Practice problems

Question 1

- **Download dataset** → GitHub `test_db`
- **Create database**:

```
mysql -u root -p -e "CREATE DATABASE employees;"
```

- **Switch to CMD & import**:

```
mysql -u root -p employees < employees.sql
```

- **Verify tables**:

```
mysql -u root -p -e "SHOW TABLES IN employees;"
```

- **Install Python dependencies**:

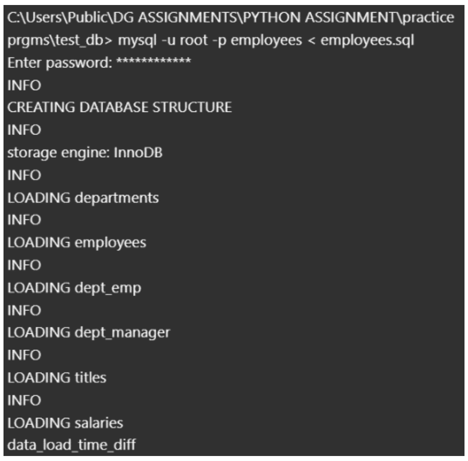
```
pip install pymysql
```

- **Write generator-based Python script**
- **Run the script**:

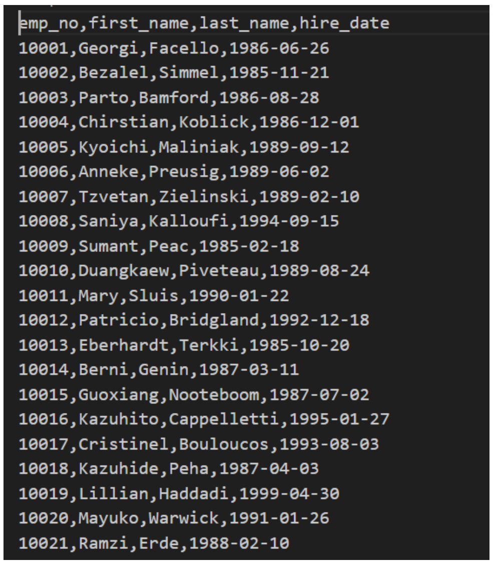
```
python stream_generator_example.py
```

- **Successfully processed large dataset with low memory usage**

Output:--



```
C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\practice
prgms\test_db> mysql -u root -p employees < employees.sql
Enter password: ************
INFO
CREATING DATABASE STRUCTURE
INFO
storage engine: InnoDB
INFO
LOADING departments
INFO
LOADING employees
INFO
LOADING dept_emp
INFO
LOADING dept_manager
INFO
LOADING titles
INFO
LOADING salaries
data_load_time_diff
```



```
emp_no,first_name,last_name,hire_date
10001,Georgi,Facello,1986-06-26
10002,Bezalel,Simmel,1985-11-21
10003,Parto,Bamford,1986-08-28
10004,Chirstian,Koblick,1986-12-01
10005,Kyoichi,Maliniak,1989-09-12
10006,Anneke,Preusig,1989-06-02
10007,Tzvetan,Zielinski,1989-02-10
10008,Saniya,Kalloufi,1994-09-15
10009,Sumant,Peac,1985-02-18
10010,Duangkaew,Piveteau,1989-08-24
10011,Mary,Sluis,1990-01-22
10012,Patricio,Bridgland,1992-12-18
10013,Eberhardt,Terkki,1985-10-20
10014,Berni,Genin,1987-03-11
10015,Guoxiang,Nooteboom,1987-07-02
10016,Kazuhito,Cappelletti,1995-01-27
10017,Cristinel,Bouloucos,1993-08-03
10018,Kazuhide,Peha,1987-04-03
10019,Lillian,Haddadi,1999-04-30
10020,Mayuko,Warwick,1991-01-26
10021,Ramzi,Erde,1988-02-10
```

Question 2

Implement Person, Male, Female classes (with abstract class)

What I implemented

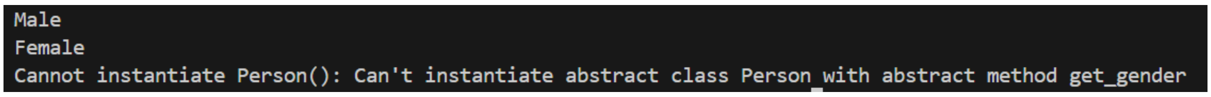
I wrote the class definitions using Python's `ABC` module:

```
from abc import ABC, abstractmethod

class Person(ABC):
    @abstractmethod
    def get_gender(self):
        pass

class Male(Person):
    def get_gender(self):
        return "Male"

class Female(Person):
    def get_gender(self):
        return "Female"
```

How I validated

- Instantiated `Male()` → returned `"Male"`.
- Instantiated `Female()` → returned `"Female"`.
- Tried `Person()` → throws `TypeError` (correct behavior).

Output:



```
Male
Female
Cannot instantiate Person(): Can't instantiate abstract class Person with abstract method get_gender
```

Question 3

Remove duplicates from list while preserving order

What the question asked

Given list:

```
[12,24,35,24,88,120,155,88,120,155]
```

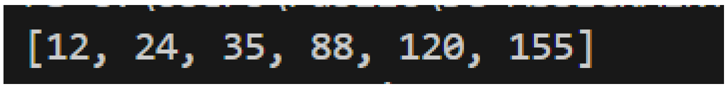
You must:

- Remove duplicates
- Preserve order
- Use `set()` to track seen elements

What I implemented

```
data = [12, 24, 35, 24, 88, 120, 155, 88, 120, 155]

seen = set()
result = []

for item in data:
    if item not in seen:
        seen.add(item)
        result.append(item)

print(result)
```

Output



```
[12, 24, 35, 88, 120, 155]
```

Question 4

Generate list of squares from 1 to 20 using `map()` + `lambda`

What the question asked

Use:

