**HOSPITAL PATIENT RECORD & BILLING SYSTEM**

**PROJECT DOCUMENTATION**

**Table of Contents**

1. Abstract
2. System Overview
3. Architecture
4. Dependencies
5. Database Schema Overview
6. Database Connection Module (db\_config.py)
7. Main CLI Controller (hospital\_main.py)
8. Person Module (person.py)
9. Patient Module (patient.py)
10. Doctor Module (doctor.py)
11. Service Module (service.py)
12. Appointment Module (appointment.py)
13. Billing Module (billing.py)
14. CSV Import Module (import\_csv.py)
15. CLI Workflow Images

**ABSTRACT**

The Hospital Patient Record & Billing System (or) Hospital Management System (HMS) is a comprehensive, menu-driven application designed to streamline and automate the administrative and operational tasks within a hospital environment. Built using Core Python and MySQL, this system offers an integrated platform to manage essential hospital functions including patient registration, doctor information, service tracking, appointments, and billing operations.

The system employs object-oriented architecture, with modular classes like Patient, Doctor, Appointment, Service, and Billing, each encapsulating relevant business logic and database interactions. The database operations are executed securely using MySQL connector with structured queries and validation checks to maintain data integrity.

Key features include:

* Auto-ID Generation for patients, doctors, and bills.
* Input Validation and error handling to ensure data consistency.
* Invoice Generation with detailed consultation and service charges.
* CSV Exporting for billing and appointment summaries.
* Command-Line Interface (CLI) with intuitive menu options for users.

This system enhances the efficiency of hospital operations by minimizing manual errors, improving data accessibility, and ensuring organized storage of records. It is suitable for small to medium-sized hospitals looking to digitize their core processes without a web-based overhead.

**HOSPITAL PATIENT RECORD & BILLING SYSTEM**

**System overview**

The Hospital Patient Record & Billing System is built to efficiently manage patient appointments and billing operations within a healthcare facility. It uses a MySQL database for reliable and persistent data storage and incorporates strong validation, reporting, and error-handling mechanisms to ensure accuracy and smooth system performance.

**Architecture**

* **Programming Language:** Python
* **Database:** MySQL
* **Modular Design:** The system is structured into distinct modules, separating appointment management and billing functionalities.
* **Database Connectivity:** All database interactions are handled through a shared get\_connection() function located in the db\_config.py module.

**Dependencies**

* Python 3.x
* Mysql Database
* Mysql-connector-python for database communication
* Python’s built-in csv module for data export
* Note:

Ensure the db\_config.py file is correctly set up for database access.

The serviceusagedb class, which supports billing operations, should be implemented in a separate module named service.py.

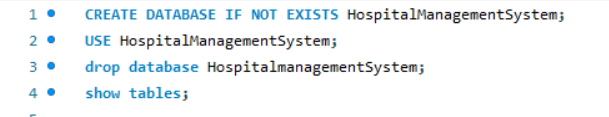
**DATABASE SCHEMA OVERVIEW**

* The system uses the following key tables (not exhaustive):
* appointments – Stores details such as appointment ID, patient, doctor, date, diagnosis, and charges.
* patients – Contains master data for all patients.
* doctors – Contains master data for all doctors.
* billing – Stores bill records including bill ID, patient ID, total amount, and billing date.
* services – Lists all available hospital services.
* billed\_services – Tracks services billed under each bill.
* temp\_service\_usage – Temporarily holds service usage data before the billing process is completed.

A screenshot of a computer

AI-generated content may be incorrect.

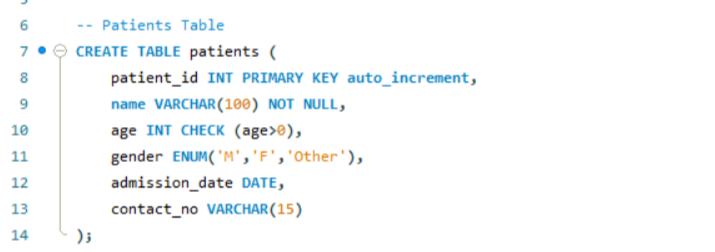
The Hospital Management System database schema created, which consists of multiple interrelated tables to manage patients, doctors, services, appointments, and billing in a hospital system.



* Creates a database named HospitalManagementSystem if it doesn't already exist.
* USE sets the context to this database for subsequent operations.
* Deletes the entire database and all its tables.
* Should be used with caution (typically for reset/testing purposes).

**1. patients Table**

* Stores basic information about hospital patients.

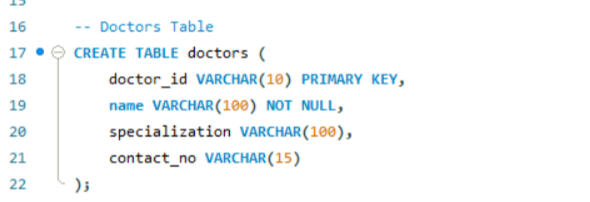


Fields:

* patient\_id: Auto-incremented unique ID.
* name: Patient's full name (mandatory).
* age: Must be a positive number (using CHECK constraint).
* gender: Must be one of 'M', 'F', or 'Other'.
* admission\_date: Date of admission.
* contact\_no: Phone number (up to 15 characters).

**2. doctors Table**

Stores doctor information.

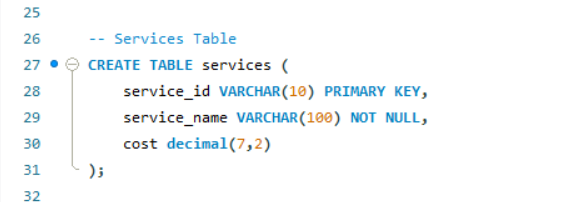


Fields:

* doctor\_id: Unique doctor identifier.
* name: Doctor's name (mandatory).
* specialization: Area of expertise (e.g., Cardiology).
* contact\_no: Phone number.

**3. services Table**

* Stores information about hospital services.

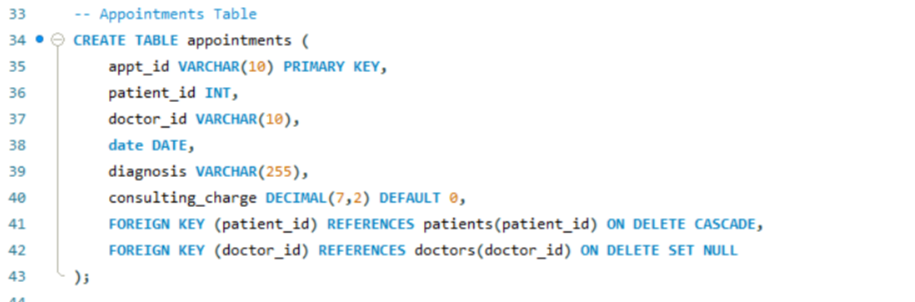


Fields:

* service\_id: Unique ID for each service.
* service\_name: Describes the service (mandatory).
* cost: Service cost, allows up to 99999.99 (7 total digits, 2 after decimal).

**4. appointments Table**

Manages appointments between patients and doctors.



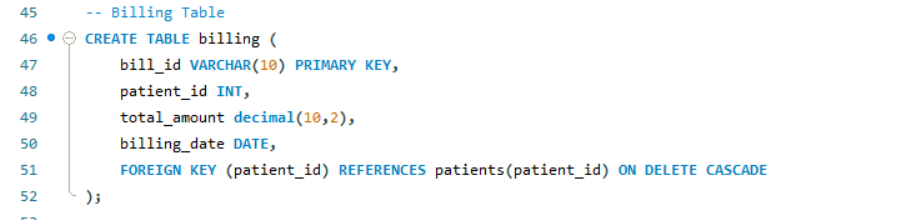
Fields:

* appt\_id: Unique appointment ID.
* patient\_id: Foreign key referencing patients.
* doctor\_id: Foreign key referencing doctors.
* date: Appointment date.
* diagnosis: Diagnosis from consultation.
* consulting\_charge: Charge for the consultation (defaults to 0).

Constraints:

* ON DELETE CASCADE: Deletes appointments if the patient is deleted.
* ON DELETE SET NULL: If a doctor is removed, the field is set to NULL.

**5. billing Table**

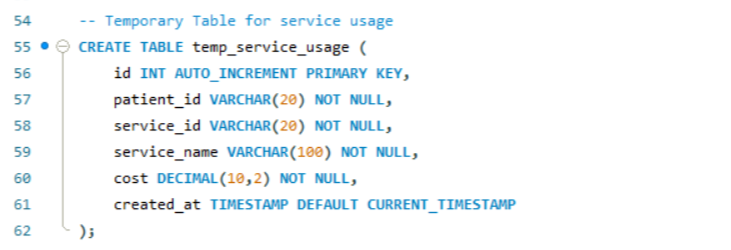
* Stores billing details for a patient.

Fields:

* bill\_id: Unique ID for the bill.
* patient\_id: Patient being billed.
* total\_amount: Sum total of all charges.
* billing\_date: Date the bill was generated.

**6. temp\_service\_usage Table**

* Stores services used by a patient temporarily before generating the final bill

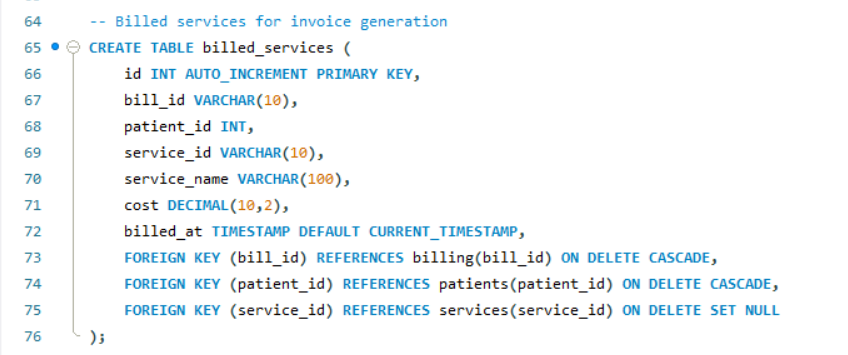


Use Case:

* Temporary log of services a patient used (like a cart).
* Later moved to billed\_services upon billing.

**7. billed\_services Table**

* Final log of services billed to the patient.



Purpose:

* Tracks which services were billed in a specific invoice.
* Useful for generating printable bills or reports.

Constraints:

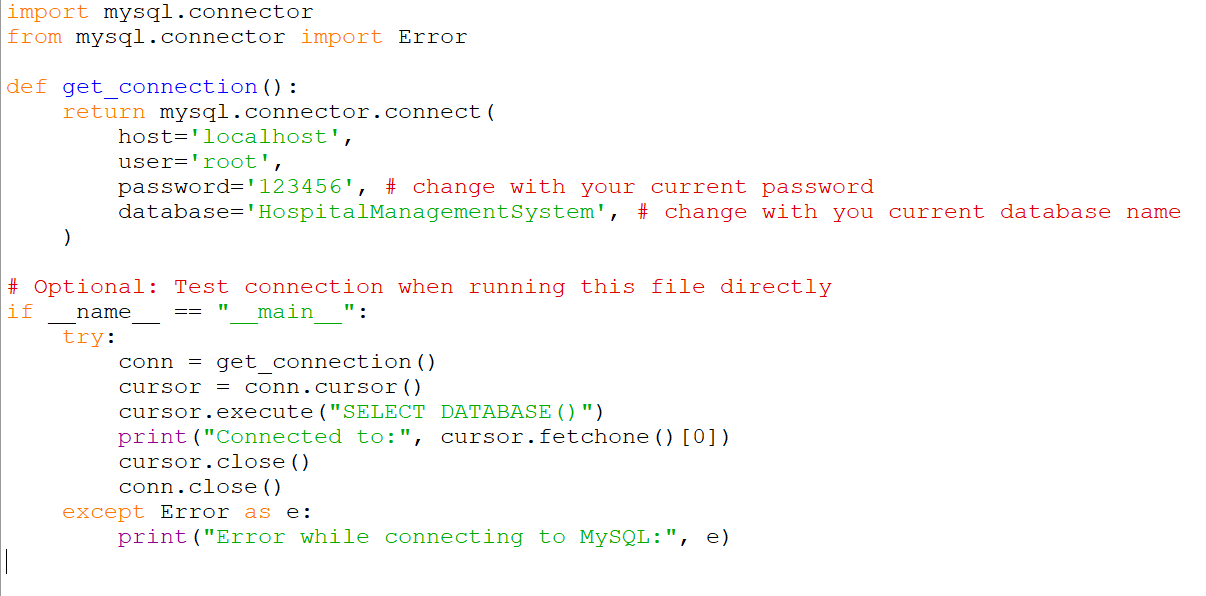
* Deletion of a patient or bill also removes the related billed services.
* If a service is removed, it will set service\_id to NULL but retain billing data.

**Table Relationships**

* patients ↔ appointments: One-to-many.
* doctors ↔ appointments: One-to-many.
* patients ↔ billing: One-to-many.
* billing ↔ billed\_services: One-to-many.
* services ↔ billed\_services: Many-to-one.
* temp\_service\_usage: Temporary staging table not directly linked by foreign keys.

**DB connection Module (db\_config.py)**

This script provides a reusable method for establishing a connection to a MySQL database. It also includes a basic test routine to verify the connection when the script is executed directly.



**Code Breakdown & Explanation**

* **import mysql.connector**: Imports the MySQL Connector module which enables Python to interact with MySQL databases using a Pythonic API.
* **from mysql.connector import Error**: Specifically imports the Error class to handle exceptions that occur during database operations.

**Function: get\_connection()**

* This function returns a **MySQL database connection object** using mysql.connector.connect().
* It uses the following **connection parameters**:
* host='localhost': Connects to the database server on the local machine.
* user='root': Uses the MySQL root user (typically an administrative account).
* password='123456': Password for the MySQL root user (should be secured in production environments).
* database='HospitalManagementSystem': Connects to the specific database used for the project.

**Purpose**: Centralizes database connection logic to ensure **code reuse** and **maintainability** across different modules of the application

**Test Connection Block**

* **if \_\_name\_\_ == "\_\_main\_\_":**
* Ensures the code block only runs when this file is executed directly, not when imported as a module.
* **conn = get\_connection()**
* Calls the get\_connection() function to establish a connection to the database.
* **cursor = conn.cursor()**
* Creates a cursor object used to execute SQL statements.
* **cursor.execute("SELECT DATABASE()")**
* Executes an SQL query to retrieve the name of the currently connected database.
* **print("Connected to:", cursor.fetchone()[0])**
* Fetches and prints the result of the query (i.e., database name).
* **cursor.close() and conn.close()**
* Closes the cursor and connection to free up resources.
* **except Error as e:**
* Catches and prints any database connection or query execution errors for debugging purposes.

**Use Case in Project**

This module is intended to be imported into other components of **Hospital Management System**, such as:

* patient.py
* billing.py
* doctor.py
* service.py
* appointment.py

**Security Tip**

* Never hardcode database credentials (like root and 123456) in production.
* Use environment variables or configuration files with proper access control.

**Main CLI Controller (hospital\_main.py)**

This is the **main menu-driven controller** for the Hospital Management System. It provides a Command Line Interface (CLI) that lets users access various modules including patient records, doctor details, services, appointments, billing, and export functionality.

**Imports**

* **db\_config**: Imports the get\_connection function used internally in each module for database interaction.
* **Patient, Doctor, Service, Appointment, Bill**: These are custom classes/modules that encapsulate logic and menus for their respective data entities.
* **tabulate**: (Imported but unused here) A third-party library for displaying tabular data in a formatted CLI table (likely used in submodules).

**export\_menu() Function**

This function displays a sub-menu for exporting reports in CSV format.

**Options:**

1. **Export Billing Summary:** Prompts the user for a filename and exports billing details using Bill.export\_billing\_summary\_to\_csv().
2. **Export Appointment Summary:** Exports appointment data to CSV using Appointment.export\_appointment\_summary\_to\_csv().
3. **Return to Main Menu:** Exits the export menu loop.

**Looped Input Handling:** Keeps running until the user chooses to return to the main menu.

**Validation:** Ensures proper file extension (.csv) and default naming when no input is provided.

**main\_menu() Function**

The main interactive menu for navigating the system.

| **Option** | **Action** | **Method Called** |
| --- | --- | --- |
| 1 | Patient Records | Patient.patient\_menu() |
| 2 | Doctor Records | Doctor.doctor\_menu() |
| 3 | Service Records | Service.service\_menu() |
| 4 | Appointment Records | Appointment.appointment\_menu() |
| 5 | Billing Records | Bill.billing\_menu() |
| 6 | Export Records | export\_menu() |
| 7 | Exit System | Breaks the loop and exits |

**Input Validation**:

* Only accepts valid numbered options.
* Displays a message for invalid entries.

**User Flow**:

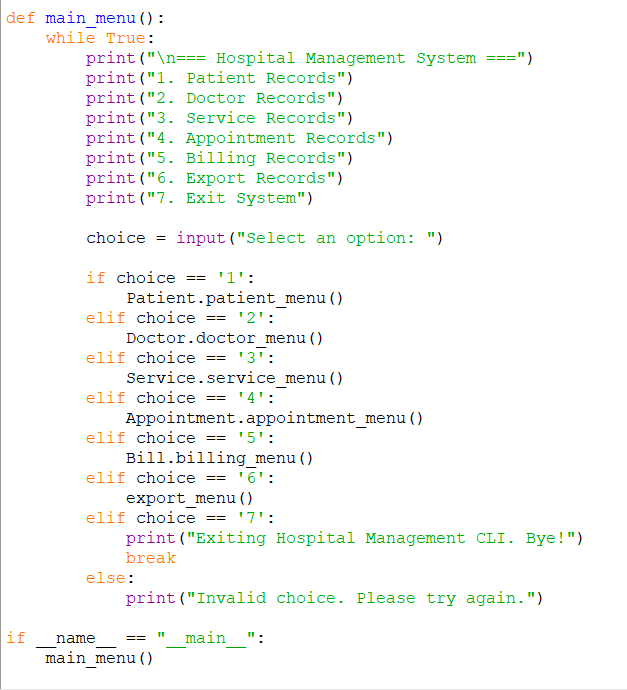
* Menu keeps displaying until user chooses to exit.

**Entry Point**

*if \_\_name\_\_ == "\_\_main\_\_":*

*main\_menu()*

This block ensures the main menu is launched only when this script is run directly (not when imported).



