**Question1: Student Grades and Courses**

Your school's administration wants to keep track of student grades across different courses.

Tasks:

1. Create a dictionary called students where keys are student names (e.g., "Alice", "Bob"). and values are dictionaries. Each inner dictionary should contain keys representing. courses (e.g., "Math", "Science") and values representing their grades in those courses.

Include at least three students and two courses per student.

2. Print the students dictionary.

3. Iterate through the students dictionary and print each student's name and their average grade across all courses.

4. Add a new course and grade for one of the existing students.

**Question2: Movie Ticket Booking**

**1. Data Setup**

o Create a list called theaters. Each element is a dictionary with keys:

 "name" (theater name)

 "screens" (a nested list of screen dictionaries)

o Each screen dictionary has:

 "screen\_number" (int)

 "seats" (dict mapping seat IDs like "A1" to True/False for booked status)

**2. Print Theaters**

o Print the full theaters list.

**3. Show Availability**

o For a given theater and screen, list all unbooked seats.

**4. Book a Seat**

o Write a function that takes theater name, screen number, and seat ID and marks

it booked.

**5. Cancel a Screen**

o Remove one screen dictionary from a selected theater.

**Question 3: Filtering and Creating a Subset CSV**

You have a large customer database in a CSV file, and you need to extract only customers from a specific city.

**Functional Requirement:**

1. Create a CSV file named customers.csv with the following content:

Code snippet

**customers.csv ( create a file with the below rows )**

customer\_id,name,email,city

C101,John Doe,john@example.com,New York

C102,Jane Smith,jane@example.com,Los Angeles

C103,Peter Jones,peter@example.com,New York

C104,Alice Brown,alice@example.com,Chicago

C105,Robert Green,robert@example.com,New York

2. Write a Python script to open customers.csv in read mode.

3. Filter the customers to only include those from "New York".

4. Write the filtered customer data into a new CSV file named new\_york\_customers.csv.

This new file should also include the header.

5. Print the content of new\_york\_customers.csv to confirm the filtering.

**Question 4: Web Scraping Assignment**

Read a list of URLs from urls.txt, verify each returns HTTP **status 200**, scrape **all <a href="..."> links**, normalize them to absolute URLs, and write results to separate output files:

* googleurls.txt
* ciscourls.txt
* wikiurls.txt

Also produce a short **summary report** with counts and basic link analytics.

**urls.txt**

https://www.google.com

https://www.cisco.com

https://www.wikipedia.org

**Functional Requirement:**

* Read urls.txt line-by-line; **ignore blanks and comments** (lines starting with #).
* For each URL:
  + Make an HTTP GET request with a **custom User-Agent** header and a **timeout** (e.g., 10s).
  + If **status != 200** → record it in summary.csv with total\_links=0 and skip scraping.
  + If **status == 200**:
    - Parse HTML and extract **all <a> tags that have href**.
    - Convert relative links to **absolute URLs** using the page’s base URL.
    - Keep only http/https links
    - Write one link per line to the site-specific file:
      * googleurls.txt for www.google.com
      * ciscourls.txt for www.cisco.com
      * wikiurls.txt for www.wikipedia.org
    - Count **unique domains** from the collected links for the summary.
* Must handle **any number of URLs** (don’t hardcode to 3).
* Handle **errors gracefully** (timeouts, invalid URLs) and still produce summary.csv.

**Question 5:**

**Using os library:**

1. Display all files in the **current directory** (non-recursive).
2. Display all files from **C:** (on Windows) or **/** (on macOS/Linux).
3. Display the **present working directory**.
4. Display the **current login/username**.
5. Display only the files ending with **.txt** in the current directory.
6. Display all files in the current directory **with their sizes** (in bytes).
7. Count how many **files** and how many **folders** exist in the current directory. Also list the **top 5 largest files** by size.
8. **Recursively walk** from a chosen start folder and produce a CSV-style report with columns: path, is\_dir, size\_bytes, modified\_iso, ext.
9. Compute the **total size** of all regular files in the current directory and print it in **human-readable** form (KB/MB/GB).
10. List files **modified in the last N days** (N is an input). Print their paths and modified times.
11. Create a **workspace tree** in the current directory: data/, temp/, logs/, backups/. Print which folders were created vs already existed.
12. Find and print all files whose names **match a user-provided pattern** (e.g., \*.log or report\_\*.csv) in the current directory.

