

Lecture 7 - Practice Exercises

Lists in realistic small programs

Guided Exercise - Event Registration List | Problem

- A small workshop stores the names of its registered participants in the same order in which registrations were received. During the registration period, the list may change: new participants can register, some participants may cancel, and occasionally a priority participant must be inserted at a specific position.
- The organizer therefore needs a simple program that can inspect and update the registration list while preserving the meaning of its order.
 - The program starts from an initial list of names and performs a sequence of common list operations.
 - At the end, the organizer should be able to see both the current registration order and an alphabetically sorted view of the participants without losing the original order.

```
participants = ["Ana", "Bruno", "Carla", "Diogo"]
```

Guided Exercise - Event Registration List | Requirements

- The program must:
 - display all participants with their positions, starting from 1;
 - ask for a name and check whether it is already registered;
 - if absent, add the new participant at the end;
 - ask for a priority participant and a target position;
 - insert the participant only if the position is valid.
- Then it must:
 - ask for a cancellation and remove the name only if present;
 - display the first and last registered participant when the list is non-empty;
 - create an alphabetically sorted version without modifying registration order;
 - display both the alphabetical list and the final registration order.

Guided Exercise - Event Registration List | Implementation Tasks

- Work almost step by step:
 - 1. Use `enumerate(..., start=1)` to print positions.
 - 2. Use **membership** before deciding whether to append.
 - 3. **Validate** the position text before converting it to `int`.
 - 4. Convert the human position to a list index.
- Continue with:
 - 5. Check membership before `remove()`.
 - 6. Check that the list is not empty before `[0]` and `[-1]`.
 - 7. Use `sorted()` to create a **new alphabetical** list.
 - 8. Print both lists and verify that sorting did not change registration order.

Laboratory 1 - Shopping Basket Analysis | Problem

- A customer enters the prices of all products currently contained in a shopping basket. Instead of entering one value at a time, all **prices** are provided **on a single line**, separated by spaces.
- The program must first transform this textual input into a **list of numeric prices**. It should then analyze the basket and produce a short **summary** that helps the customer understand the **total cost**, the observed **price range**, and how individual prices compare with the average.
- The customer also wants to identify products **below a chosen threshold** and inspect the prices in **increasing order** without losing the original order in which they were entered.

Input example: 12.50 4.90 25.00 8.30 16.40
Processing: string → split() → list of strings → list of floats

Laboratory 1 - Shopping Basket Analysis | Requirements

- Read and prepare the data:
 - read one line of space-separated prices;
 - convert every value to float and store it in a list;
 - if the list is empty, report that no prices were entered.
- Compute:
 - number of products and total cost;
 - average price;
- Also compute and display:
 - compute minimum and maximum price and print price range [minimum, maximum];
 - number of prices strictly above the average;
 - prices below a threshold entered by the user;
 - a sorted copy from lowest to highest;
 - the original list to prove it was not modified.

Laboratory 1 - Shopping Basket Analysis | Implementation Tasks

- Suggested workflow:
 - 1. `split()` the input line.
 - 2. Build prices with a loop and `append(float(...))`.
 - 3. Use `len()`, `sum()`, `min()` and `max()`.
 - 4. Compute average as *total / number of products*.
- Then:
 - 5. Count values above average with a loop and condition.
 - 6. Build the below-threshold list with a loop or comprehension.
 - 7. Use `sorted()` for the ordered copy.
 - 8. Format monetary values clearly with f-strings.

Laboratory 2 - Tag List Cleaner | Problem

- A website allows users to associate **textual tags** with a post. Tags are entered manually on a single line and separated by commas. Because the input is written by users, it may contain inconsistent capitalization, unnecessary spaces, repeated tags, or empty entries.
- Before storing the tags, the website wants to **normalize the input**. Tags that differ only in capitalization must be treated as the same tag, empty entries must be ignored, and duplicates must be removed while preserving the order in which each tag first appeared.
- The final result should be a **clean list** that can also be displayed **alphabetically** and converted back into a **formatted string**.

```
Input:  Python, data, loops, python, , Strings  
Clean: ["python", "data", "loops", "strings"]  
Final: python | data | loops | strings
```

Laboratory 2 - Tag List Cleaner | Requirements

- Starting from the complete input string:
 - split it using commas as separators;
 - strip spaces around each item;
 - convert tags to lowercase;
 - ignore empty entries.
- Build and display:
 - a clean list containing every valid tag once, preserving first-occurrence order;
 - the duplicated tags found, if any;
 - an alphabetically sorted version of the clean list;
 - a normalized string using " | " as separator.

Laboratory 2 - Tag List Cleaner | Implementation Tasks

- Suggested workflow:
 - 1. `raw_tags = line.split(",")`
 - 2. For every item, compute `strip().lower()`
 - 3. Skip the item when the result is an empty string
 - 4. Use membership to decide whether to append to `clean_tags`.
- Then:
 - 5. Record duplicated tags without repeating them in the duplicate list.
 - 6. Use `sorted()` for the alphabetical view.
 - 7. Use `" | ".join(...)` for the final normalized text.
 - 8. Test with capitalization, extra spaces, empty items and duplicates.

Independent 1 - Student Scores Report | Problem

- A **lecturer** stores the **scores** obtained by a group of students in the same order in which the students appear in the attendance register. All scores are entered on **one line and separated by spaces**.
- Because values are entered manually, input **errors are possible**. A valid score must be between 0 and 20, inclusive. Any value outside this interval must be reported as invalid and excluded from all subsequent calculations.
- After cleaning the input, the program must produce an **overall class report**, identify the strongest and weakest results, classify students into performance groups, and provide a simple visual summary of the distribution.