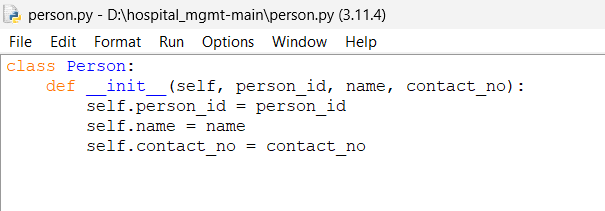
**Person Module (person.py)**

The Python class you’ve shared is a foundational OOP (Object-Oriented Programming) representation of a person.

| **Component** | **Explanation** |
| --- | --- |
| class Person: | Declares a new class named Person. |
| \_\_init\_\_ method | Constructor that gets called when a new object is created. |
| person\_id | Unique identifier for the person (could represent a patient or doctor ID). |
| name | Name of the person. |
| contact\_no | Contact number of the person. |
| self | Refers to the current instance of the class. |

This class could as a base class for specific types of people in your system, such as:

* Patient
* Doctor



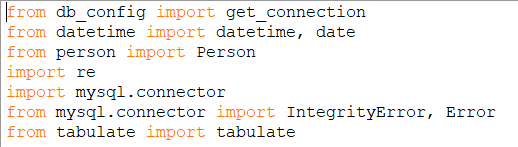
**Patient Module (patient.py)**

This class manages patient records in a hospital management system and inherits from a Person class (assumed to provide name and contact\_no attributes). Key features include:

**Functionalities:**

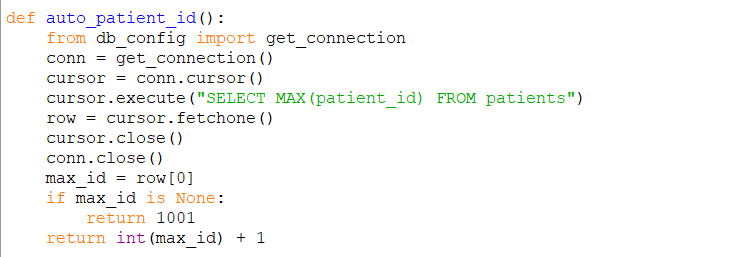
* Auto-generate patient ID
* CRUD operations: Add, View, Update, Delete
* Search by name
* Calculate admission duration
* Validate input (name, age, gender, contact number, and admission date)

**Imports**



* **get\_connection:** Custom function to connect to the MySQL database.
* **datetime, date:** For handling date comparisons.
* **Person:** A parent class providing shared attributes like name and contact\_no.
* **re:** Used for regex validation.
* **mysql.connector:** For interacting with MySQL.
* **tabulate:** Formats output in tabular form for better readability.

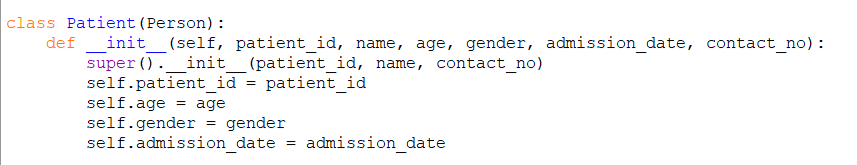
**Auto-Increment Logic**



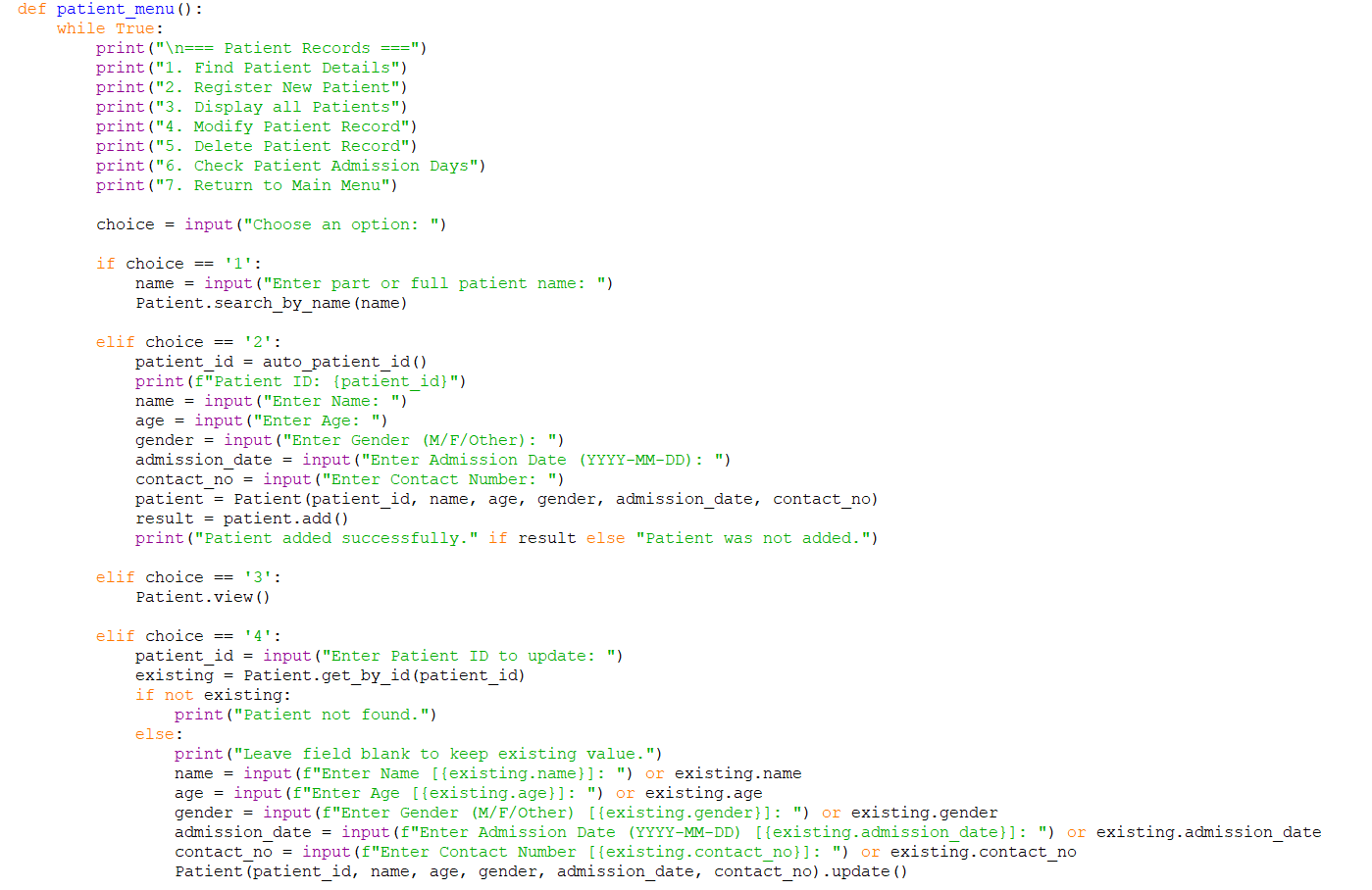
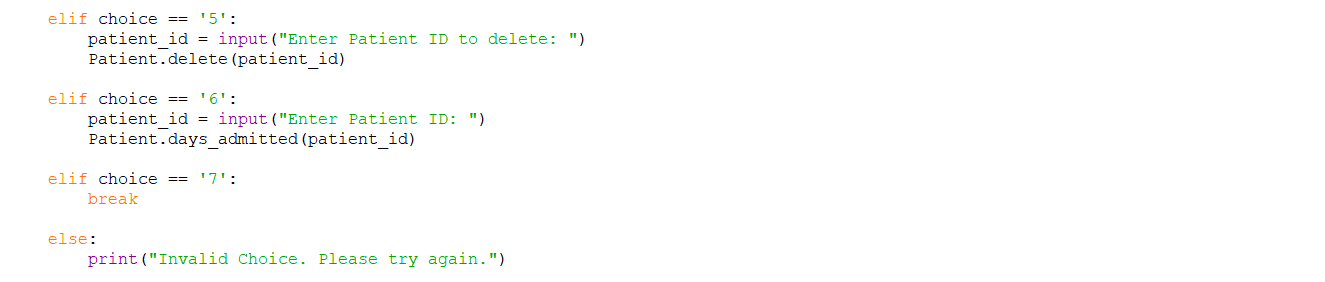
This function fetches the maximum patient\_id from the database and increments it to generate a unique new ID. If no record exists, it starts at 1001.

**Patient Class**

Inherits from Person and initializes patient-specific details like age, gender, and admission\_date.



**Menu Navigation**



Displays a command-line interface to navigate through patient-related operations:

1. Search patient by name
2. Register new patient
3. View all patients
4. Modify patient data
5. Delete patient record
6. Calculate days admitted
7. Return to main menu

**CRUD Methods**

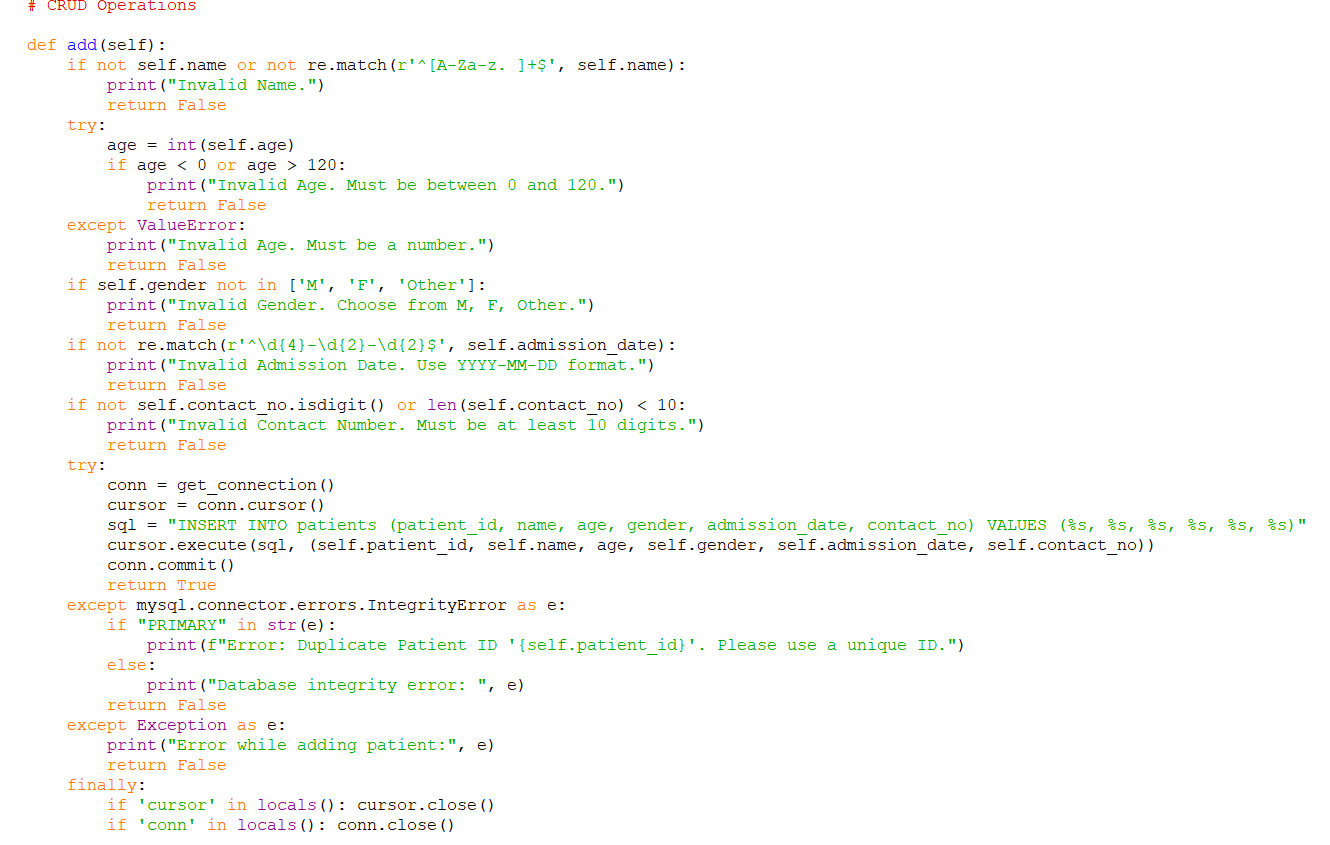
**add()**

Validates and inserts a new patient record.

Validation includes:

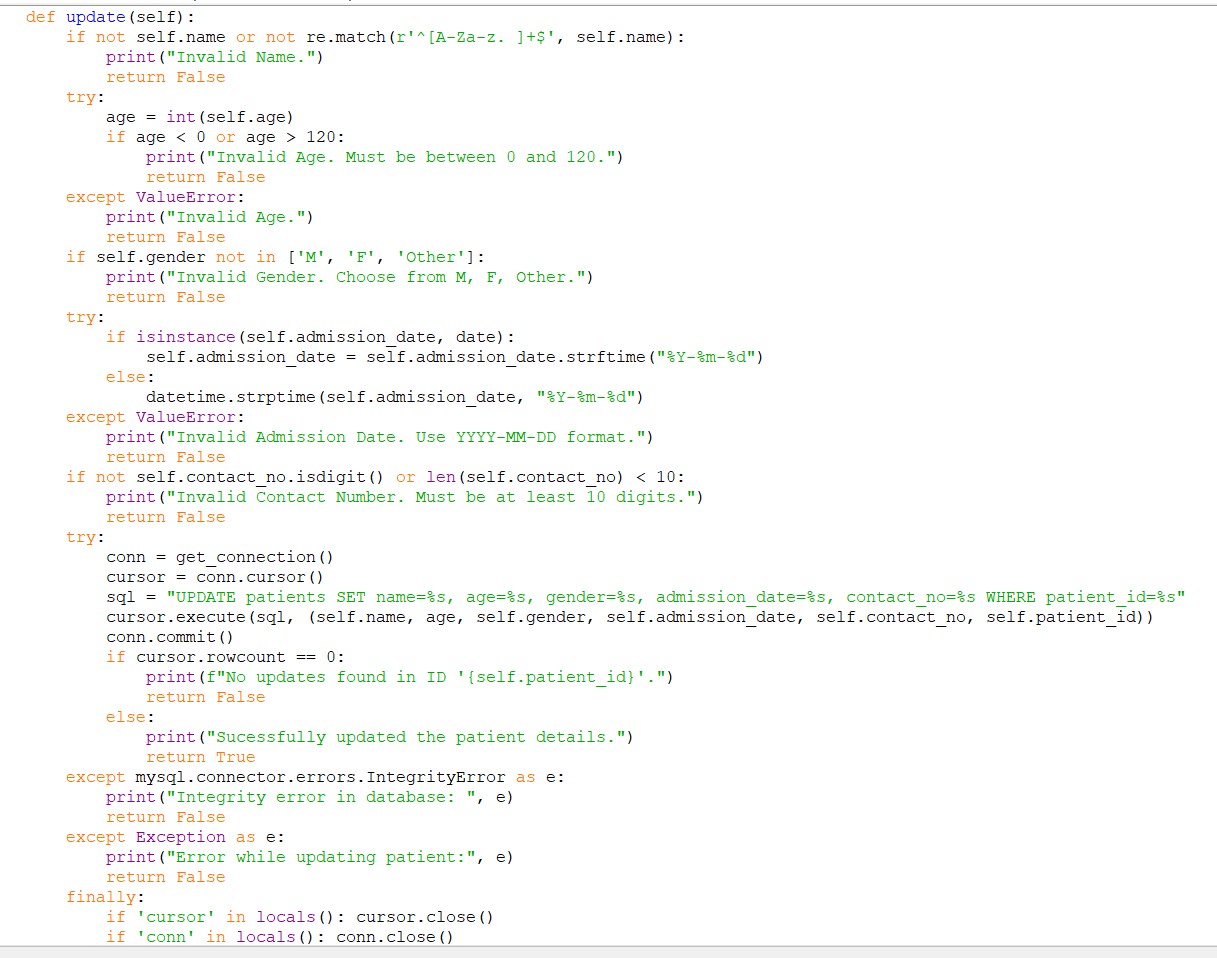
* Name must be alphabetic and contain only letters, periods, or spaces.
* Age must be between 0–120.
* Gender must be one of 'M', 'F', or 'Other'.
* Admission date must follow the YYYY-MM-DD format.
* Contact number must be numeric and at least 10 digits.

Handles integrity errors (e.g., duplicate patient\_id) and commits valid data to the patients table.



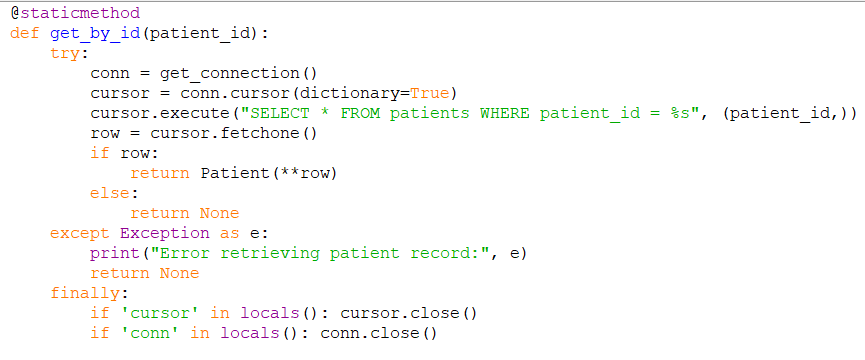
**update()**

Allows updating an existing record. Validation is similar to the add() method. Converts date objects into string if necessary and performs update query. Confirms success based on affected rows.



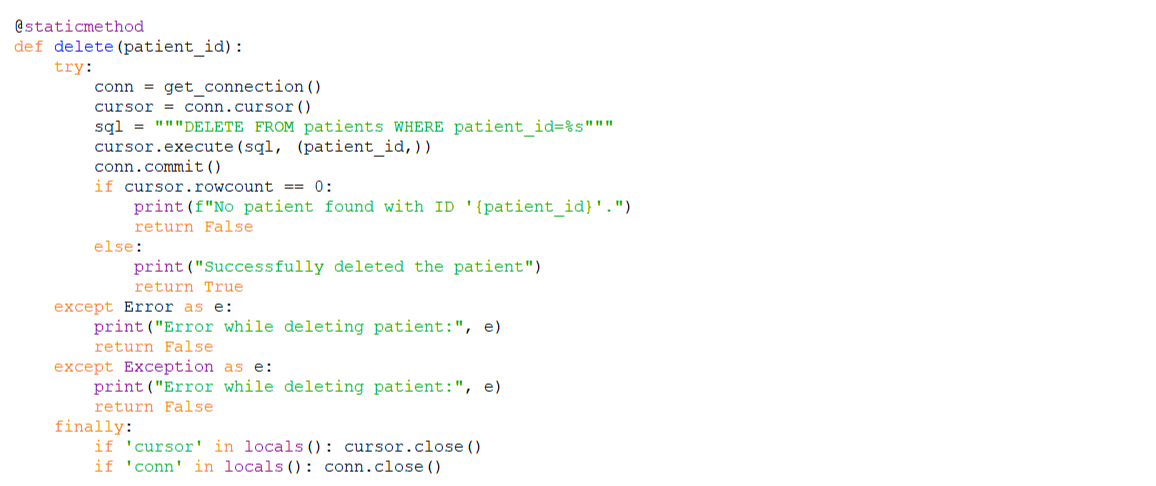
**get\_by\_id(patient\_id)**

Fetches a patient record based on patient\_id. Returns a Patient object if found, else returns None.