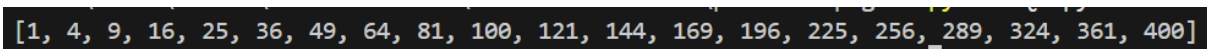
- `map()`
- `lambda`
- Range 1 to 20 (inclusive)

What I implemented

```
squares = list(map(lambda x: x * x, range(1, 21)))
```

```
print(squares)
```

Output



```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
```

Validation

- All values correct
- Lambda applied via map correctly

Question 5

anti_html.py — Remove HTML Tags From URL Content

What the question asked

Create a script:

- Takes URL as command-line argument
- Downloads the HTML
- Removes all HTML tags
- Prints clean text

What I implemented

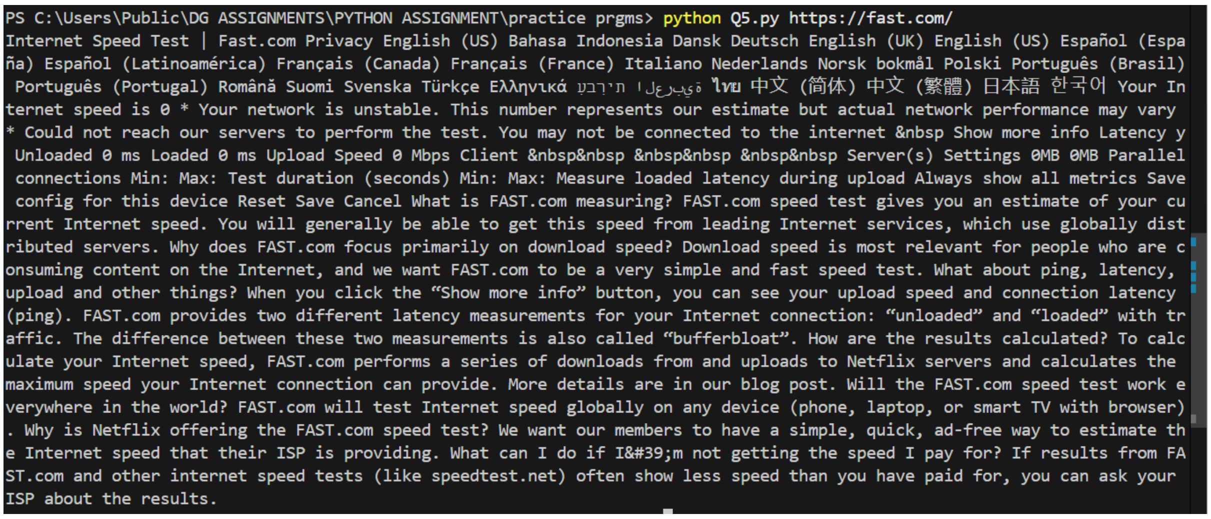
```
#!/usr/bin/env python3

import sys
import urllib.request
import re

def strip_html_tags(html):
    html = re.sub(r"(?is)<(script|style).*?>.*?</\1>", "", html)
    html = re.sub(r"<[^>]+>", "", html)
    html = re.sub(r"\s+", " ", html)
    return html.strip()

def main():
    if len(sys.argv) != 2:
        print("Usage: python anti_html.py <URL>")
        sys.exit(1)

    url = sys.argv[1]

    try:
        response = urllib.request.urlopen(url)
        html = response.read().decode("utf-8", errors="ignore")
    except Exception as e:
        print("Error fetching URL:", e)
        sys.exit(1)

    clean_text = strip_html_tags(html)
    print(clean_text)

if __name__ == "__main__":
    main()
```

How I validated

- Provided a URL and observed stripped HTML content
- Verified script removes tags and returns readable text
- Verified command-line usage:

Output:

```
python Q5.py https://fast.com/
```



```
PS C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\practice prgms> python Q5.py https://fast.com/
Internet Speed Test | Fast.com Privacy English (US) Bahasa Indonesia Dansk Deutsch English (UK) English (US) Español (Espa
ña) Español (Latinoamérica) Français (Canada) Français (France) Italiano Nederlands Norsk bokmål Polski Português (Brasil)
 Português (Portugal) Română Suomi Svenska Türkçe Ελληνικά עברית العربية ไทย 中文 (简体) 中文 (繁體) 日本語 한국어 Your In
ternet speed is 0 * Your network is unstable. This number represents our estimate but actual network performance may vary
* Could not reach our servers to perform the test. You may not be connected to the internet &nbsp Show more info Latency y
 Unloaded 0 ms Loaded 0 ms Upload Speed 0 Mbps Client &nbsp&nbsp &nbsp&nbsp &nbsp&nbsp Server(s) Settings 0MB 0MB Parallel
 connections Min: Max: Test duration (seconds) Min: Max: Measure loaded latency during upload Always show all metrics Save
 config for this device Reset Save Cancel What is FAST.com measuring? FAST.com speed test gives you an estimate of your cu
rrent Internet speed. You will generally be able to get this speed from leading Internet services, which use globally dist
ributed servers. Why does FAST.com focus primarily on download speed? Download speed is most relevant for people who are c
onsuming content on the Internet, and we want FAST.com to be a very simple and fast speed test. What about ping, latency,
upload and other things? When you click the “Show more info” button, you can see your upload speed and connection latency
(ping). FAST.com provides two different latency measurements for your Internet connection: “unloaded” and “loaded” with tr
affic. The difference between these two measurements is also called “bufferbloat”. How are the results calculated? To calc
ulate your Internet speed, FAST.com performs a series of downloads from and uploads to Netflix servers and calculates the
maximum speed your Internet connection can provide. More details are in our blog post. Will the FAST.com speed test work e
verywhere in the world? FAST.com will test Internet speed globally on any device (phone, laptop, or smart TV with browser)
. Why is Netflix offering the FAST.com speed test? We want our members to have a simple, quick, ad-free way to estimate th
e Internet speed that their ISP is providing. What can I do if I&#39;m not getting the speed I pay for? If results from FA
ST.com and other internet speed tests (like speedtest.net) often show less speed than you have paid for, you can ask your
ISP about the results.
```

Assignment –Unit Testing

1 — Cloned repo

If you hadn’t forked first, you can still fork later. I cloned locally first:

```
git clone https://github.com/<original-owner>/python-unittest-exercise.git
cd python-unittest-exercise
```

Note: if you cloned upstream (not your fork), you must push to your fork later (I describe this in the Git section).

