistant before deciding what to do.

C-2 A local alternative when the network is unavailable

A Git remote may also be another directory on this computer. Create a separate, empty directory and run:

```
git init --bare --initial-branch=main
```

A bare repository stores history without working files; here it acts as the remote. Setting its initial branch to `main` prevents clone from looking for a nonexistent master branch after `main`

has been pushed. Use its absolute path as origin, then perform C-1's push, clone, second commit and pull. Keep each clone outside the original repository; do not nest repositories.

This verifies synchronization but remains on the same computer, so **it cannot protect against losing that computer's disk**. State which route you used in the answer card.

C-3 Explain the distinctions

Answer in the card:

1. What changes when you edit a file, add, commit and push? Why is a local commit not yet a remote backup?
2. Why does a clone already contain earlier commits? What does pull do?
3. Why use a private repository for this exercise, and why keep authentication material out of submission notes?

Submitting

Use [Homework 2](#). Submit the six function tasks separately. Their existing tests and automatic points remain unchanged: 4 points each, **24 in total**.

- Write the explanations for 6-1 and 6-2 in the notes dialog shown when submitting those tasks.
- Answer 6-7 in **L06 • Container choice**.
- Put Part B's tables, observations and explanations in **L06 • List and set lookup costs**.
- Put Part C's commands and explanations in **L06 • Git remote synchronization**, without authentication material.

The text questions in these three cards are reviewed by teaching assistants, not graded automatically. Reopen a submitted card to check that your responses were retained.

Before submitting

- ☐ All six functions have annotations and the specified return types

- ☐ 6-1 leaves the input unchanged and explains duplicate values with an example
- ☐ 6-2 retains first-appearance order and uses a set for membership
- ☐ 6-3 returns a sorted list with no repeated name
- ☐ 6-4 retains order and repetitions within each group and handles its first word
- ☐ 6-5 trims every field, produces integer scores and keeps a later repeated score
- ☐ 6-6 handles empty text, punctuation-only tokens, frequency ties and n above the vocabulary size
- ☐ Each of 6-7's six choices has a reason tied to the requirement
- ☐ Part B distinguishes counts from timings and invents no internal set counts
- ☐ Part C completes remote synchronization or its local substitute without exposing secrets