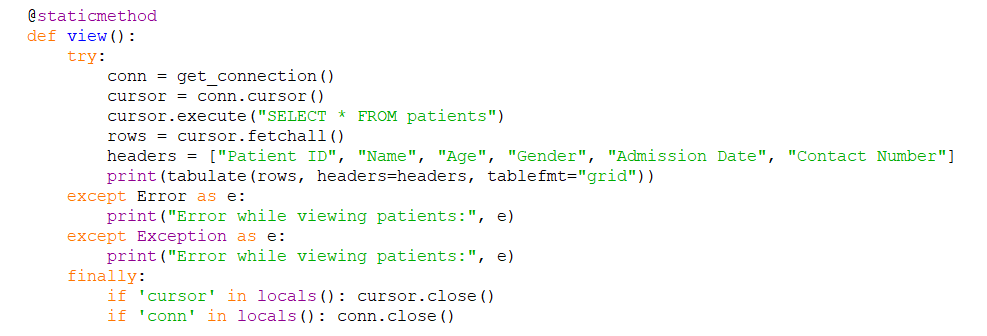
**delete(patient\_id)**

Deletes the patient record matching the patient\_id. Outputs confirmation based on deletion success.



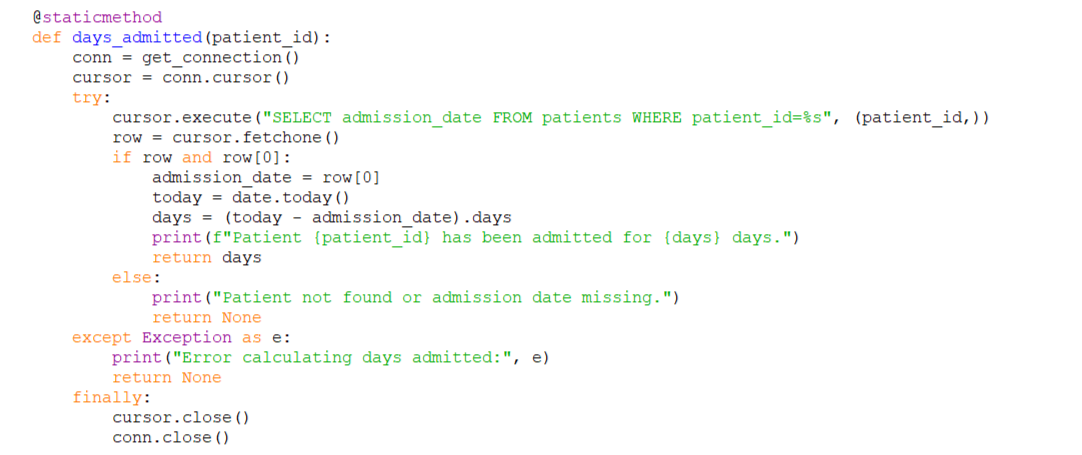
**view()**

Displays all patient records in a clean table format using the tabulate library.



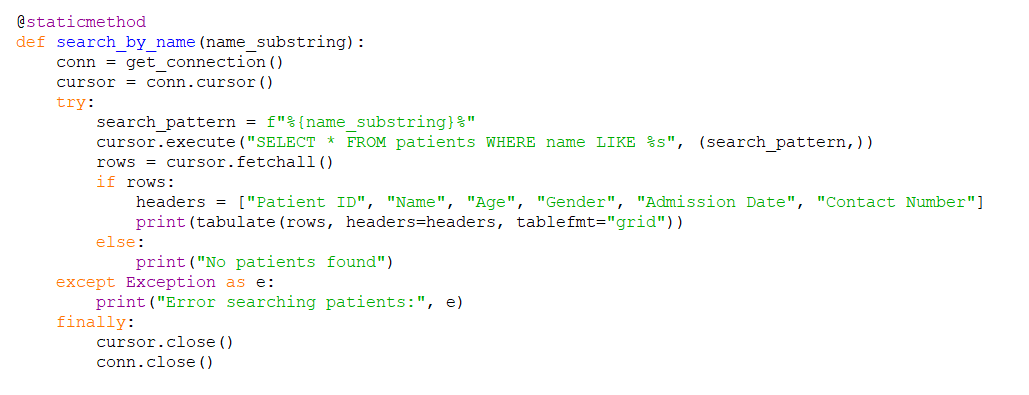
**days\_admitted(patient\_id)**

Calculates how many days a patient has been admitted by subtracting the admission\_date from the current date.



**search\_by\_name(name\_substring)**

Searches for patients whose name contains the given substring (case-insensitive). Useful for partial name lookups.



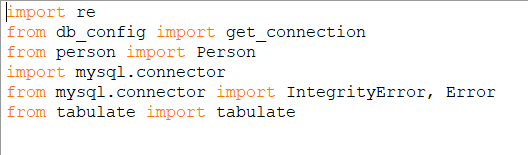
**Error Handling**

* Catches exceptions like IntegrityError, invalid input formats, and connection errors.
* Ensures that database connections and cursors are closed after every operation using finally.

**Doctor Module (doctor.py)**

This module provides functionality to manage doctor records in the Hospital Management System. It includes creating, updating, viewing, deleting, and searching doctor information using a menu-driven CLI.

**Imports**



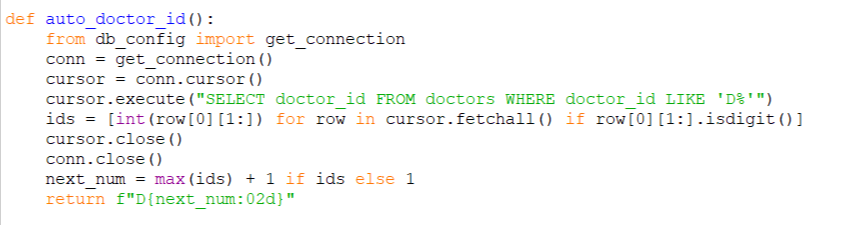
* **get\_connection**: Handles MySQL database connections.
* **Person class**: Base class inherited by Doctor.
* **re**: For validating name and specialization formats.
* **tabulate**: Pretty-prints results in a table format on the CLI.

**Function: auto\_doctor\_id()**

Generates a new unique doctor ID in the format D01, D02, etc.

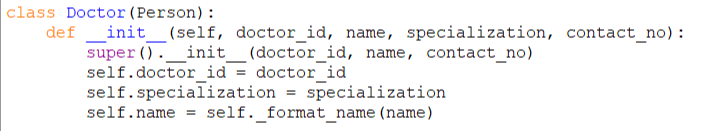
Logic:

* Fetches all doctor IDs starting with D.
* Extracts the numeric part.
* Returns the next available ID formatted with zero padding.



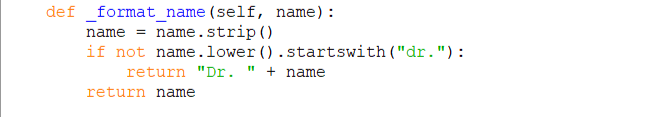
**Class: Doctor**

* Inherits doctor\_id, name, and contact\_no from Person.
* Applies formatting to the name to ensure it begins with "Dr.".
* Stores specialization.



**Private Method: \_format\_name(name)**

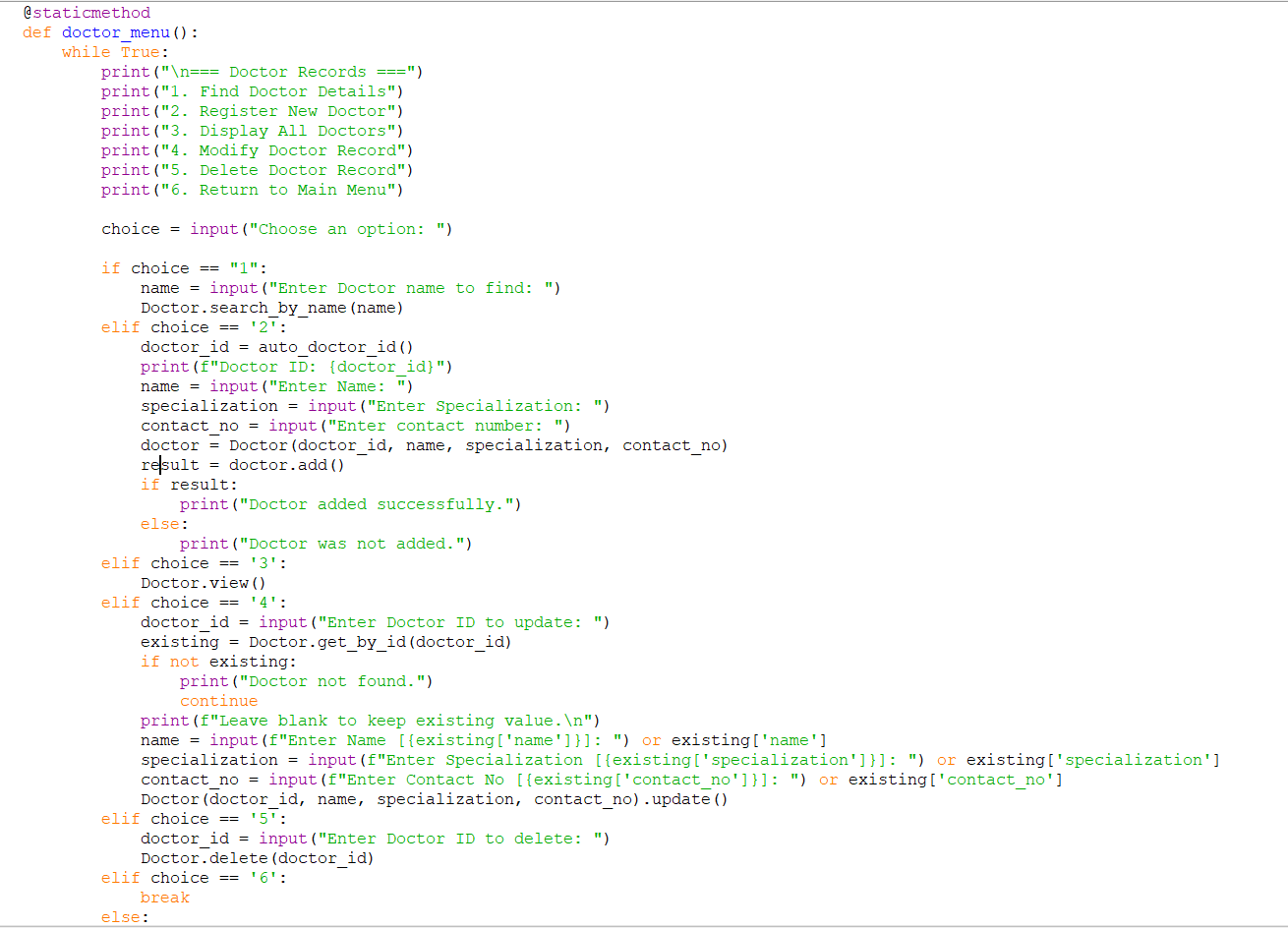
Ensures all doctor names are consistently formatted with "Dr." as a prefix.

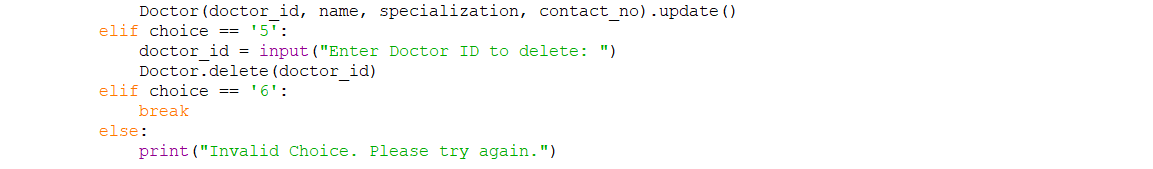


**CLI Interface: doctor\_menu()**

Provides a user-driven interface to interact with the doctor module.

| **Option** | **Action** |
| --- | --- |
| 1 | Search for doctor by name |
| 2 | Add/register a new doctor |
| 3 | View all doctors |
| 4 | Modify doctor details |
| 5 | Delete a doctor by ID |
| 6 | Return to main menu |



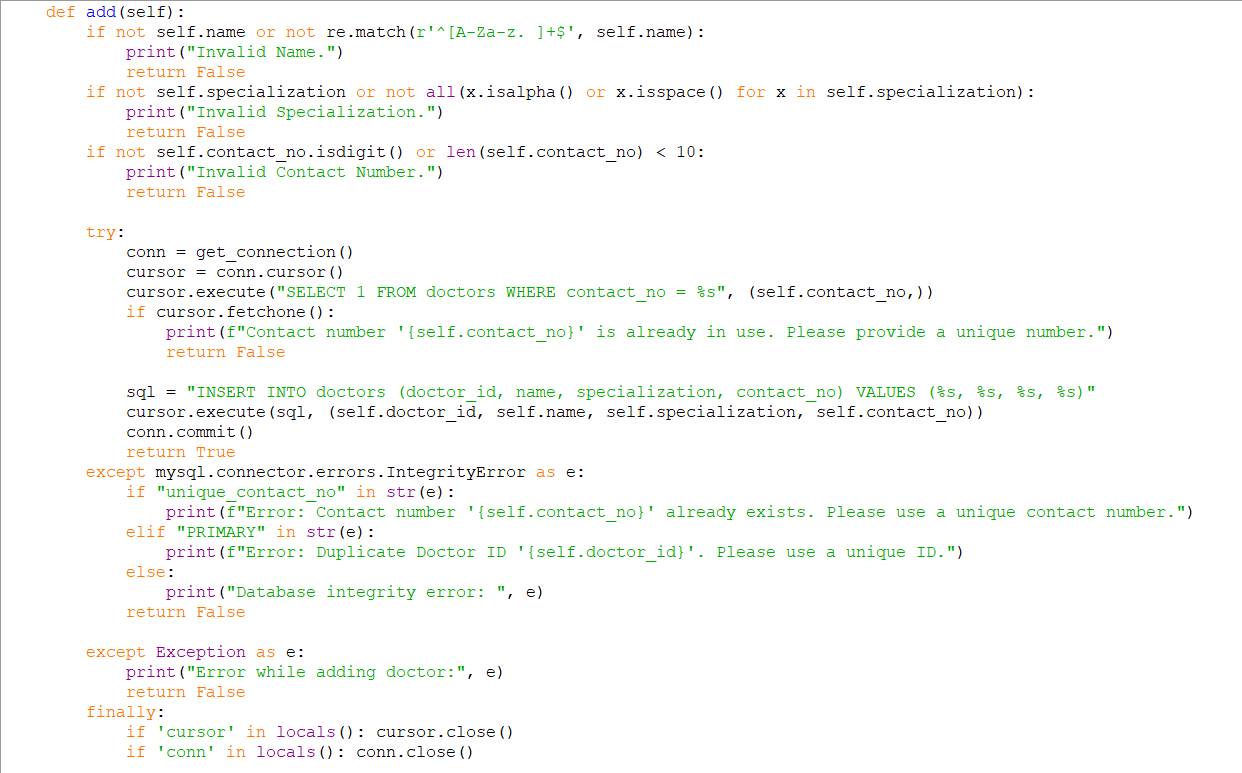


**add(self)**

Inserts a new doctor record into the database after validation.

**Validations:**

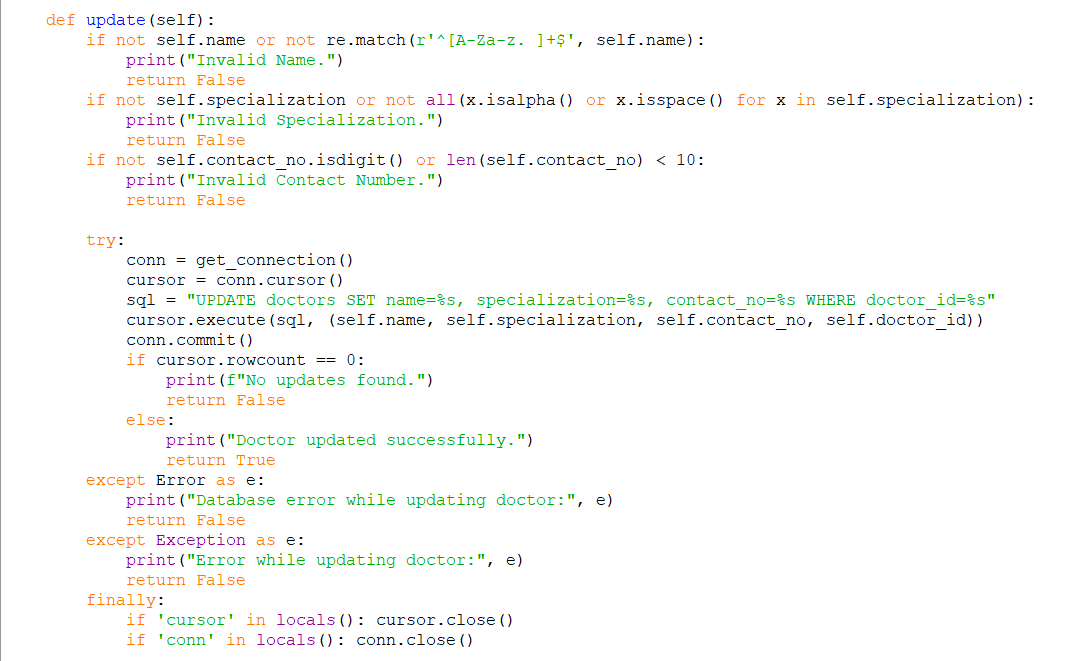
* **Name**: Alphabet, dot, or space only.
* **Specialization**: Letters and spaces only.
* **Contact number**: Must be digits and at least 10 characters.
* Checks for unique contact number in the database.



**update(self)**

Updates an existing doctor record based on doctor\_id.

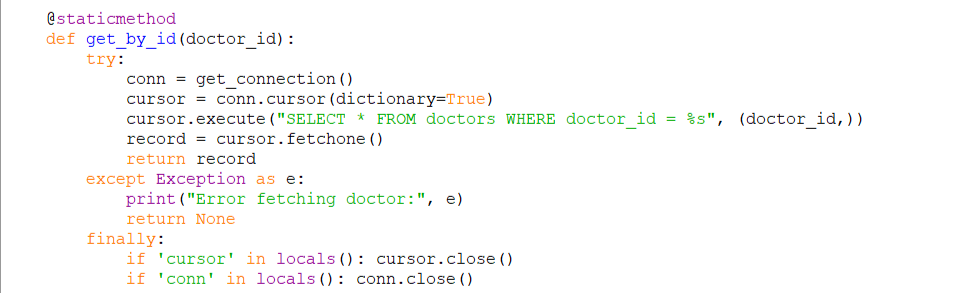
* Performs similar validations as add.
* Uses an UPDATE query to modify doctor data.
* Confirms update success via rowcount.



**get\_by\_id(doctor\_id)**

Fetches a doctor record using the given doctor ID.