Input example: 12 18 7 15 21 10 -2 16 19 8 14
Valid range: 0 ... 20 Invalid examples: 21, -2

Independent 1 - Student Scores Report | Requirements I

- Input cleaning and global analysis:
 - read and convert all scores to integers;
 - separate invalid and valid scores;
 - preserve the original order of valid scores;
 - display the number of valid scores and the invalid values removed;
 - calculate the average score;
- Extremes and ranking:
 - compute `min()` and `max()` valid score and display the range `[min, max]`;
 - display the **three lowest scores** from lowest to highest;
 - display the **three highest scores** from highest to lowest;
 - if fewer than three values exist, display all available values.

Independent 1 - Student Scores Report | Requirements II

- Classify each valid score:
 - Failing: 0-9
 - Pass: 10-13
 - Good: 14-16
 - Excellent: 17-20
- For every category:
 - display count and percentage;
 - for non-empty categories, display the observed [min, max].
- Visual summary and copy:
 - print a **horizontal histogram** (a bar for each class) where each “*” represents one student;
 - create a separate copy of the valid-score list;
 - replace every failing score with 0 only in the copy;
 - display original valid scores and adjusted copy to verify that the original was not modified.

Independent 1 - Student Scores Report | Implementation Tasks

- Plan your solution in phases:
 - input conversion and validation;
 - global statistics and top/bottom values;
 - classification into four lists;
- Then complete:
 - category report and histogram;
 - independent copy and modification;
 - final output formatting.
- Choose suitable list operations yourself.
Do not hard-code results for the example input.

Independent 2 - Transport Delay Analyzer | Problem

- A **transport company** records the **departure delay**, in minutes, of all services operated during one day. The observations are stored in **chronological order**: the first value corresponds to the first departure of the day, the second value to the next departure, and so on.
- A **negative value** means that the service departed **earlier** than scheduled, 0 means that it departed exactly on time, and a **positive value** represents a **delay**.
- The company wants to analyze both **overall punctuality** and **how the magnitude of delays evolves throughout the day**. For the graphical profile, early departures are treated as zero delay, but the original chronological data must remain unchanged.

Example: -3 0 2 7 4 12 15 9 3 0 -1 5
negative = early 0 = on time positive = delayed

Independent 2 - Transport Delay Analyzer | Requirements I

- Basic analysis:
 - read and convert the chronological observations;
 - display the total number of departures;
 - count early, on-time and delayed departures;
 - create a list containing only positive delays.
- For positive delays:
 - if at least one exists, display [min, max] delay range;
 - display the three largest delays, largest first;
 - if fewer than three exist, display all available delays;
 - ask for a threshold and list delays strictly above it.

Independent 2 - Transport Delay Analyzer | Requirements II

- Chronological analysis:
 - compare every departure with the immediately preceding one:
 - count how many times the recorded delay increased;
 - display the increasing consecutive pairs.
 - The order of the list must be preserved.
- Daily delay profile:
 - create a corrected copy of the original where negative values become 0;
 - leave the original list unchanged;
 - Histogram:
 - print one line for every departure;
 - each * represents one minute of delay.

Independent 2 - Transport Delay Analyzer | Implementation Tasks

- Organize the program into:
 - input and classification;
 - positive-delay filtering and extreme values;
 - threshold filtering;
 - consecutive-value analysis.
- Finish with:
 - creation of a visualization copy;
 - chronological * bars using the corrected values.
- Remember: comparing consecutive observations requires positions or indexes.

```
Departure 1:  
Departure 2:  
Departure 3: **  
Departure 4: *****  
Departure 5: ****
```

Independent 3 - Competition Ranking Manager

| Problem

- The final order of a competition is stored as a **list of participant names**. The position of each element represents the participant's **ranking**: the first element is the winner, the second element is in second place, and so on.
- After the initial results have been recorded, the organizers may need to apply corrections. Two participants may have been recorded in the wrong order, one participant may be disqualified, or another participant may need to be inserted after an official review.
- The organizers must retain the original ranking for comparison, so every correction must be applied only to a separate working copy.

```
ranking = ["Ana", "Bruno", "Carla", "Diogo", "Eva"]  
original ranking → preserved    working ranking → corrected
```

Independent 3 - Competition Ranking Manager I

- Prepare and inspect the ranking:
 - create an independent copy for modifications;
 - preserve the original list unchanged;
 - display positions starting from 1;
 - ask for a participant and report whether the name is present;
 - if present, display the current human-readable position.
- Apply corrections:
 - ask for two names and swap them only if both are present;
 - ask for a disqualified participant and remove only if present;
 - ask for a participant to insert and a target position;
 - validate the requested position before insertion.

Independent 3 - Competition Ranking Manager I

Requirements II

- Final views:
 - display the podium: first three participants;
 - display separately all remaining participants starting from fourth place;
 - compare corrected and original rankings;
 - report whether any modification occurred.
- Final verification:
 - display the original ranking;
 - display the corrected ranking;
 - make it clear that the original ranking was preserved.

Independent 3 - Competition Ranking Manager | Implementation Tasks

- Plan the main phases:
 - copy and initial display;
 - search and position reporting;
 - swap and removal;
- Then implement:
 - validated insertion;
 - podium / remaining slicing;
 - comparison and final display.
- Choose the appropriate list methods and indexing operations yourself.

Lecture 7 - Working with the Materials

- For each exercise you receive a student starter and a complete commented solution.
- Guided exercise: complete the starter together in class using the implementation-task slide as a step-by-step map.
- Laboratory exercises: work semi-autonomously in class; the implementation-task slides provide strong hints about the processing sequence.
- Independent exercises: use the requirements as an exam-style specification. The implementation-task slides identify only the main phases, leaving the detailed solution choices to you.
- All programs should work with inputs following the stated format, **not only with the examples** shown in the slides.