**Using shutil**

1. **Copy** a selected file from data/ to backups/ Confirm by showing source and destination details. ( create data folder and backups folder)
2. **Move** files **older than X days** from data/ to archive/ (create archive/ if needed). Print how many were moved.
3. **Delete only empty folders** under a chosen root (do not remove folders with files). Print how many were deleted.
4. Create a **timestamped ZIP backup** of the data/ folder in backups/ (e.g., backup\_YYYYmmdd\_HHMMSS.zip). Print the final archive path and its size.
5. Perform a **one-way directory sync** from data/ → mirror/:

* Copy new/updated files,
* Remove files in mirror/ that no longer exist in data/,
* Print a summary of added/updated/removed.

**Using psutil**

1. **Sample CPU & memory** every second for **N seconds**. At the end, print min/avg/max for CPU% and memory%.
2. List the **top 5 processes** by **CPU** or **Memory** usage (user chooses). Show PID, name, and usage%.
3. Show **disk usage** for each mounted partition (total/used/free and usage%). **Warn** if any partition is above **80%** usage.
4. Show **network I/O counters** since boot (bytes sent/received). Also show per-interface stats.
5. If available, print **battery percentage** and **time remaining**; otherwise print a clear message that no battery is detected.

**Using sys & platform**

1. Produce a **Python environment report**: sys.version, sys.executable, sys.platform, the first **five** entries of sys.path, and sys.getsizeof("hello"). Save it to a text file.
2. Produce a **platform/OS report**: platform.uname(), system, release, machine, processor, python\_version(), and an **OS-specific tip** (e.g., Windows vs POSIX path separators). Save it to a text file.

Using random library:

1. Generate 10 random numbers between 1 to 100
2. Create a list with all the numbers from 101 to 200 and shuffle all the numbers in the list and display it.
3. Display a random decimal number between 0 and 1

**Question 6:**

From the working folder (e.g., C:\Users\Desktop\yourprogramsdirectory on Windows), upload **all .txt files** to a given **FTP server**. Add robust **validations**, **exception handling**, and **logging**. Produce a **summary report** at the end.

####################### ftp server details #####################

Host: ftp.dlptest.com

Port: 21

User: dlpuser

Password: rNrKYTX9g7z3RgJRmxWuGHbeu

##########################################################

**Functional Requirement:**

1. **Source scope:** Operate only on the **current working directory** (no recursion).
2. **File selection:** Consider files that end with **.txt** (case-insensitive).
3. **Connection:**
   * Connect to an FTP server using **host**, **port** (default 21), **username**, **password**.
   * Support **passive mode** (default ON).
4. **Remote destination:** Upload to a specified **remote directory**; create it if it doesn’t exist (when permitted).
5. **Upload behavior:**
   * Use **binary upload**.
6. **Summary report:** After completion, generate upload\_summary.csv with columns:  
   filename, size\_bytes, status (UPLOADED/SKIPPED/ERROR), remote\_path, duration\_s, notes
7. **Console output:** Print a concise progress line per file and a final totals line:
   * Total files found, Uploaded, Skipped, Errors, Total time.

**Logging Requirement:**

* Create a logs/ folder in the current directory.
* Write to **logs/ftp\_upload.log** at **INFO** level (and above), with timestamps.
* Also print **INFO** messages to console.
* Include at least:
  + Start/stop of program with timestamps.
  + Connection attempt, login success/failure (mask password).
  + Per-file actions: selected/skip reason/upload start/upload success (bytes, duration)/error message.
  + Final totals.

Question 7 :

**Objective**

Connect to a Linux server via **SSH**, run two commands—date and ls -la—capture their outputs, and save them to a single text file. Also produce a basic log file of what happened.