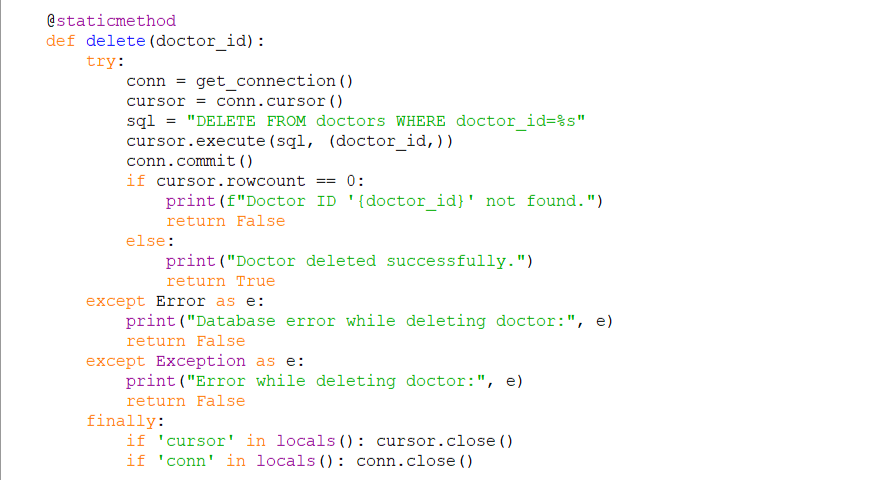
* Returns a dictionary of the record or None if not found.



**delete(doctor\_id)**

Deletes a doctor from the database using the doctor\_id.

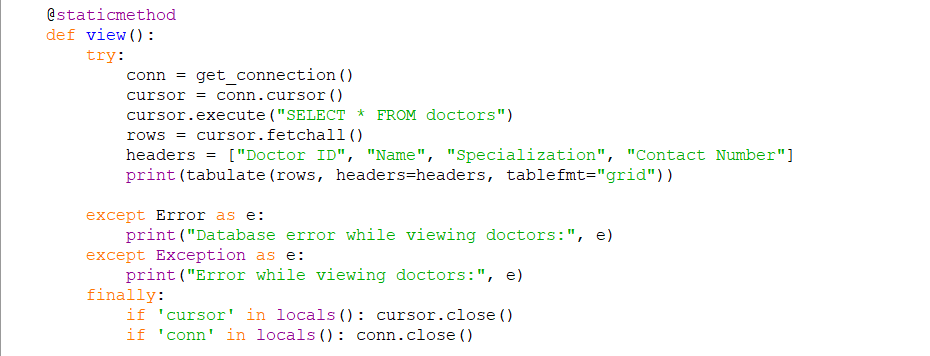
* Confirms deletion based on affected rows.



**view()**

Displays all doctor records in a tabular format using tabulate.

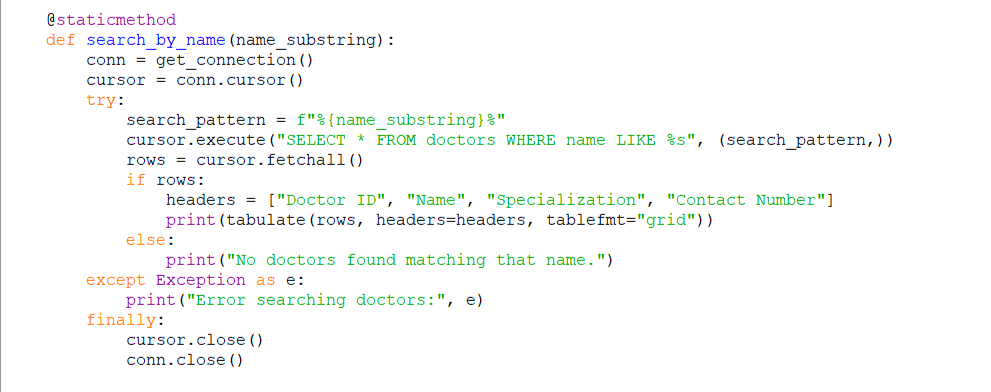
* Columns: Doctor ID, Name, Specialization, Contact Number.



**search\_by\_name(name\_substring)**

Searches for doctors whose names match or contain the input substring.

* Uses SQL LIKE pattern match.
* Displays results in tabular format.

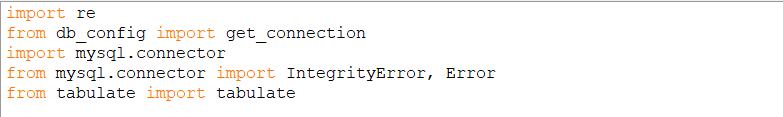


**Service Module (service.py)**

The code is a Python module designed to manage services in a Hospital Management System using a MySQL database. It handles:

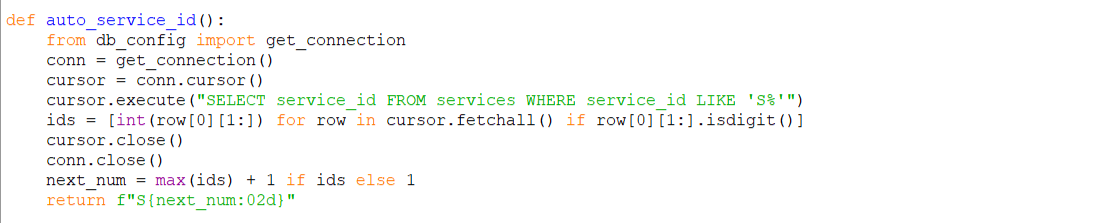
* Registering and managing services (like tests, procedures).
* Assigning those services to patients.
* Viewing, updating, and deleting service records.
* Temporary service usage tracking for billing or treatment history.

**Imports**



* re: Used for regex-based input validation.
* get\_connection: Imports a function from db\_config to connect to the MySQL database.
* mysql.connector: MySQL database driver.
* IntegrityError, Error: Specific exceptions for handling database errors.
* tabulate: For formatting outputs in table format.

**Method auto\_service\_id() Function**

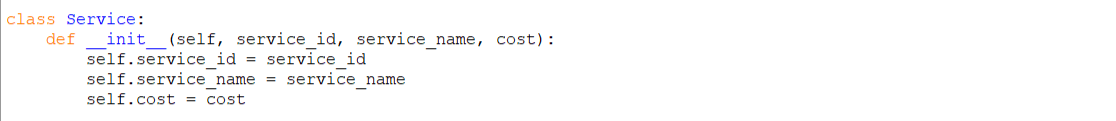


Auto-generates a new service ID (like S01, S02, …).

* Fetch all service\_ids starting with S.
* Strip the S, convert the numeric part to an integer.
* Find the max number and add 1.
* Return the new ID as SXX.

**Service Class**

Manages all operations related to hospital services.

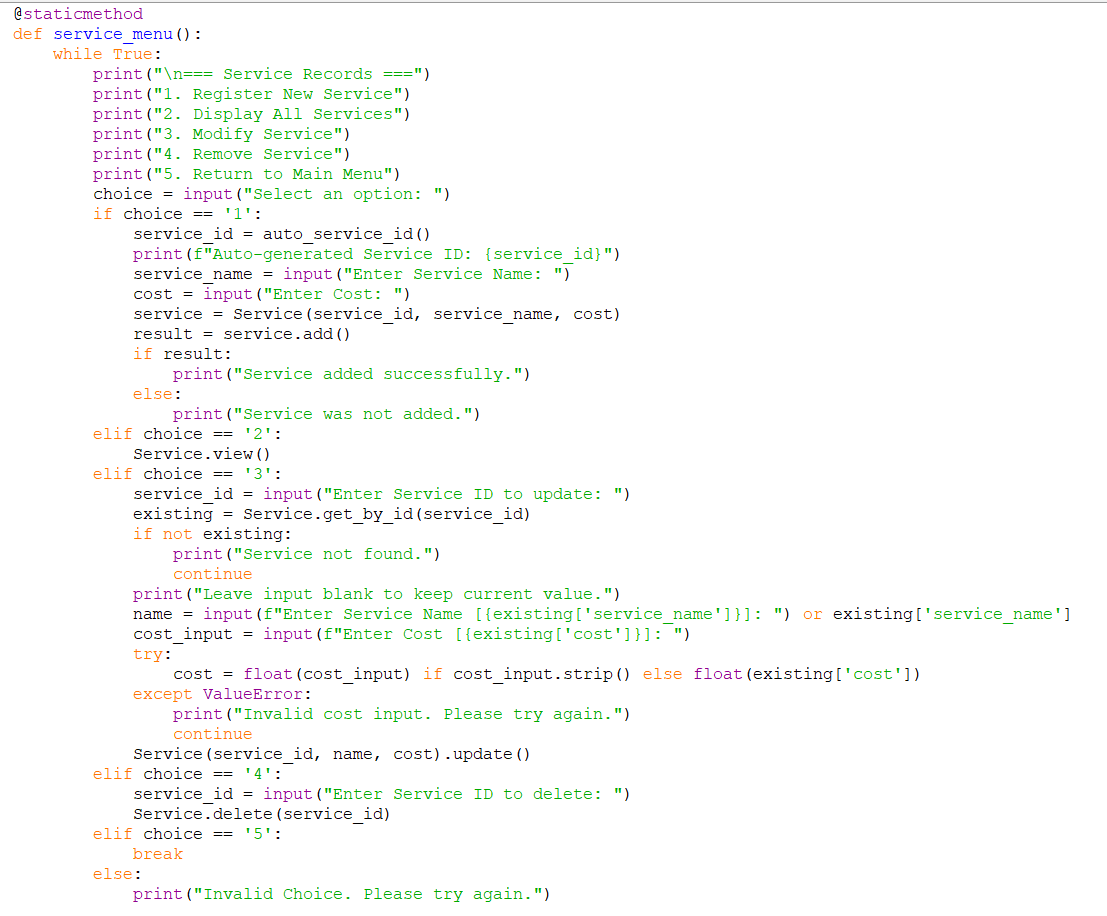


**service\_menu() – CLI Menu**

Allows the user to:

1. Register a new service.
2. View all services.
3. Modify existing services.
4. Delete a service.
5. Exit menu.

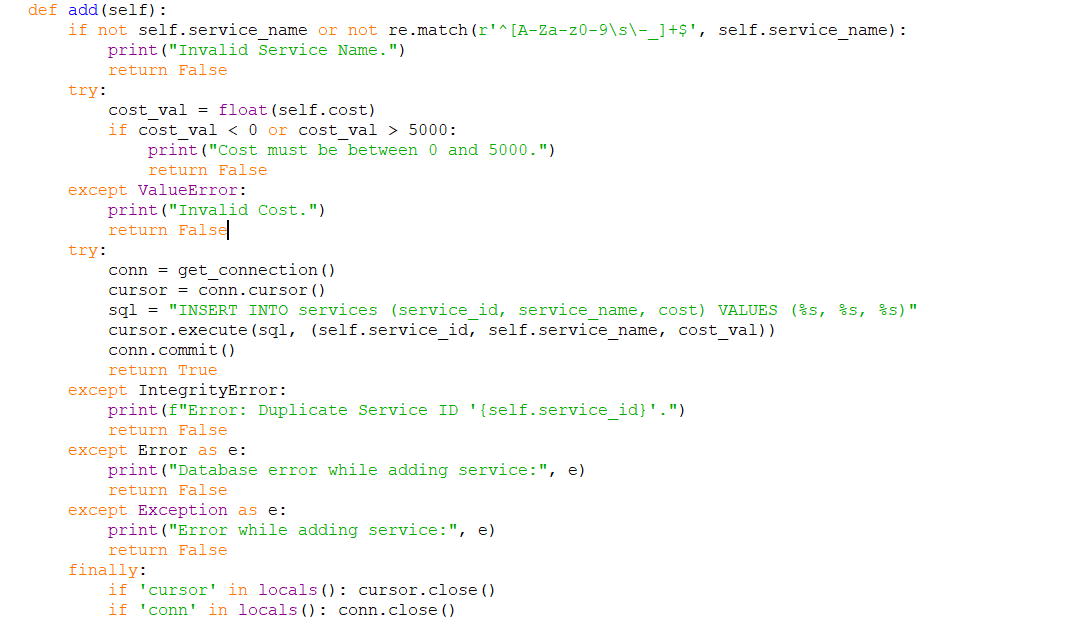
This is a user interface loop using standard input and output.



**add(self)**

Adds a new service to the database.

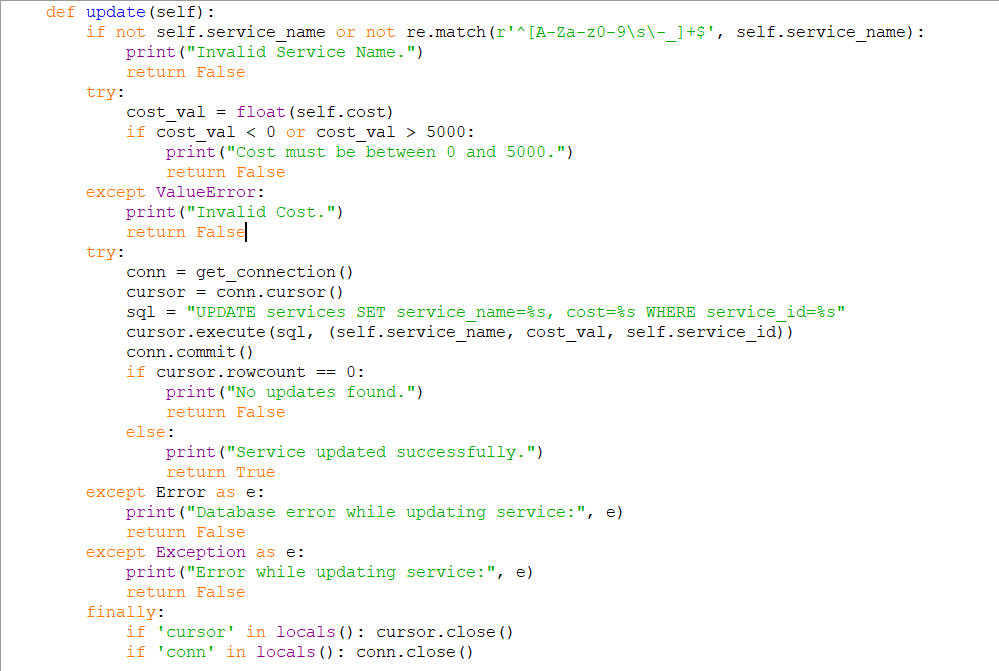
* Validates:
  + service\_name must match allowed characters (alphanumeric, hyphen, underscore).
  + cost must be numeric and within 0–5000.
* If validation passes:
  + Inserts record into services table.
* Handles exceptions:
  + Duplicate IDs, invalid cost, database errors.



**update(self)**

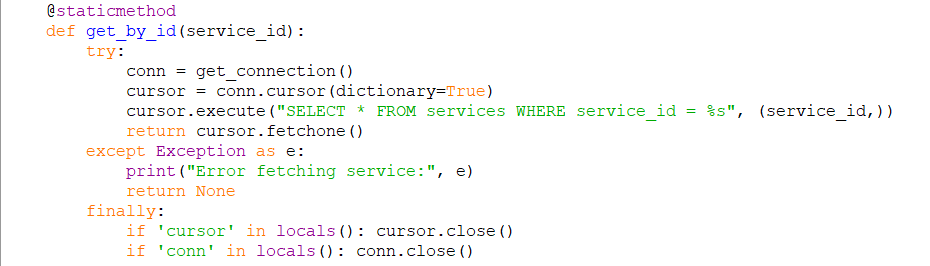
Updates an existing service.

* Validates updated name and cost.
* Updates the record where service\_id matches.
* Displays success/failure.



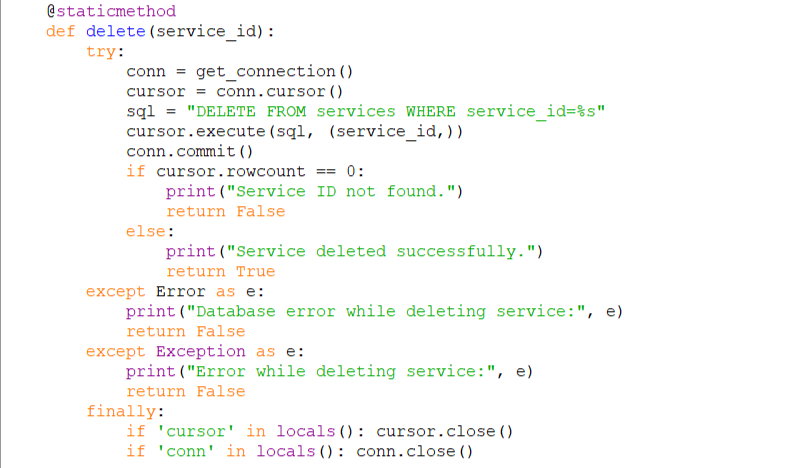
**get\_by\_id(service\_id)**

* Fetches a service row by service\_id as a dictionary using cursor(dictionary=True).
* Used during modification to retrieve current values.



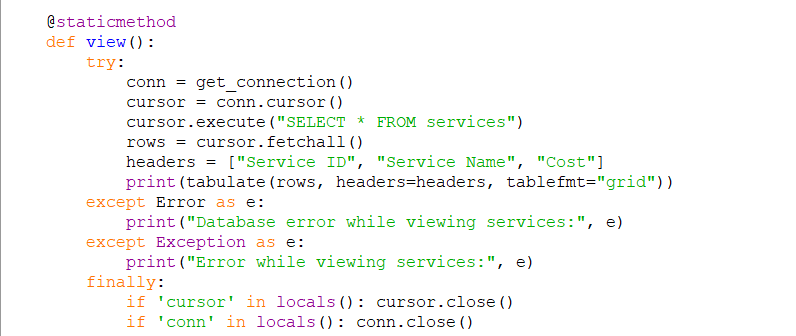
**delete(service\_id)**

* Deletes a service from the services table by ID.
* Checks cursor.rowcount to confirm if deletion occurred.



**view()**

* Displays all services using tabulate() for readability.