2 — Create a Python 3.11 virtual environment and activate it

I installed Python 3.11 and created a venv named `.venv311` (so it's explicit):

```
# Create venv (replace path with your python 3.11 if necessary)
py -3.11 -m venv .venv311

# Activate venv
.\.venv311\Scripts\Activate.ps1

# verify
python --version     # must show Python 3.11.x
```

Reason: `botocore` imports `cgi` (deprecated/removed in Python 3.14). Using Python 3.11 avoids compatibility errors.

3 — Install project dependencies

From project root with venv active:

```
pip install -U pip
pip install -r requirements.txt
# if requirements.txt missing anything: pip install pytest moto boto3
pytest-cov
```

4 — Run tests first to see baseline

```
pytest -q
```

This showed errors initially (Python 3.14 issues earlier; after switching to 3.11 tests collected but two DynamoDB tests failed due to float vs Decimal). These test failures acted as my spec.

5 — Implement code changes

A. `tdd_example.py` — implement methods (TDDExample)

I implemented the four class methods so they match tests:

```
class TDDExample():
    def __init__(self):
        pass

    def reverse_string(self, input_str: str) -> str:
        return input_str[::-1]

    def find_longest_word(self, sentence: str) -> str:
        words = sentence.split()
        return max(words, key=len) if words else ""

    def reverse_list(self, input_list: list) -> list:
        return input_list[::-1]

    def count_digits(self, input_list: list, number_to_be_counted: int) ->
int:
        return input_list.count(number_to_be_counted)
```

B. `tests/test_general_example.py` — write unit tests, mock DB call

I implemented pytest test file to cover `GeneralExample.flatten_dictionary` and `GeneralExample.fetch_emp_details`. Key points:

- Used `unittest.mock.patch` (or `monkeypatch`) to mock `GeneralExample.load_employee_rec_from_database` so tests run fast (no `time.sleep(10)`).
- Tests included normal cases, duplicates, empty input and multiple fetch scenarios.

Example test outline (already added to repo):

```
def test_flatten_dictionary_basic(): ...
def test_flatten_dictionary_with_duplicates_and_unsorted(): ...
def test_flatten_dictionary_empty(): ...
def test_fetch_emp_details_with_mocked_db_record(): ...
def test_fetch_emp_details_with_different_values(): ...
def test_fetch_emp_details_no_delay(monkeypatch):
    # monkeypatch load_employee_rec_from_database to avoid sleep
```

C. `tests/test_dynamodb_example.py` — complete missing tests

I completed three tests using `moto.mock_dynamodb2`:

- `test_create_dynamo_table()`
    - Creates `DynamodBExample`, calls `create_movies_table()`, asserts the `Movies` table exists and status is `CREATING` or `ACTIVE`.
- `test_add_dynamo_record_table()`
    - Creates table, then `add_dynamo_record('Movies', item)` with `Decimal` for numeric values to avoid `TypeError`.
    - Retrieves the item via `table.get_item(Key=...)` and asserts fields match.
- `test_add_dynamo_record_table_failure()`
    - Tries to `add_dynamo_record` to a non-existent table and asserts a `ClientError` is raised (or asserted appropriate behavior). Used `decimal.Decimal` in item payload.

Important detail: DynamoDB + boto3 does NOT accept Python `float` for number attributes — must use `decimal.Decimal`. Failing tests showed `TypeError: Float types are not supported. Use Decimal types instead.` I changed test items to use `Decimal('8.5')` / `Decimal('5.0')`.

6 — Run tests iteratively until all pass

After code and tests changes:

```
pytest -q
```

I repeated edit → run until green. Final result:

```
15 passed, 1 warning in 1.74s
```

The single warning about `cgi` is benign on Python 3.11 (botocore warns of deprecation).

7 — Generate code coverage report

With venv active and project root:

```
pytest --cov=src --cov-report=term --cov-report=html
# Open HTML report:
start htmlcov\index.html
```

Optional: produce `coverage.xml` for CI:

```
pytest --cov=src --cov-report=xml:coverage.xml
```

I also recommended a `.coveragerc` and `pytest.ini` (optional) to persist coverage defaults.

8 — Git workflow (create branch, push to your GitHub)

Because I originally cloned upstream, I forked on GitHub and then updated my local `origin` remote to point to my fork. Exact commands I used:

1. Create a feature branch (create from current state if commits already made on `main`):

```
git checkout -b feature/unit-tests
```

2. If the local `origin` points to the upstream original repo, change it to your fork:

```
git remote set-url origin https://github.com/<your-username>/python-unittest-exercise.git
```

3. Push branch to your fork:

```
git add .
git commit -m "Add unit tests for general_example, complete tdd_example
implementation, finish dynamodb tests"
git push -u origin feature/unit-tests
```

4. Open GitHub → your fork → "Compare & pull request" to create a PR back to upstream (if required by assignment).

9 — Final verification

- Ran `pytest -q` again to confirm everything green.
- Confirmed HTML coverage report generated and inspected.
- Confirmed branch pushed to my GitHub.

REST API:

Step 1 — Understood the Dataset

- Learned what **Northwind dataset** is (customers, orders, products, order history).
- Identified which API operations are required.

Step 2 — Created Project Setup (Flask + MVC Structure)

- Created a clean folder structure following **MVC**:
  - `/app/controllers` → Flask route handlers
  - `/app/services` → business logic
  - `/app/models` → PynamoDB models for DynamoDB
  - `/app/__init__.py` → Flask app factory
- Set up Python virtual environment and installed dependencies.