**What to do**

1. **Inputs** (provide via arguments or prompts):
   * host (e.g., example.com)
   * username
   * password
2. **Connect** to the server using SSH (timeout ~15s).
3. **Run commands (in order):**
   * date
   * ls -la
4. **Write outputs** to command\_output.txt in this format:
5. ==== date ====
6. (stdout here)
7. [exit\_code: X]
8. ==== ls -la ====
9. (stdout here)
10. [exit\_code: Y]
    * If there’s any **stderr**, append it under each section as:
    * [stderr]
    * (stderr text)
11. **Logging**
    * Create a logs/ folder.
    * Write logs/ssh\_task.log with timestamped INFO messages:
      + program start/stop
      + host/port (mask password)
      + connection success/failure
      + each command run, exit code, duration
      + any error messages
12. **Console summary**
    * Print a one-line summary:
13. **Exit code**
    * Exit **0** if both commands ran (even if outputs differ).
    * Exit **1** if connection/auth failed or any command couldn’t be executed.

Question 8 :

Write a small program that uses a **class**, **objects**, and a **constructor**.

**Do this**

1. **Create a class** Employee
   * Default constructor (no parameters).
   * Attributes with simple defaults:  
     employee\_id, first\_name, last\_name, department, role, city, salary\_inr, scores (list).
   * Method: average\_score() → returns average of scores or N/A if no scores.
2. **Create objects**
   * Make **3** employees using the default constructor: e1, e2, e3.
   * After creation, **set their attributes** (IDs, names, dept, role, city, salary).
   * Add **2–3 scores** (numbers 0–100) to each.
3. **Show output on screen**
   * Print one line per employee:  
     <employee\_id> | <first\_name> <last\_name> | Dept=<department> | Role=<role> | City=<city> | Scores=<comma separated> | Avg=<value> | Salary=<salary\_inr>
4. **Write to file**
   * Save the same lines to **employees\_report.txt**.

**Keep it simple :**

* If a score is outside **0–100**, **don’t add it**.
* If salary\_inr is negative, treat it as **0**.

**Deliverable**

* employees\_report.txt (contains the 3 employee lines).

Question 9 :  
Simple URL Downloader  
  
**urls.txt**  
https://cdn.codewithmosh.com/image/upload/v1702942822/cheat-sheets/python.pdf

https://www.utc.fr/~jlaforet/Suppl/python-cheatsheets.pdf

https://itlab.uta.edu/courses/dasc-cse-5300/current-offering/General/python-cheat-sheets.pdf

## <https://realpython.com/files/python_cheat_sheet_v1.pdf> **What to do**

1. **Create folders**
   * Ensure a local folder named **downloads/** exists.
   * Ensure a local folder named **logs/** exists.
2. **Read URLs**
   * Open **urls.txt** and read line by line.
   * Ignore **blank lines** and lines starting with #.
3. **Download each URL**
   * Make an HTTP **GET** request (timeout ~**30s**).
   * If the **status is 200**, save the response body into downloads/.
   * **Filename rule (simple):**
     + Use the **last path segment** of the URL (e.g., python.pdf).
     + If that’s empty or has no name, use file\_<number>.bin.
     + If a file **already exists**, **append (1), (2), …** to avoid overwriting.
   * If the **status is not 200** or there’s an error, **skip** and record the error.
4. **Summary CSV**
   * After all URLs, write **summary.csv** in the current folder with columns:  
     url, status, saved\_as, size\_bytes, duration\_s, error
   * For failures, fill saved\_as and size\_bytes as blank and put a short message in error.
5. **Logging (basic)**
   * Write a log file **logs/download.log** with timestamped INFO lines:
     + Start/stop of program
     + Each URL started/finished (status, bytes, duration)
     + Any error messages (short)
6. **Console summary**
   * Print one line at the end, for example:  
     Total: 4 | Downloaded: 4 | Failed: 0 | Output: downloads/ | Summary: summary.csv | Log: logs/download.log
7. **Exit code**
   * **0** if all URLs were processed and files were downloaded or skipped cleanly
   * **1** if any URL failed to download (non-200 or exception)