**ServiceUsageDB Class**

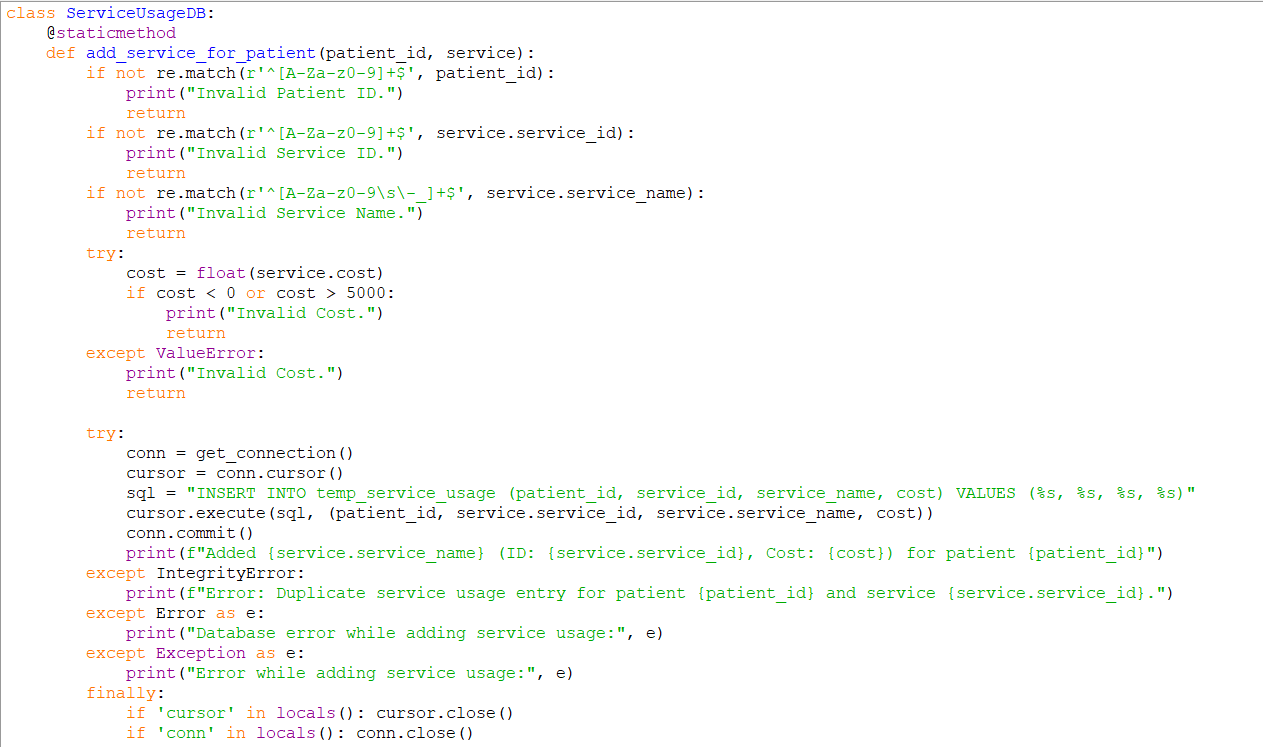
Handles tracking of which services have been used by which patients. These are **temporary records**, probably used before billing.

**Method add\_service\_for\_patient(patient\_id, service)**

**Function**: Adds a service for a given patient.

* Validates patient ID and service fields.
* Inserts record into temp\_service\_usage.

Used during patient treatment for service logging.

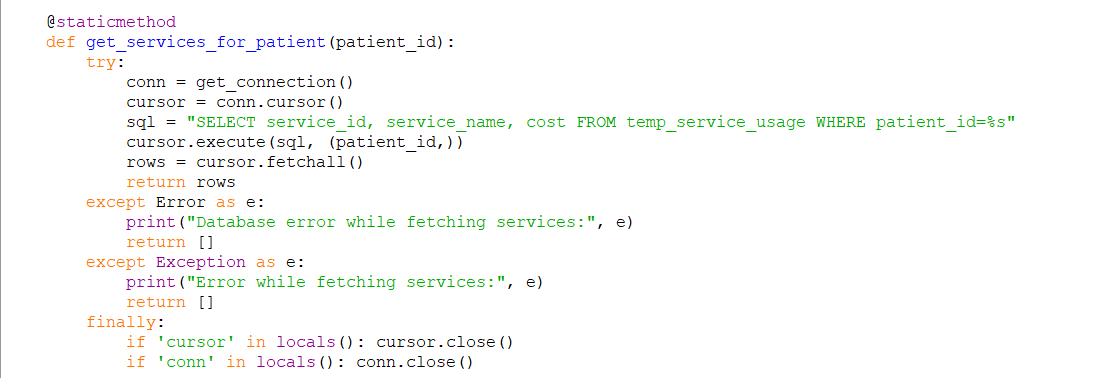


**get\_services\_for\_patient(patient\_id)**

Returns a list of all services used by a patient from temp\_service\_usage.

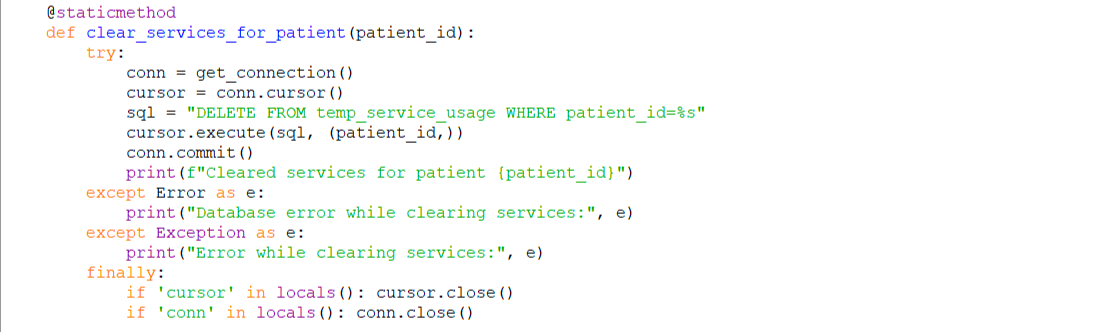
Useful for:

* Viewing patient billing summary.
* Checking services used during treatment.



**clear\_services\_for\_patient(patient\_id)**

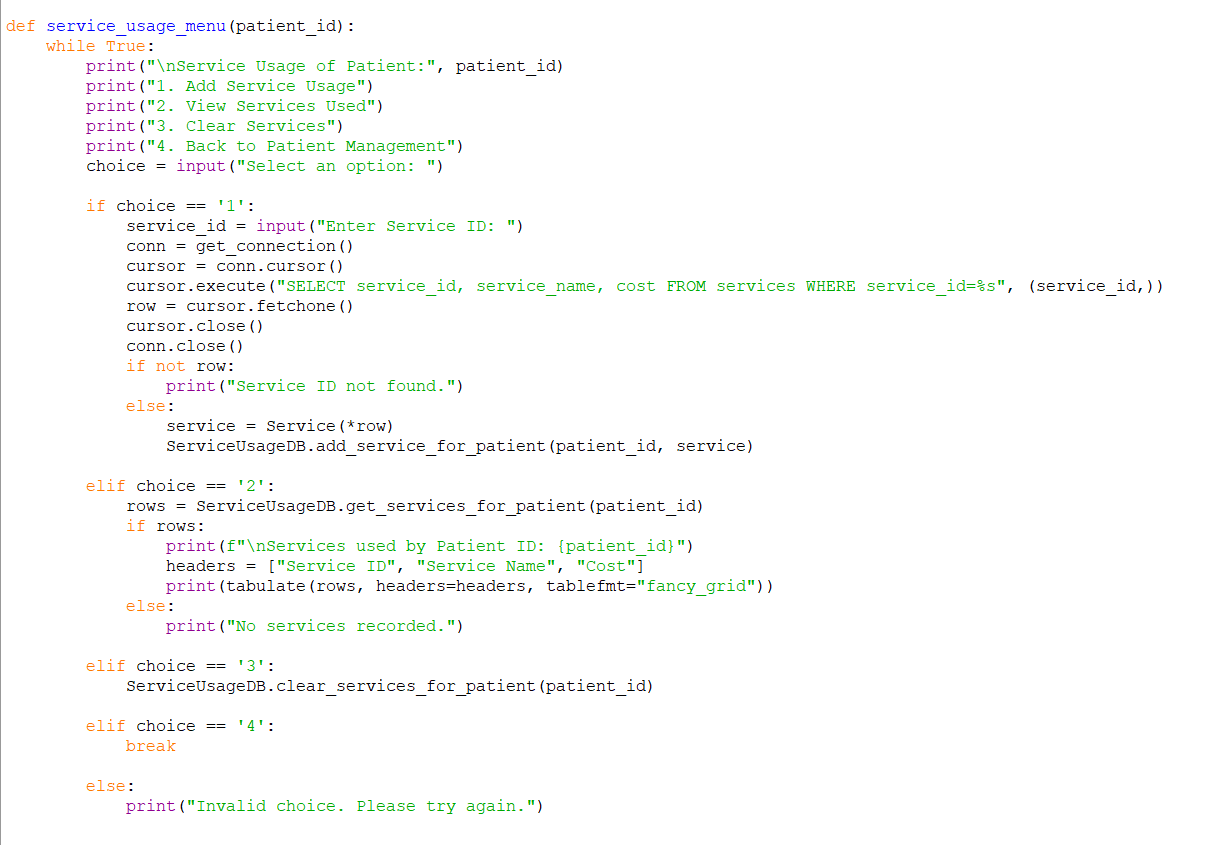
Deletes all temp service records for a patient. Usually done after billing is finalized.



**service\_usage\_menu(patient\_id)**

Menu to manage services used by a patient:

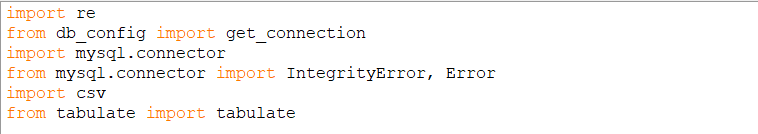
1. **Add Service Usage** – Prompts for a service ID, fetches service data, and logs it.
2. **View Services Used** – Shows a formatted table of services.
3. **Clear Services** – Removes all services used by a patient.
4. **Back to Patient Menu** – Exits.



**Appointment Module (appointment.py)**

This Appointment class in Python is part of a Hospital Management System. It handles appointment scheduling, modification, deletion, viewing, searching, CSV exporting, and date-based operations using MySQL as the backend and OOP principles in Core Python.

**Imports**

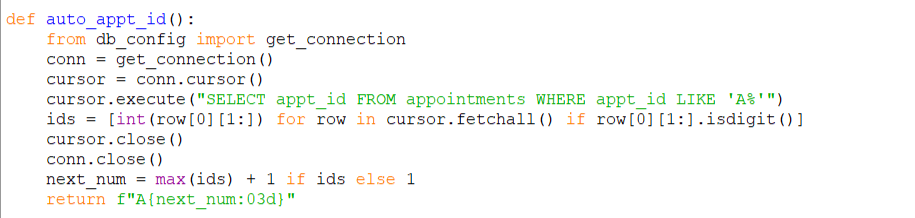


* re: For regular expressions (used in validating the date format).
* get\_connection: Custom method from db\_config.py to get a connection to the MySQL database.
* mysql.connector: Python library to interact with MySQL.
* csv: For exporting data to CSV files.
* tabulate: For displaying tabular data in a pretty format in the console.

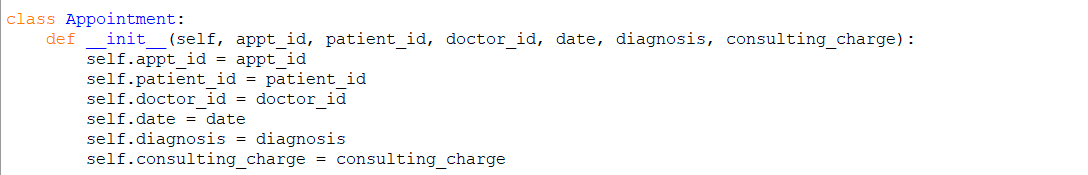
**auto\_appt\_id() Function**

Automatically generate a new unique appointment ID in format A001, A002, etc.

* Fetch all IDs starting with 'A'.
* Extract the numeric part, convert it to int.
* Increment the maximum number and return formatted ID like A004.



**Class Appointment**

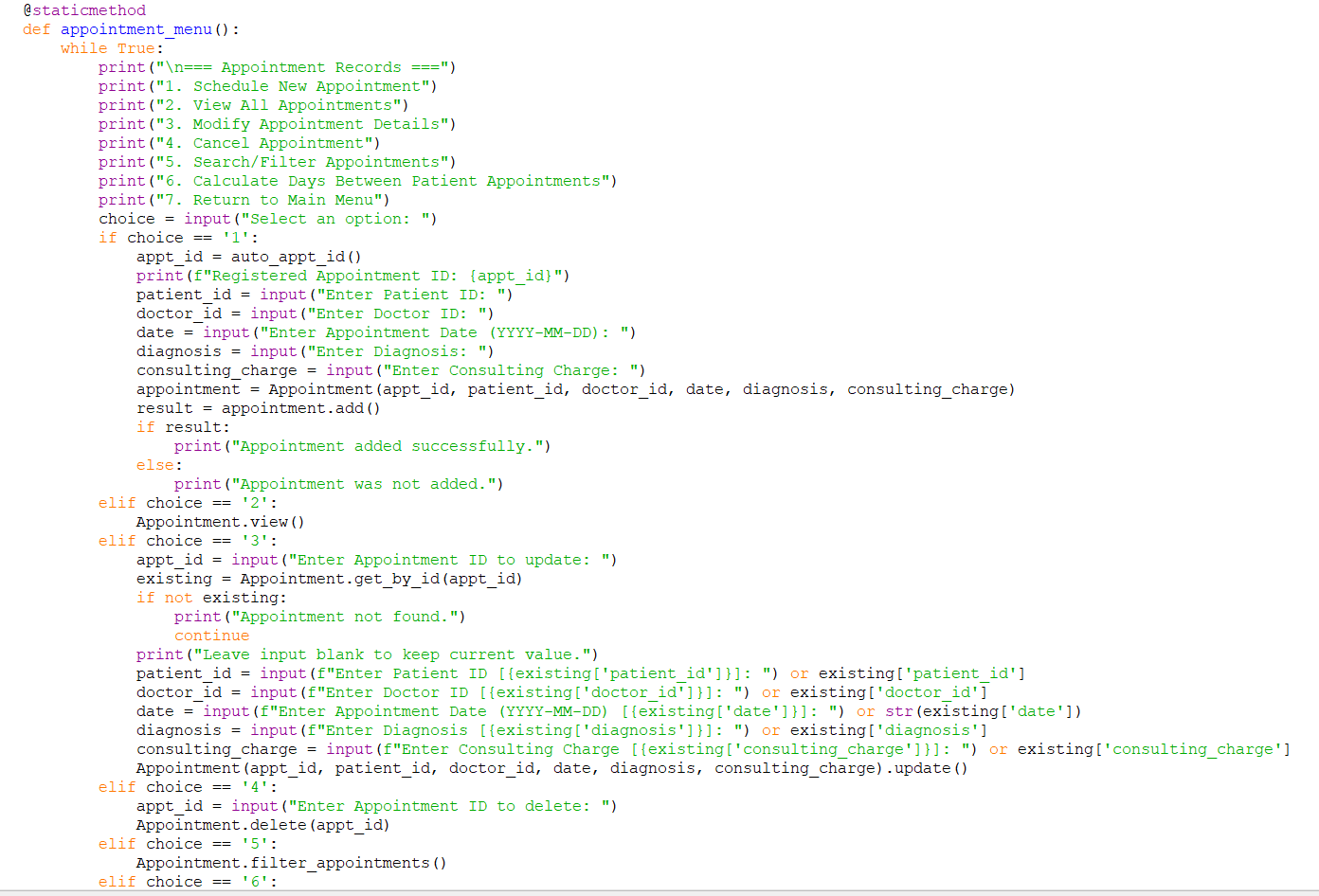
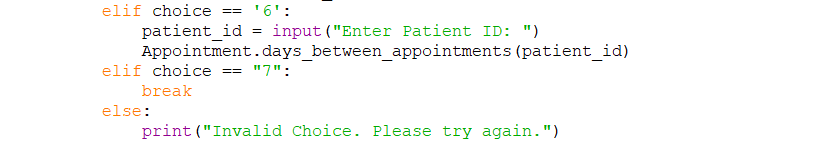


This encapsulates all functionality related to appointment records. Stores appointment details as instance variables.

**appointment\_menu() :**

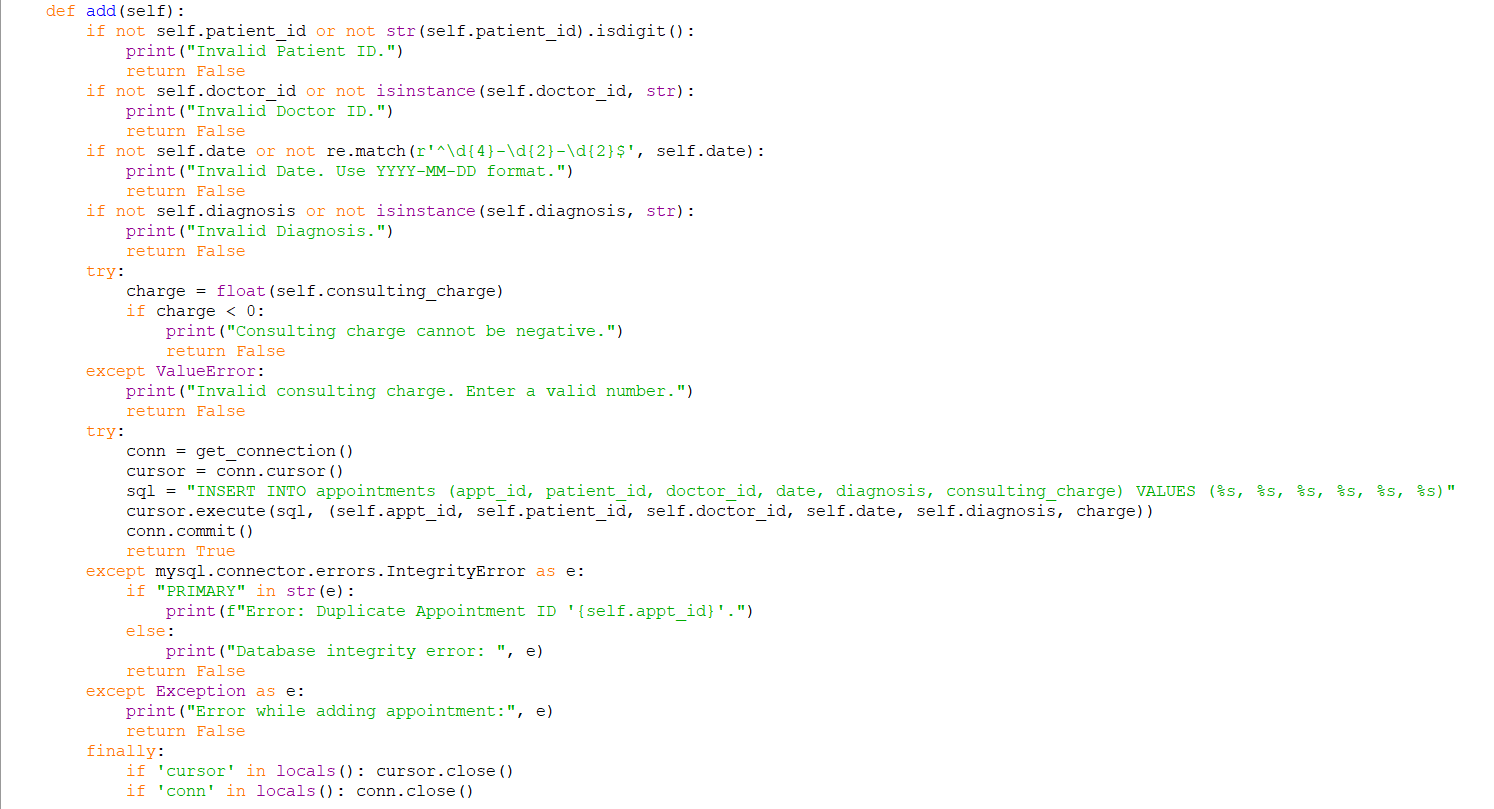
Menu-driven interface to access appointment features.

* Schedule a new appointment.
* View all appointments.
* Modify an appointment.
* Cancel an appointment.
* Search/filter appointments by date range.
* Calculate days between a patient's appointments.
* Exit.