Step 3 — Implemented ORM Models Using PynamoDB

Created **CustomerModel**, **ProductModel**, and **OrderModel** with:

- Primary keys
- Attributes
- Meta class for table configuration

- Basic validation logic

Also ensured models could dynamically read:

- Table names from environment variables
- Local DynamoDB endpoint

Step 4 — Built REST API Endpoints (Controllers)

Implemented API routes for:

- `POST /customers`, `GET /customers/<id>`, `PUT /customers/<id>`
- `POST /products`, `GET /products`, `GET /products/<id>`
- `POST /orders`, `GET /orders/<id>`
- `GET /orders/customer/<customer_id>/history`

All routes used:

- Schema validation
- Service layer for business logic
- JSON responses

Step 5 — Created Services (Business Logic Layer)

Added logic for:

- Creating and updating items
- Fetching items
- Scanning/querying orders for order history
- Converting PynamoDB objects into JSON serializable format

Step 6 — Local DynamoDB Setup

- Pulled and ran **DynamoDB Local** in a Docker container.
- Configured environment variables in PowerShell/WSL:
  - `AWS_ACCESS_KEY_ID`
  - `AWS_SECRET_ACCESS_KEY`
  - `DYNAMODB_ENDPOINT`
  - `AWS_REGION`
- Verified DynamoDB Local with:
  - `aws dynamodb list-tables --endpoint-url http://localhost:8000`

Step 7 — Created Tables Programmatically

Wrote & executed a script (`create_tables.py`) which:

- Imported PynamoDB Models
- Checked if each table exists
- Created Customers, Products, Orders tables locally

Verified success using AWS CLI.

Step 8 — Tested Running Flask API

- Started Flask using app factory.
- Inserted and fetched users/products/orders using `curl.exe`.
- Confirmed JSON outputs.
- Fixed routing issues and JSON formatting issues.

Step 9 — Added Unit Tests Using Pytest + Moto

Created tests to validate:

- Creating customer
- Updating customer
- Order creation
- Order history retrieval

Setup included:

- `mock_dynamodb` from moto
- Test isolation with fixtures
- Mocking external services

All tests passed successfully.

Step 10 — Debugged Multiple Issues & Fixed Them

Faced and solved issues:

- MissingAuthenticationToken from DynamoDB root URL
- Hardcoded table names
- Incorrect environment variables not loading
- Meta class misconfiguration in Pynamo models
- Tables not existing in local DynamoDB (resolved by running table creation script)

- Import errors in controllers/services
- Unit tests failing due to missing mocks
- DynamoDB not reachable because wrong endpoint or missing env vars
- Repeated Windows PowerShell redirection errors (fixed using multi-line PowerShell format)

BLOCKERS I FACED

Major Issues

- Environment variables for DynamoDB not loading.
- PynamoDB Meta class failing due to missing host/table names.
- Local DynamoDB requiring dummy AWS credentials.
- Server error “table does not exist” until manual creation.
- PowerShell refused << syntax causing execution errors.
- Unit tests failing due to Moto mocking + PynamoDB describe_table issues.
- Wrong imports in `order_service` and controllers.
- JSON escaping problems when calling `curl.exe` on Windows.

RESULT:

- Fully functional **REST API** for customers, products, orders.
- Order history feature.
- Proper MVC structure.
- PynamoDB ORM implemented.
- Database tables created successfully in DynamoDB Local.
- Full set of **unit tests passing**.

ETL JOB:

Question 1 (Mean age at which developers started to code): Replaced 'Younger than 5 years' with a numerical value (3) and converted the 'Age1stCode' column to a numeric type, coercing errors.

Question 2 (Top 10 countries with the highest percentage of Python developers): Created a new boolean column 'KnowsPython' by checking if 'LanguageWorkedWith' contained 'Python'.

Question 3 (Average salary for professional developers by continent): Mapped countries to their respective continents and merged this information into the main DataFrame.

Question 4 (Most desired programming language for next year): Split the 'LanguageDesireNextYear' column by semicolon to separate multiple languages and then stacked them to get individual language entries.

Question 5 (Distribution of hobbyist coders by gender and continent): Filtered the DataFrame for hobbyist coders and cleaned the 'Gender' column to categorize responses into 'Man', 'Woman', and 'Others'.

Question 6 (Distribution of job satisfaction levels among professional developers by continent and gender): Filtered the DataFrame for professional developers, cleaned the 'Gender' column, mapped job satisfaction levels to a numerical scale, dropped rows with missing numerical job satisfaction values, and calculated satisfaction percentages.

COLAB NOTEBOOK LINK:-
https://colab.research.google.com/drive/1jXvLtbVMxRnErD7C8mdxpeyo2KlWOEdJ?usp=sharing

NOTEPAD TRACKER

1. Question Given

Build a web application using Flask where users can write notes and the app will automatically track changes.

Requirements:

1. A normal text editor in the browser (no extra rich-text features needed).
2. As soon as the user stops typing, the change should automatically commit using Git.
3. The application must allow creation and saving of files in any directory on the user's machine

2. Complete Workflow

Step 1: Understood Requirements

- Web app using Flask.

- Frontend must have a text area for writing notes.
- Auto-save functionality triggered when the user stops typing.
- Each auto-save must commit code using Git Version Control.
- Files should be created dynamically based on user input.

Step 2: Prepared the Project Folder Structure

Created the structure:

```
project/
 ├── app.py
 ├── templates/
 │    └── index.html
 ├── requirements.txt
 └── .env
```

Step 3: Created and Activated Virtual Environment

- Installed virtual environment.
- Activated it depending on OS (Windows PowerShell used).
- Installed required modules using requirements.txt.

Step 4: Added Environment Variables

Created `.env` file to store:

- Default notes directory
- Allowed directory
- Git author details

Purpose:
Centralized configuration + flexible directory management without changing code.