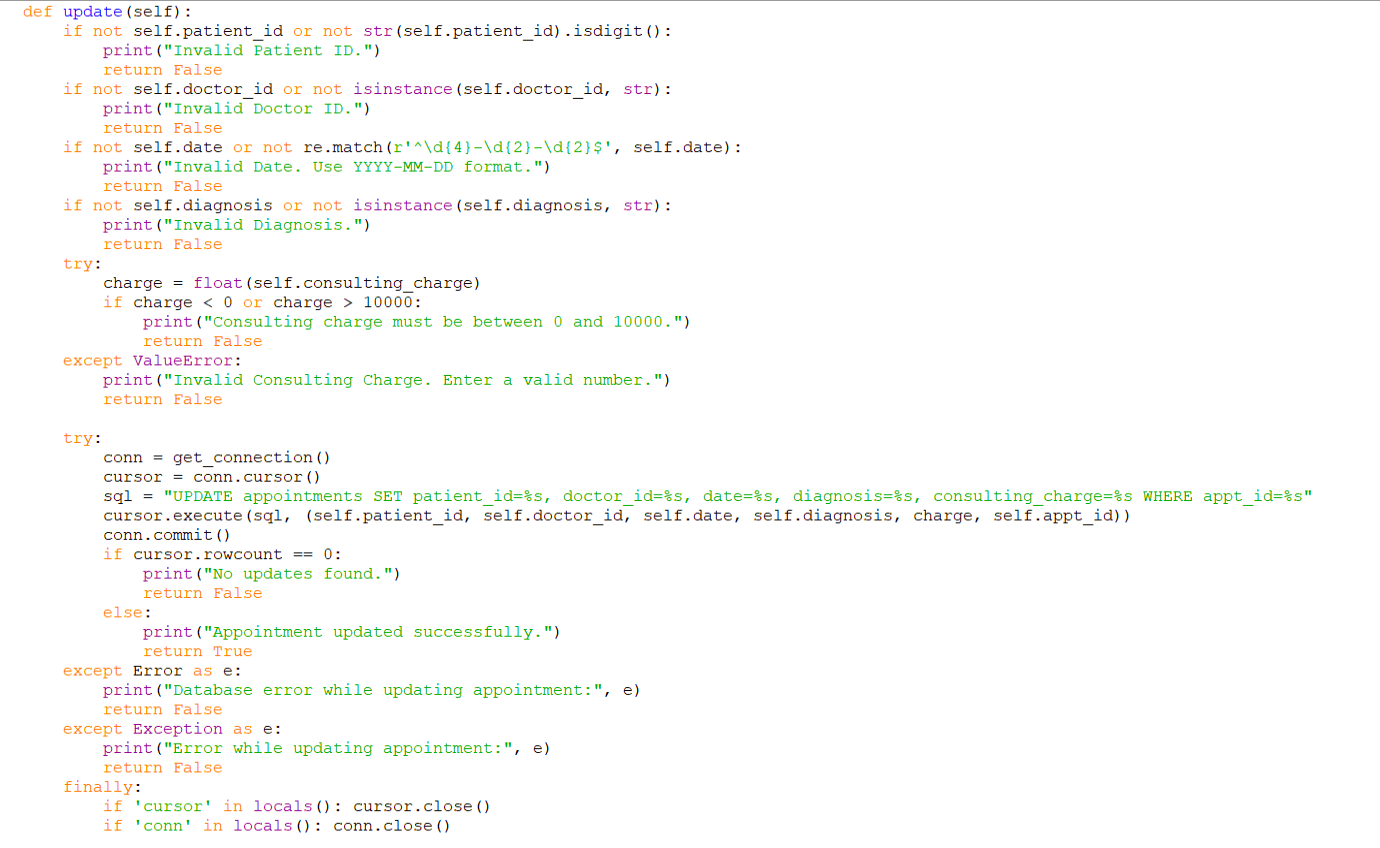


**add()**

* Validates inputs: Checks for empty/invalid patient ID, doctor ID, date format, diagnosis, charge.
* Inserts a new row into the appointments table.
* Handles errors:
  + Duplicate ID.
  + Database error.

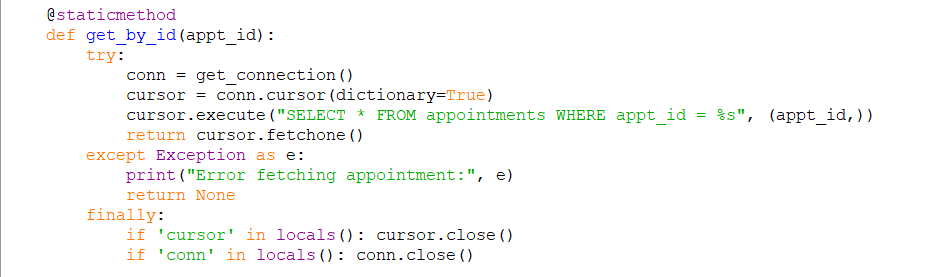


**update()**

* Similar to add(), but updates existing appointment details.
* SQL: UPDATE appointments SET ... WHERE appt\_id = %s
* Returns success or error message.

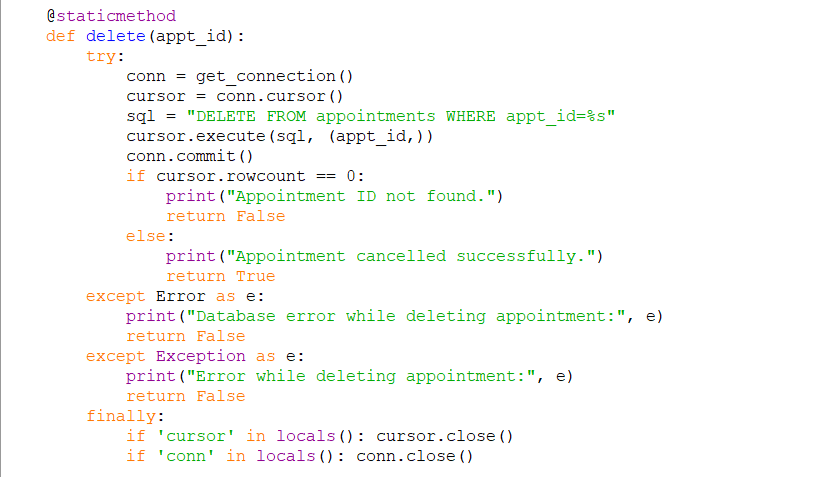
**get\_by\_id()**

Fetches appointment row by ID and returns it as a dictionary (thanks to dictionary=True).



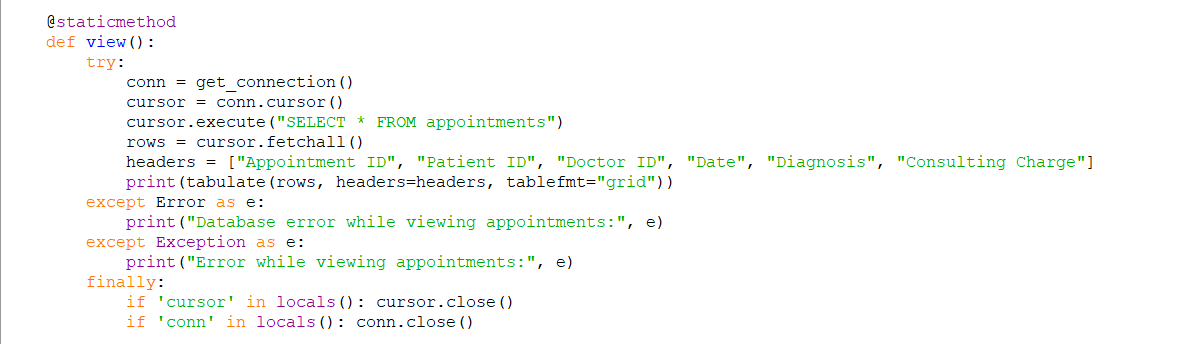
**delete()**

* Deletes appointment by ID.
* Returns whether the operation was successful.



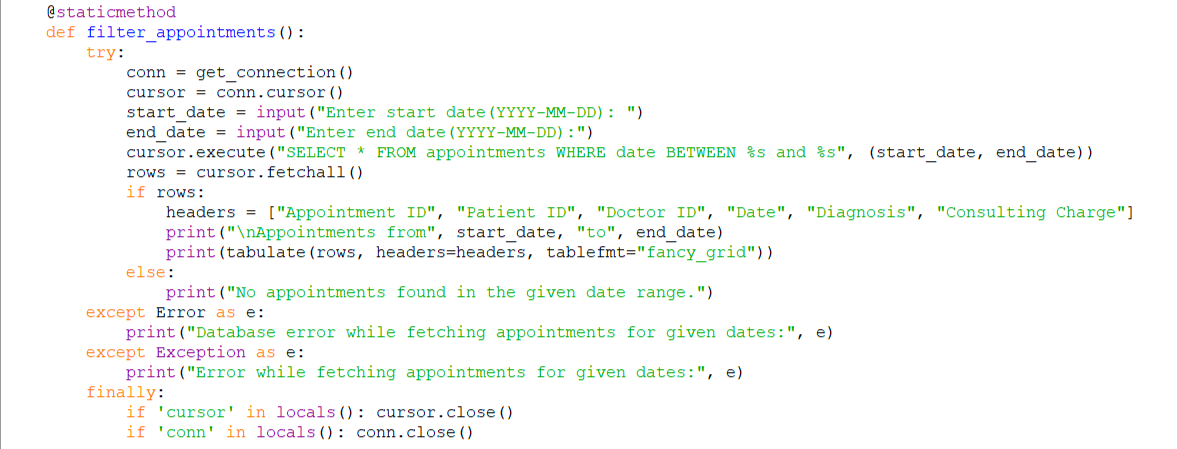
**view()**

Fetches and displays all appointments using tabulate().



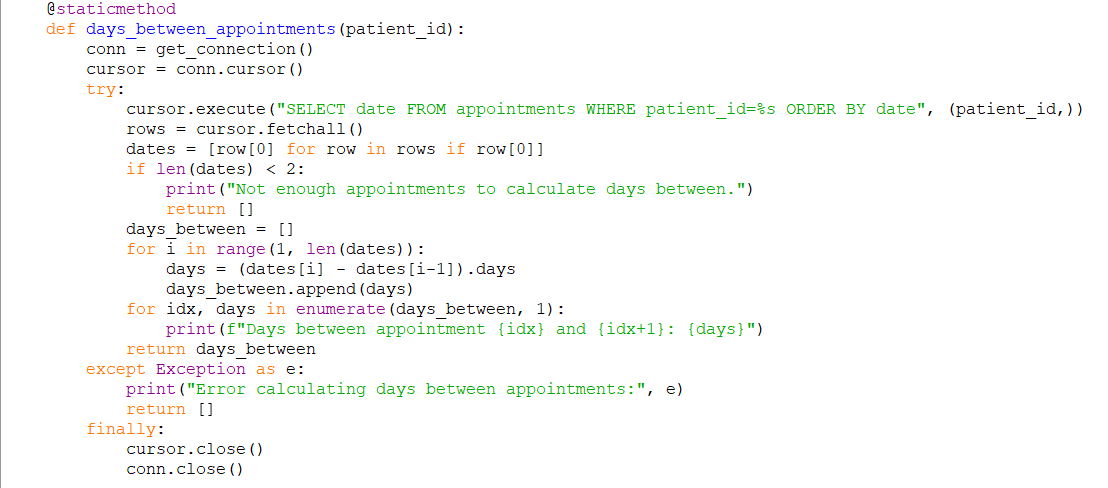
**filter\_appointments()**

* User inputs a date range.
* Fetches appointments between those dates.
* Displays them in a fancy table format.



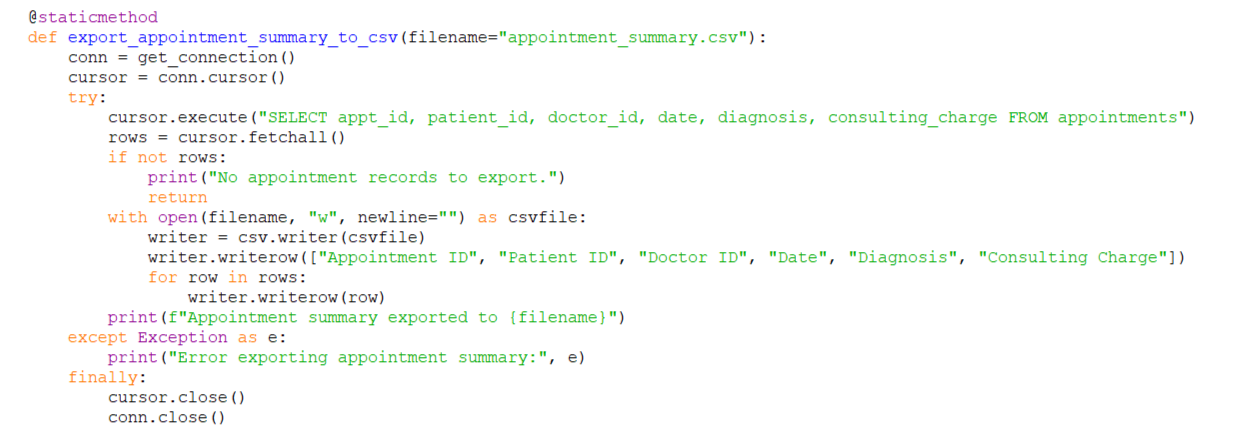
**days\_between\_appointments()**

* Calculates and prints the number of days between each pair of sequential appointments for a patient.
* Returns a list of days differences.



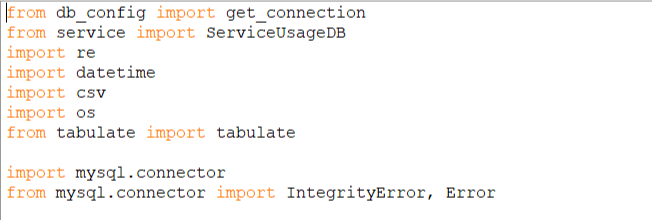
**export\_appointment\_summary\_to\_csv()**

* Writes all appointments to a CSV file.
* Headers: Appointment ID, Patient ID, Doctor ID, Date, Diagnosis, Consulting Charge.



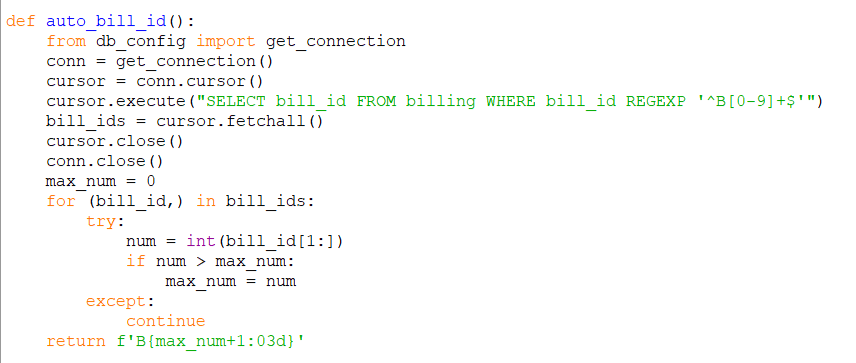
**Billing Module (billing.py)**

This module handles bill creation, modification, deletion, viewing, invoice generation, and exporting billing summaries. It also calculates total charges per patient by aggregating services and consultation fees.



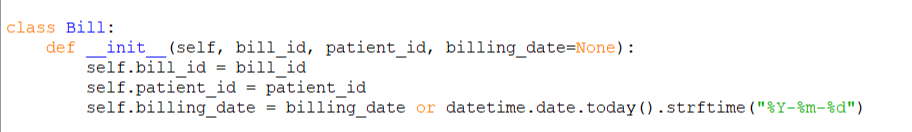
* **Database Access**: get\_connection() establishes a connection to the MySQL database.
* **ServiceUsageDB**: Temporary service usage logic for billing.
* **tabulate**: For formatted CLI output.
* **CSV & OS**: For file export and invoice creation.
* **Regex & datetime**: For validation and date processing.

**auto\_bill\_id()**

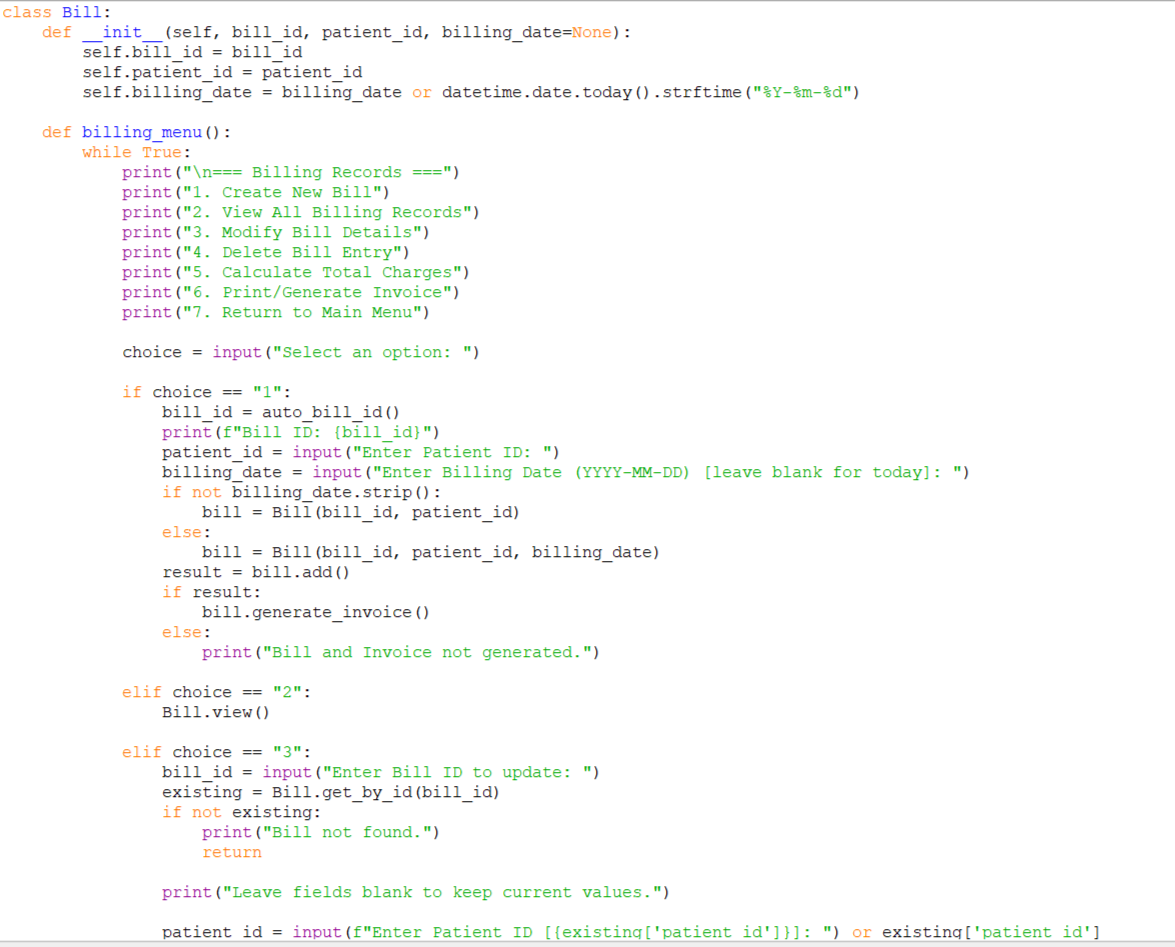


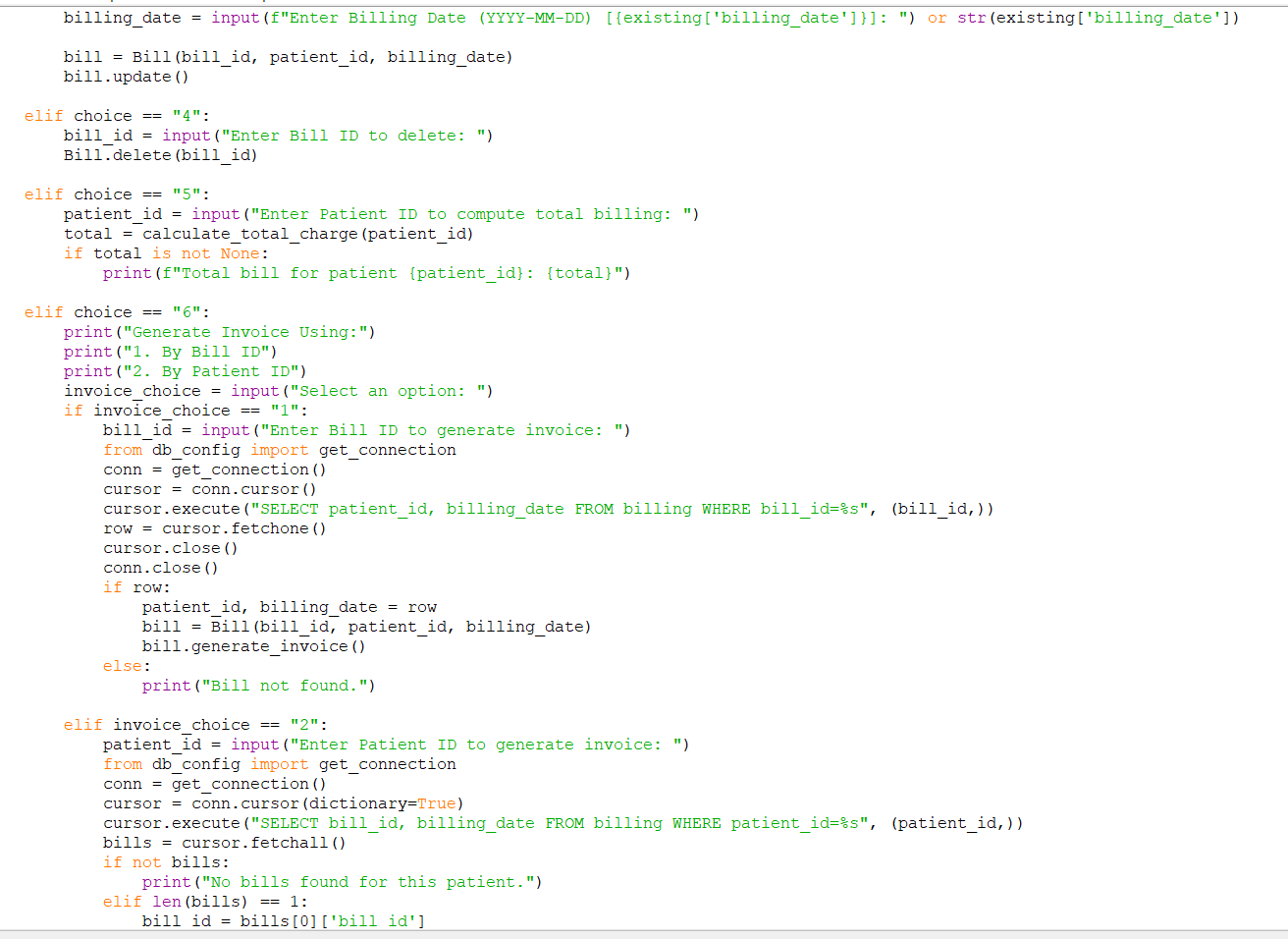
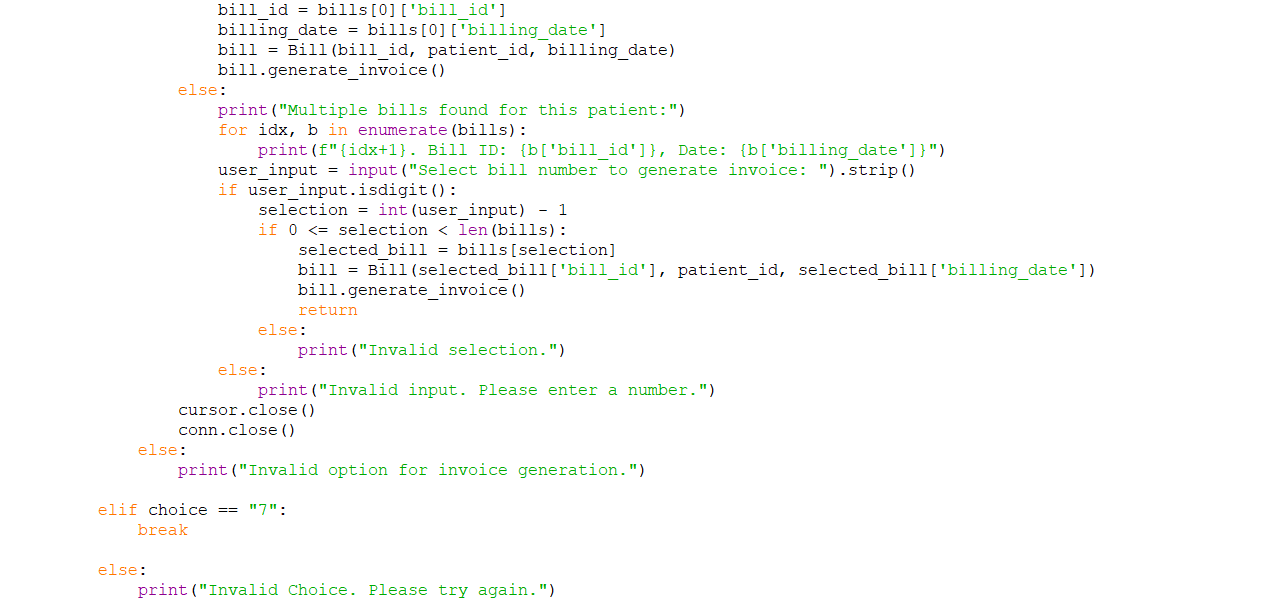
Generates a unique bill ID in the format B001, B002, ..., based on the max bill\_id found in the billing table using a REGEXP.

**Bill Class**



**billing\_menu() – Main CLI Interface**





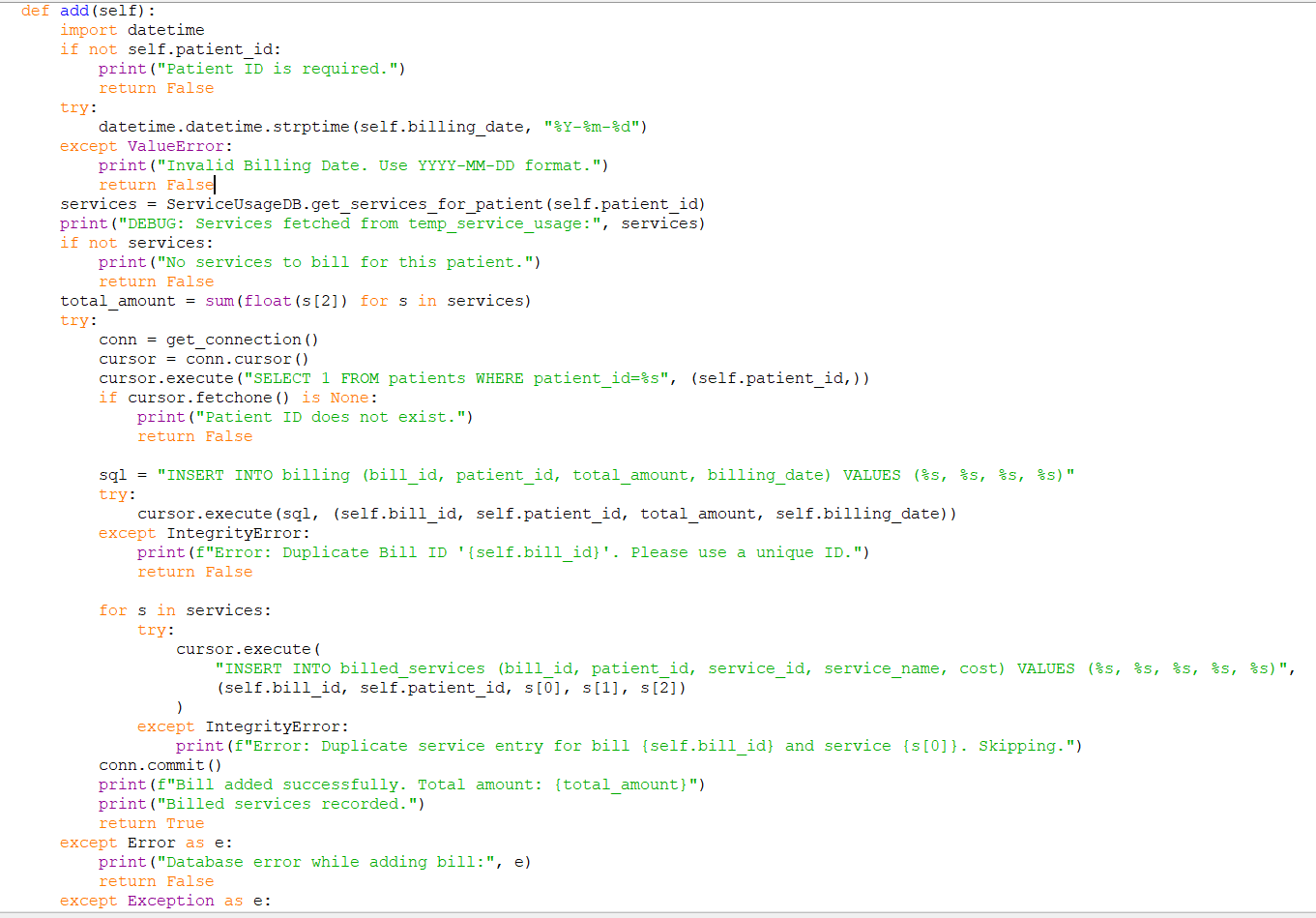
A static menu-driven CLI to handle billing tasks:

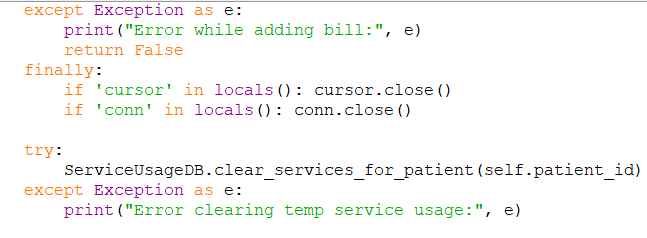
| **Option** | **Action** |
| --- | --- |
| 1 | Create new bill and generate invoice |
| 2 | View all billing records (tabulated) |
| 3 | Modify bill details |
| 4 | Delete bill entry |
| 5 | Calculate total billing (services + consulting) |
| 6 | Generate invoice (by bill ID or patient ID) |
| 7 | Exit menu |

**add(self)**

Adds a bill entry after validating and fetching service costs.

* Validates patient\_id and billing\_date.
* Fetches services used by patient from temp\_service\_usage.
* Calculates total and inserts into:
  + billing table.
  + billed\_services table (item-wise).
* Commits transaction and clears the temporary service table.

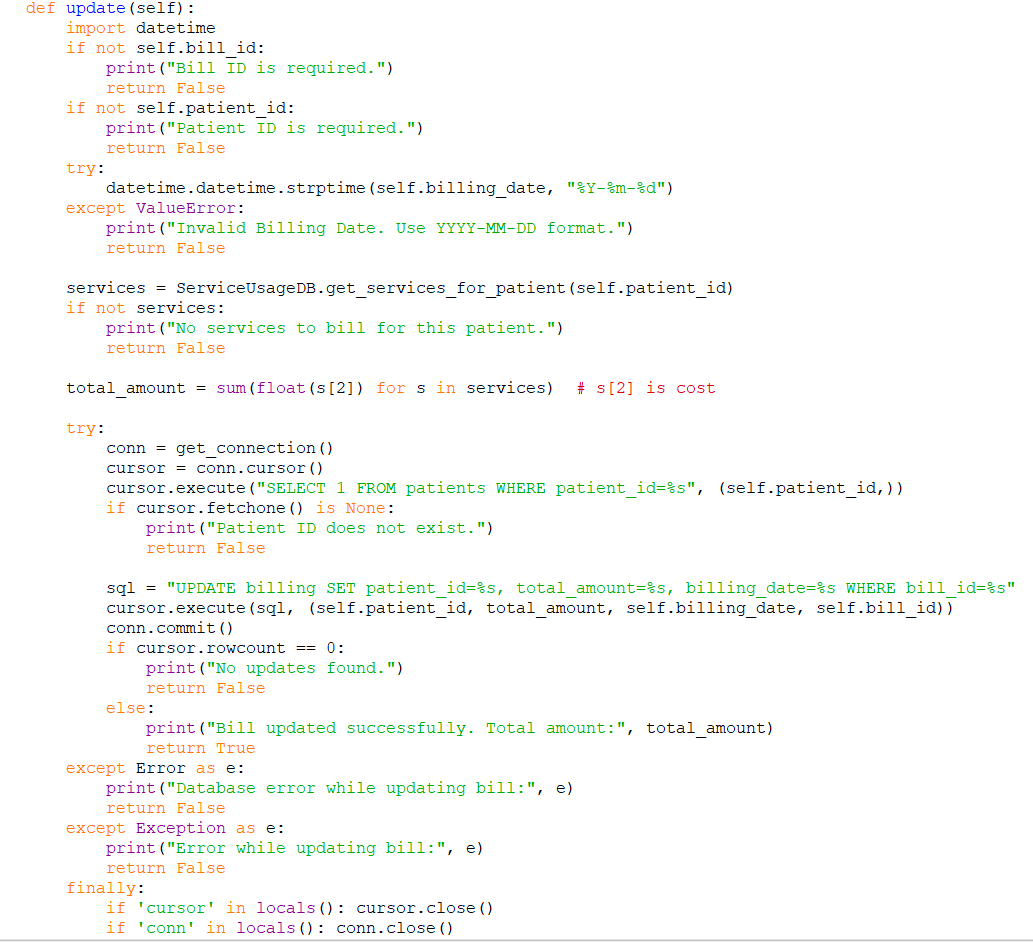


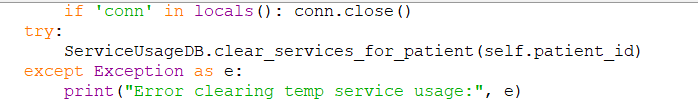


**update(self)**

Updates an existing bill:

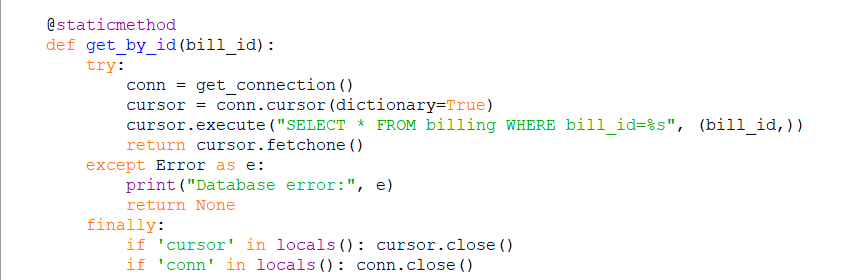
* Recalculates total charges from current services.
* Updates the billing table.
* Clears and reinserts billed services (if required).





**get\_by\_id(bill\_id)**

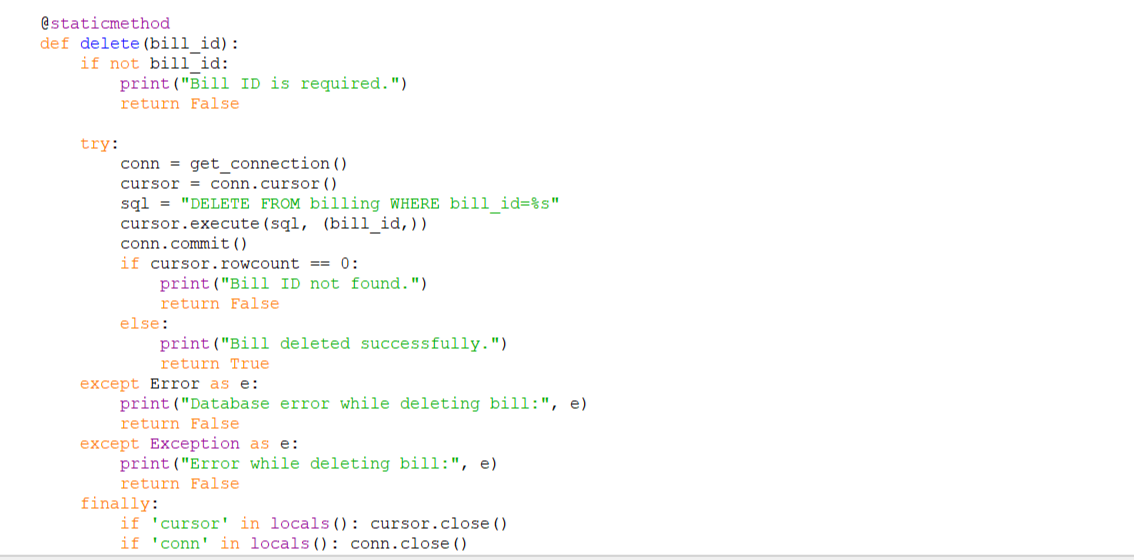
* Fetches a single bill record from the database using the bill\_id.



**delete(bill\_id)**

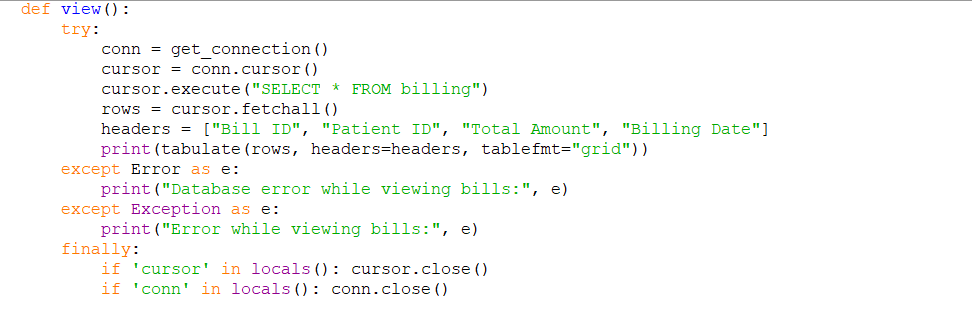
Deletes a billing record from the database.