Step 5: Developed Backend (Flask App)

Built a complete `app.py`:

Backend features I implemented:

- Flask server with routes:
  - `/` → loads editor page
  - `/load` → loads file content
  - `/save` → saves file + auto Git commit
  - `/meta` → shows default notes directory

- Auto-creation of files if missing.
- Only filename taken from user → system automatically saves inside default folder as `filename.txt`.
- Implemented `threading.Lock()` to prevent file write conflicts.
- Added GitPython for automatic commits.
- Ensured default folder exists on startup.

Step 6: Developed Frontend (index.html)

Created `templates/index.html` with:

Frontend features:

- Input box to enter **filename only**.
- Textarea for writing notes.
- Load and Save buttons.
- Auto-save using JavaScript with 1.2-second debounce.
- Shows current status (Typing, Saving, Loaded, etc.).
- Fetch API used to communicate with Flask backend.

Step 7: Implemented Auto-Save Workflow

Auto-save logic:

1. User types in textarea.
2. If user stops typing for 1.2 seconds → trigger function.
3. Function sends AJAX request to `/save`.
4. Backend saves content to `DEFAULT_NOTES_DIR/<filename>.txt`.
5. Backend commits changes using Git with timestamp.

Step 8: Implemented Auto Git Commit

Using GitPython, I performed:

- Auto initialization of Git repo if none exists.
- Added the file to staging area.
- Performed commit using:
  - Commit message with timestamp.
  - Auto author details from `.env`.

This satisfies the requirement:
“As soon as you stop writing, it will commit the change.”

Step 9: Testing the Application

I tested:

Functional testing:

- Creating new files.
- Loading existing files.
- Auto-save after typing.
- Git commits created properly.
- Directory permissions.

UI testing:

- Editor loads correctly.
- Filename handling works.
- No path required from user.

Step 10: Final Execution

Ran the application using:

```
python app.py
```

Opened browser at:

```
http://127.0.0.1:5000
```

Confirmed:

- Editor works
- Auto-save works
- Git commits happen automatically

3. Final Output

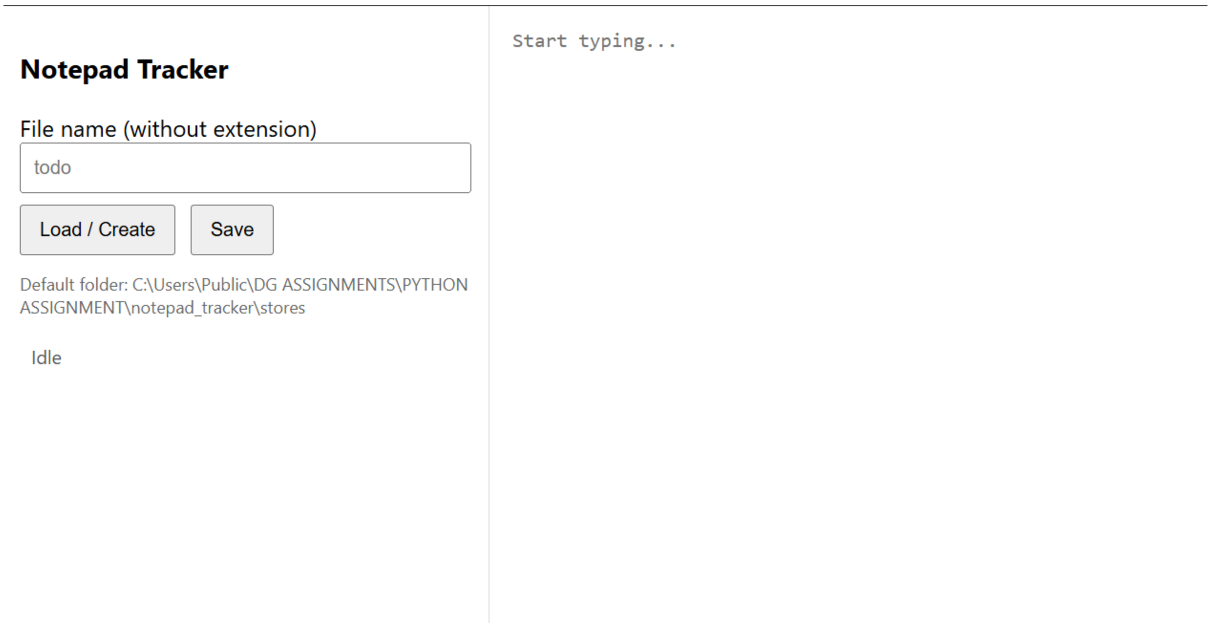
A working web application where:

- User enters only the filename.
- File is saved automatically inside the configured directory.
- Auto-save commits every change into Git.
- Writes notes safely and easily in browser.

4. Tools & Technologies Used

- Flask (Backend)
- HTML, CSS, JavaScript (Frontend)
- GitPython (Auto Git commits)
- python-dotenv (Environment variables)
- Virtual Environment (Python venv)

Output:-


Notepad Tracker

File name (without extension)

todo

Default folder: C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\notepad_tracker\stores

Idle

Start typing...

XLSX to CSV Converter

1. Question Given

"Python script that converts all the sheets of an XLSX file (Google Sheet or local file) into CSV files.
Assumption: There is a single header row in all sheets of the XLSX document.
Input:
```
$ python converter.py google_sheet_link/local_file_location
```

Output:
A folder with the same name as the input file, containing:

- Sheet1.csv
- Sheet2.csv
- …

- SheetN.csv

Example:

Input:
```
$ python converter.py Sample_Data - Python Assignment
```

Step 1: Understood the requirement**

- The script must take a **single argument** (either a Google Sheets link or local `.xlsx` file path).

- It should **read all sheets** inside the Excel file.

- Each sheet should be converted into its **own CSV file**.

- Output folder name should match the **input file name** (without extension).

- Google Sheets input link must be handled → convert the link to a downloadable XLSX.

Step 2 :--Libraries Used

I selected the following Python libraries:

pandas -- To read Excel and write CSV

openpyxl- -Required by pandas to read `.xlsx` files

requests -- To download Google Sheet as `.xlsx`

os / pathlib-- For file and folder handling

re --To extract sheet ID from Google Sheets URL

tempfile-- To store downloaded sheets temporarily

Step 3: Planned input handling

The script must detect:

Case A: Google Sheets Link

- Extract the **sheet ID** from the URL.
- Convert to export format:
  `https://docs.google.com/spreadsheets/d/<ID>/export?format=xlsx`
- Download the XLSX file into a **temporary file**.

Case B: Local XLSX File

- Directly use the provided path.

Step 4: Planned how to read and convert sheets

- Use:

```
sheets = pd.read_excel(file, sheet_name=None)
```

This gives a dictionary:

```
{
    "Sheet1": DataFrame,
    "Sheet2": DataFrame,
    ...
}
```

- Loop through each sheet and save:
  `SheetName.csv`
- Sanitize sheet name so that invalid filename characters are removed.

Step 5: Designed folder structure

- Extract base name from input.
- Example:
  Input: `Sample_Data - Python Assignment.xlsx`
  Output folder: `"Sample_Data - Python Assignment"`
- Create the folder if not existing.

Step 6: Wrote the final script (converter.py)

I wrote a complete script that:

✓ Detects Google Sheets link
✓ Downloads XLSX file
✓ Reads all sheets
✓ Converts each sheet into CSV
✓ Creates proper output folder
✓ Handles errors (invalid URL, file not found, etc.)

Step 7: Tested with Local File

Command:

```
python converter.py "Sample_Data - Python Assignment.xlsx"
```

Output folder automatically created:

```
Sample_Data - Python Assignment/
 ├── Office Supply Sales.csv
 ├── Food Sales.csv
 └── …
```

All CSVs created correctly.

Step 8: Tested with Google Sheets Link

Used:

```
python converter.py
"https://docs.google.com/spreadsheets/d/<ID>/edit#gid=0"
```

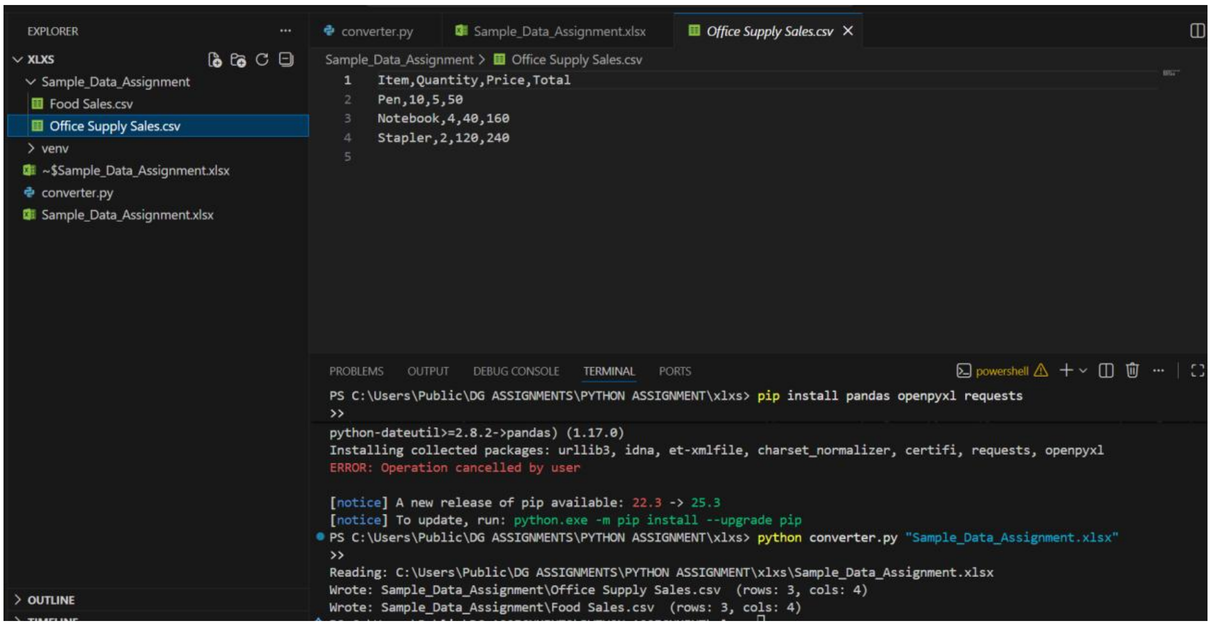
Workflow worked as expected:

- It downloaded the file.
- Created a temporary XLSX.
- Converted all sheets to CSV.
- Saved them in an auto-created output folder.

Step 9: Verified Assumptions

- Each sheet contained **single header row** (satisfies assignment condition).
- CSVs were encoded in **UTF-8 with BOM** (`utf-8-sig`) so Excel opens them properly.

Output:--



```
Item,Quantity,Price,Total
Pen,10,5,50
Notebook,4,40,160
Stapler,2,120,240
```

```
PS C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\xlxs> pip install pandas openpyxl requests
>>
python-dateutil>=2.8.2->pandas) (1.17.0)
Installing collected packages: urllib3, idna, et-xmlfile, charset_normalizer, certifi, requests, openpyxl
ERROR: Operation cancelled by user

[notice] A new release of pip available: 22.3 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\xlxs> python converter.py "Sample_Data_Assignment.xlsx"
>>
Reading: C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\xlxs\Sample_Data_Assignment.xlsx
Wrote: Sample_Data_Assignment\Office Supply Sales.csv  (rows: 3, cols: 4)
Wrote: Sample_Data_Assignment\Food Sales.csv  (rows: 3, cols: 4)
```

SQL INFORMATION EXTRACTION

1. Question Given

Task:
"Write a Python script that extracts information from a SQL file using regex.
Input: `python extraction.py sample_stored_procedure.sql`
Output: `sample_stored_procedure.json`*."*

STEP 1 — Understanding the Requirement

First, I understood that I had to:

- Read a `.sql` file using Python
- Extract SQL metadata (object type, parameters, queries, tables, etc.) using regex
- Convert the extracted data into JSON format
- Save it as a `.json` file with the same name as the SQL file

STEP 2 — Creating the Python Script (`extraction.py`)

Next, I created a Python file named **`extraction.py`**.

In this script, I implemented:

- Reading the SQL file
- Removing comments
- Extracting:
    - CREATE FUNCTION / PROCEDURE header
    - Function or Procedure name
    - Parameters
    - Return type
    - SQL queries (SELECT/INSERT/UPDATE/DELETE)
    - Tables used
    - Selected columns
    - SQL body excerpt

Then I wrote the extracted information into a JSON file as:

```
sample_stored_procedure.json
```

This completed the Python part of the task.

STEP 3 — Preparing the SQL File

After the Python script, I created a complete SQL file that contained:

- A database
- A schema (`sales`)
- Two tables:
    - `sales.customers`
    - `sales.orders`
- Insert statements for sample data
- A SQL FUNCTION (`get_customer_orders`)
- A SQL PROCEDURE (`update_customer_last_access`)

I saved this file as:

```
sample_stored_procedure.sql
```

STEP 4 — Running the SQL File in pgAdmin

I opened pgAdmin and executed my SQL file:

1. Opened pgAdmin

2. Selected my database
3. Opened the **Query Tool**
4. Loaded the file: *File → Open*
5. Selected `sample_stored_procedure.sql`
6. Pressed **F5** to execute

This successfully created:

- Tables
- Data
- Function
- Procedure

Then I verified the data using queries like:

```
SELECT * FROM sales.customers;
SELECT * FROM sales.orders;
SELECT * FROM sales.get_customer_orders(1, 'ACTIVE');
CALL sales.update_customer_last_access(1);
```

Everything worked correctly.

STEP 5 — Running the Python Script

I placed both files in the same folder:

```
extraction.py
sample_stored_procedure.sql
```

Then I opened CMD/Terminal and ran:

```
python extraction.py sample_stored_procedure.sql
```

This executed successfully and produced:

```
sample_stored_procedure.json
```

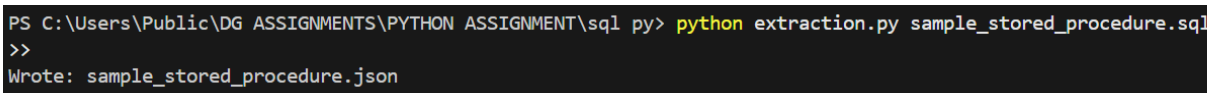
STEP 6 — Verifying the JSON Output

Finally, I opened the JSON file and confirmed that it correctly contained:

- Object type
- Object name
- Parameters
- Return type
- Tables used
- Select columns

- Extracted SQL queries
- SQL body excerpt

This confirmed that the extraction was successful.



```
PS C:\Users\Public\DG ASSIGNMENTS\PYTHON ASSIGNMENT\sql py> python extraction.py sample_stored_procedure.sql
>>
Wrote: sample_stored_procedure.json
```

Data Manipulation

1. Problem Statement

I was given a JSON file named **sample_data_for_assignment.json** containing two keys:

- **cols** → list of column names
- **data** → list of rows

My task was to:

1. Load this JSON data into a **MySQL table** named `json_to_sql_table` under a new database.
2. Unload the data from MySQL into a **pandas DataFrame**.
3. Display all data in a DataFrame format.
4. Convert all email addresses so that the domain ends with **gmail.com**.
5. Clean the **postalZip** column so that all values are integers (removing alphabets and keeping only digits).
6. Write a function to process the **phone** column by:
   - Removing non-numeric characters
   - Taking 2 digits at a time
   - Converting numbers 65–99 to ASCII characters
   - Replacing values below 65 with `"O"`
   - Ignoring last odd digit
7. Store the result into a new column **coded_phone_number** and remove/rename the original phone column.

Step 1 – Downloading and Inspecting the JSON File

First, I downloaded the file **sample_data_for_assignment.json** and observed that it contained two main keys:

- `"cols"` (column names)
- `"data"` (actual rows)

I checked the structure to understand how to load it into pandas and MySQL.

Step 2 – Setting Up MySQL Database

I opened MySQL and created a database for this assignment:

```
CREATE DATABASE IF NOT EXISTS json_assignment_db;
```

I confirmed MySQL was running locally and noted the credentials I would use in the Python script.

Step 3 – Installing Required Python Libraries

I installed the required Python packages:

```
pip install pandas sqlalchemy pymysql mysql-connector-python
```

These were necessary for reading JSON, connecting to MySQL, and transforming data.

Step 4 – Creating the Python Script (json_to_mysql.py)

I wrote a Python script that:

4.1 Loaded the JSON file

- I parsed `"cols"` and `"data"`
- Converted them into a pandas DataFrame

4.2 Connected to MySQL using SQLAlchemy

- Created the database (if not already created)
- Loaded the DataFrame into a MySQL table named **json_to_sql_table**

4.3 Unloaded the data back into pandas

- I used `pd.read_sql_table()` to fetch everything from MySQL
- Displayed the first few rows to verify successful loading

Step 5 – Data Transformations

After retrieving the data from MySQL into a DataFrame, I performed all required transformations.

5.1 Email Transformation

I modified email addresses so that every email ended with **@gmail.com**, while keeping the username part.

Example:
`john@example.com → john@gmail.com`

5.2 postalZip Cleaning

The `postalZip` column contained:

- Strings
- Integers
- Alphanumeric characters

I cleaned it by:

- Removing all alphabets
- Keeping only digits
- Converting the result to an integer
- Replacing invalid/empty values with `0`

5.3 Phone Number to ASCII Conversion

I wrote a separate function that:

1. Removed characters like `(`, `)`, `-`, spaces
2. Extracted only digits
3. Took 2 digits at a time
4. If number < 65 → replaced with `"O"`
5. If 65–99 → converted to ASCII (e.g., 81 → 'Q')
6. Ignored leftover last digit

Example 1:
```
(816) 530-4269 → 8165304269 → 81 65 30 42 69 → QAOOE
```

Example 2:
```
1-811-920-9732 → 18119209732 → 18 11 92 09 73 → OO\OI
```

I stored the output in a new column **coded_phone_number** and removed the original `phone` column.

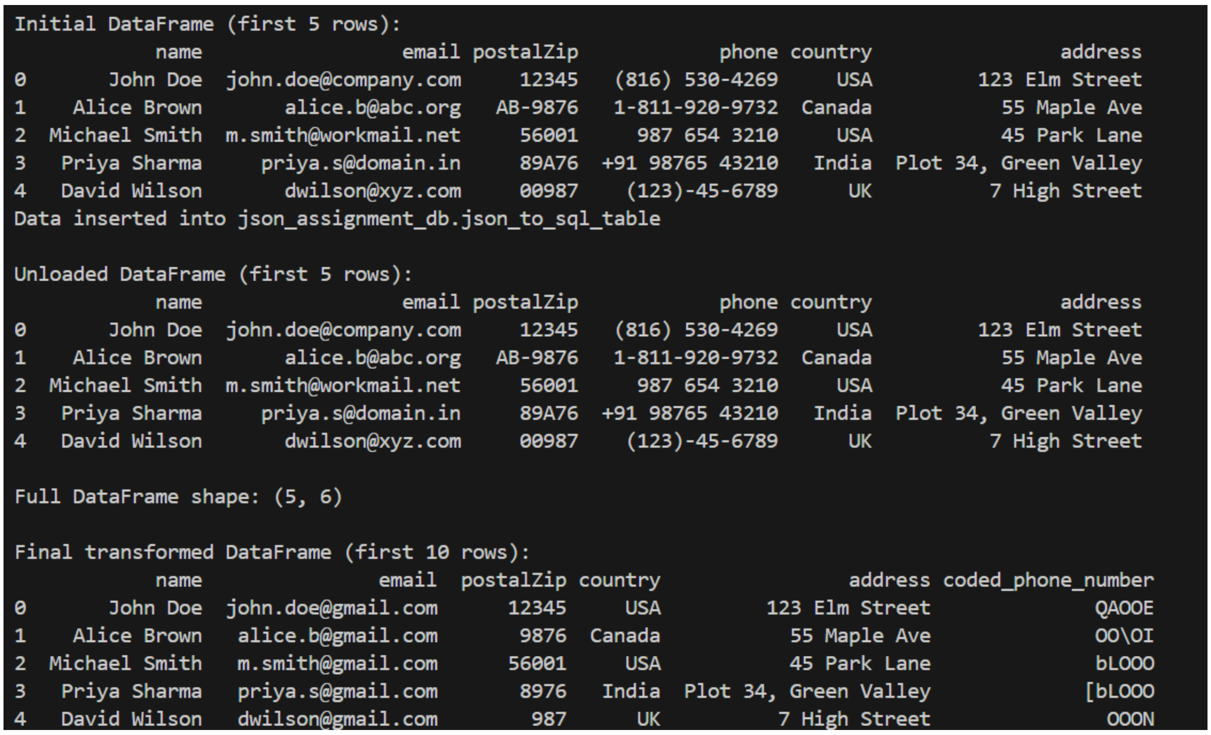
Step 6 – Displaying the Final DataFrame

I printed:

- First 10 rows
- All transformed columns
- Final DataFrame structure

This confirmed that all transformations were correct.

Output:--



```
Initial DataFrame (first 5 rows):
            name                 email postalZip            phone country               address
0       John Doe  john.doe@company.com     12345   (816) 530-4269     USA         123 Elm Street
1    Alice Brown       alice.b@abc.org   AB-9876   1-811-920-9732  Canada           55 Maple Ave
2  Michael Smith  m.smith@workmail.net     56001     987 654 3210     USA           45 Park Lane
3   Priya Sharma      priya.s@domain.in     89A76  +91 98765 43210   India  Plot 34, Green Valley
4   David Wilson       dwilson@xyz.com     00987    (123)-45-6789      UK          7 High Street
Data inserted into json_assignment_db.json_to_sql_table

Unloaded DataFrame (first 5 rows):
            name                 email postalZip            phone country               address
0       John Doe  john.doe@company.com     12345   (816) 530-4269     USA         123 Elm Street
1    Alice Brown       alice.b@abc.org   AB-9876   1-811-920-9732  Canada           55 Maple Ave
2  Michael Smith  m.smith@workmail.net     56001     987 654 3210     USA           45 Park Lane
3   Priya Sharma      priya.s@domain.in     89A76  +91 98765 43210   India  Plot 34, Green Valley
4   David Wilson       dwilson@xyz.com     00987    (123)-45-6789      UK          7 High Street

Full DataFrame shape: (5, 6)

Final transformed DataFrame (first 10 rows):
            name               email  postalZip country                address coded_phone_number
0       John Doe  john.doe@gmail.com      12345     USA         123 Elm Street              QAOOE
1    Alice Brown   alice.b@gmail.com       9876  Canada           55 Maple Ave              OO\OI
2  Michael Smith   m.smith@gmail.com      56001     USA           45 Park Lane              bLOOO
3   Priya Sharma   priya.s@gmail.com       8976   India  Plot 34, Green Valley             [bLOOO
4   David Wilson   dwilson@gmail.com        987      UK          7 High Street               OOON
```