* Validates the bill ID.
* Deletes the record from the billing table.



**view()**

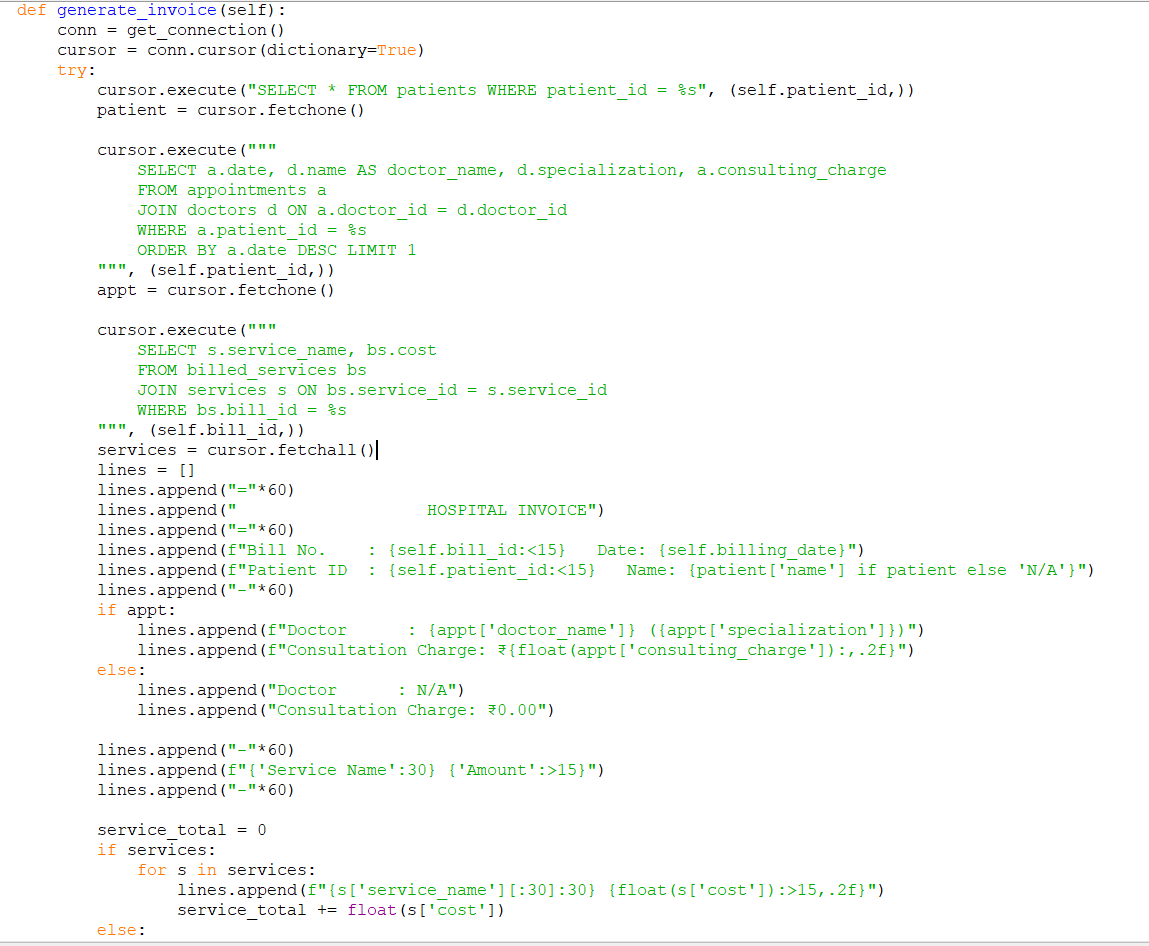
* Displays all bill records in a formatted table using tabulate

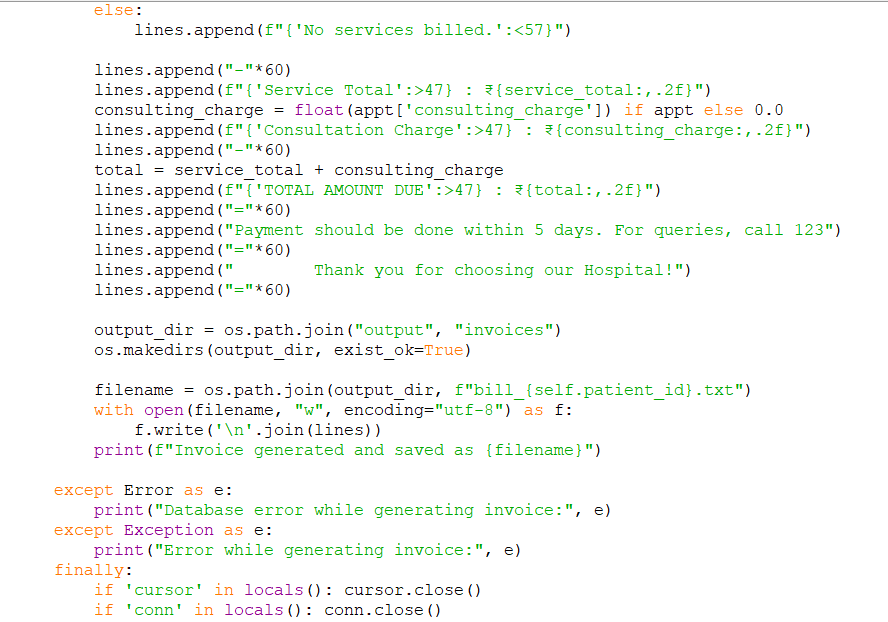


**generate\_invoice(self)**

Creates a text invoice with patient, doctor, and service details:

* Fetches:
  + Patient name.
  + Latest doctor appointment and charge.
  + Services associated with the bill.
* Calculates totals and writes to a .txt file inside output/invoices/.
* Invoice format includes:
  + Hospital heading.
  + Bill and patient details.
  + Doctor consultation details.
  + Itemized service costs.
  + Total due amount.

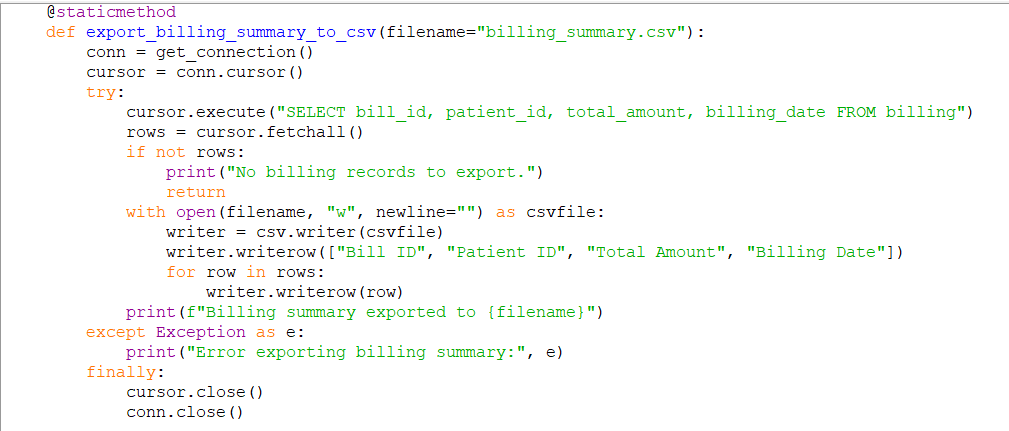




**export\_billing\_summary\_to\_csv(filename)**

Exports all billing data to a CSV file with columns:

* Bill ID
* Patient ID
* Total Amount
* Billing Date

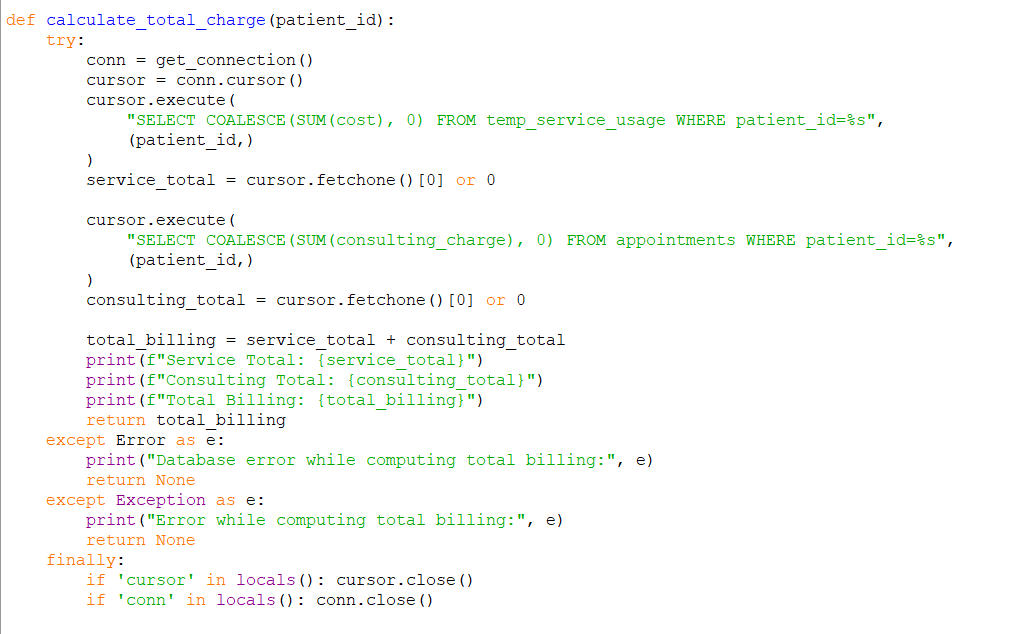


**calculate\_total\_charge(patient\_id)**

Calculates the total bill amount for a patient by summing:

* Services used (from temp\_service\_usage).
* Consultation charges (from appointments).

Outputs the intermediate and final totals for clarity.



**Import CSV module (import\_csv.py)**

Using Python along with the pandas and mysql.connector libraries to import multiple CSV files into a MySQL database called “hospitalmanagementsystem”

1. **Import Required Modules**

* pandas: For reading and processing CSV files.
* mysql.connector: To connect and interact with the MySQL database.
* os: To check file paths and get the current working directory.

2. **Print Current Working Directory**

Helps confirm that Python is executing from the expected directory (useful for debugging paths).

3. **Connect to MySQL Database**

* Establishes a connection to the MySQL database using credentials.
* Creates a cursor for executing SQL commands.

4. **Reusable Function: import\_csv\_to\_table()**

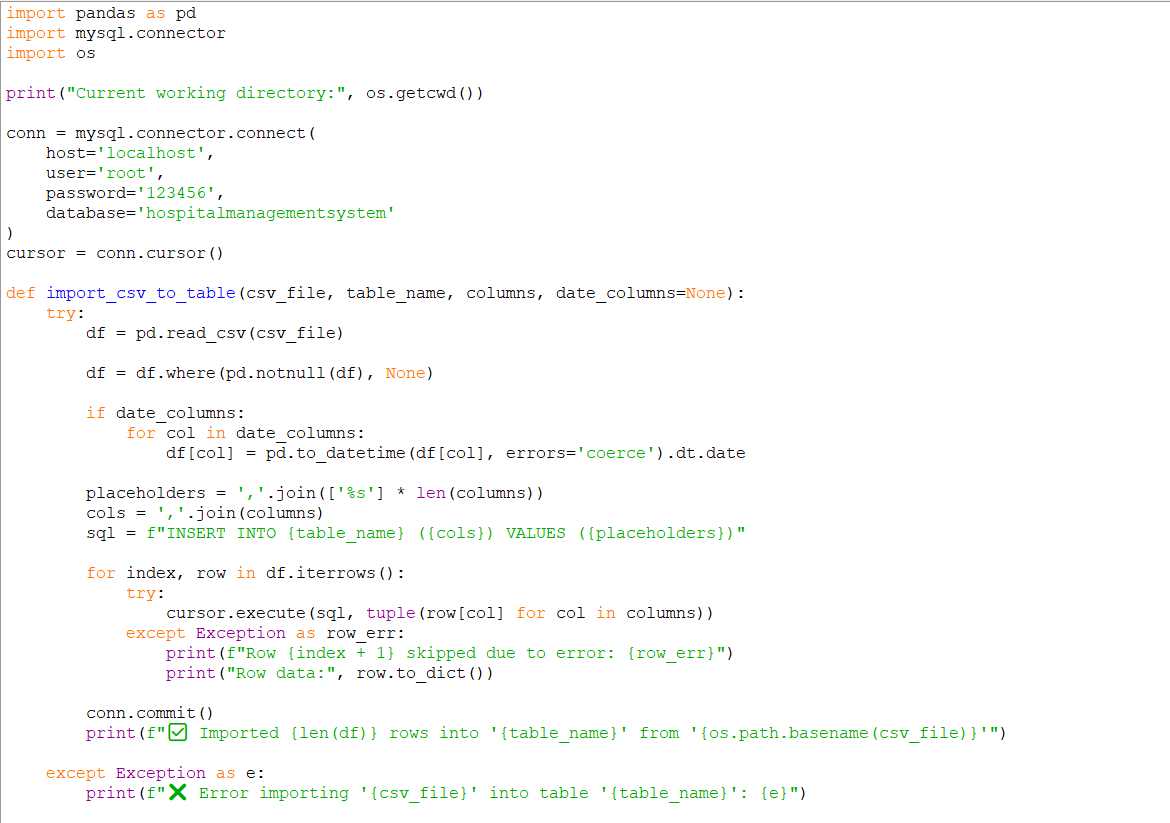
A flexible function that:

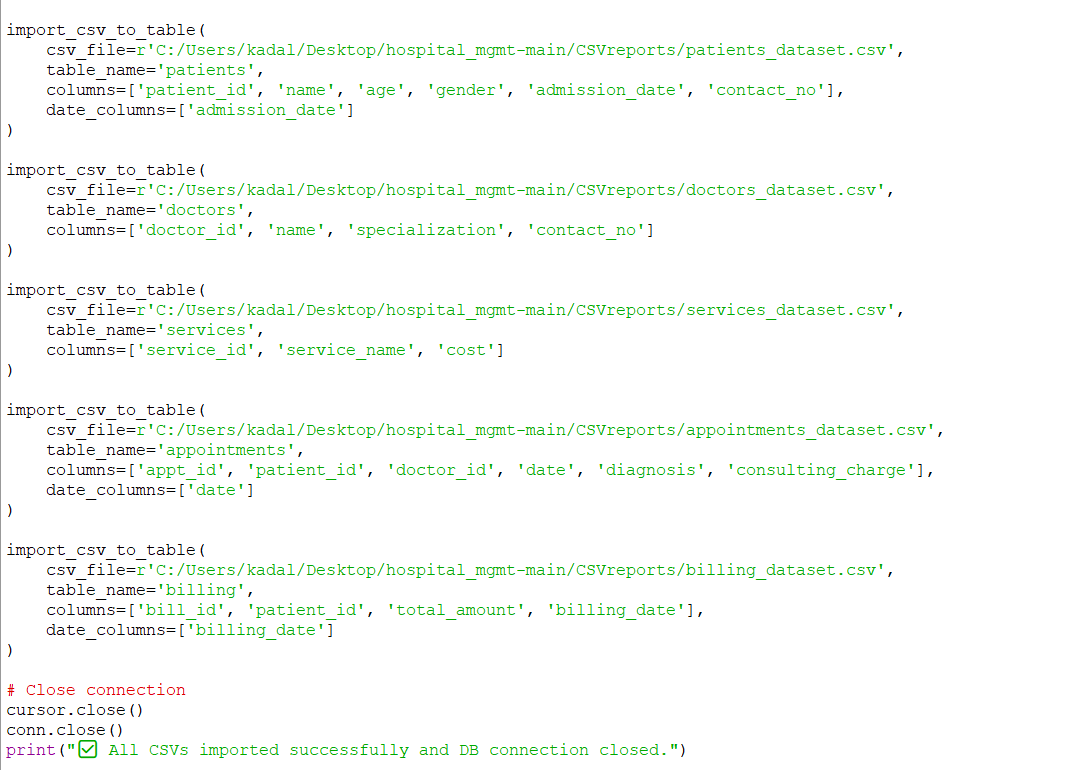
* Takes the path to a CSV file
* Specifies which database table to insert into
* Accepts a list of column names and optional date columns
* Parses the CSV into a DataFrame
* Converts NaN to None (SQL NULL)
* Formats date columns properly
* Executes row-wise insertions into the database
* Catches and prints any row-specific errors

**5. Call Function for Each CSV File**

* Each call imports a different dataset

6. **Close the DB Connection**





**Cli workflow images**

Main Menu of each record

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

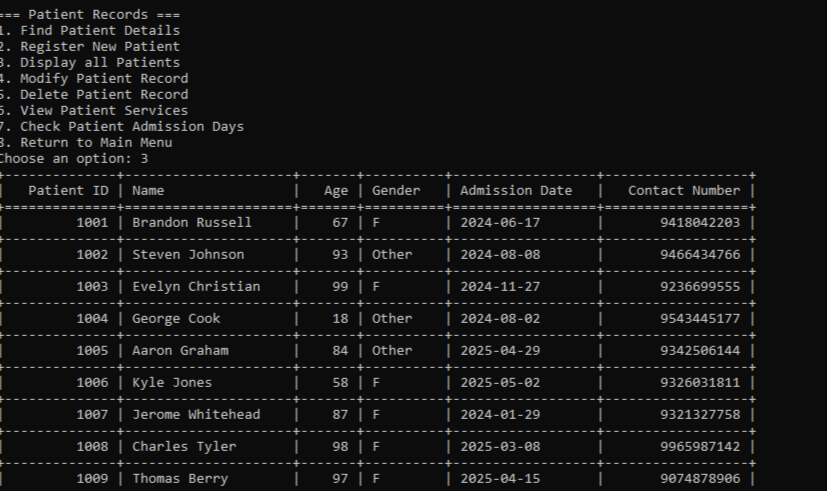
**Patient Records**

A computer screen shot of a black screen

AI-generated content may be incorrect.

A computer screen shot of a patient record

AI-generated content may be incorrect.



A computer screen shot of a patient record

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect. A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Doctor Records**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Service Records**

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Appointment Records**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

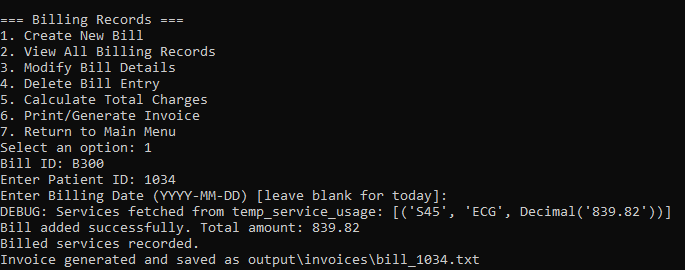
A screenshot of a computer

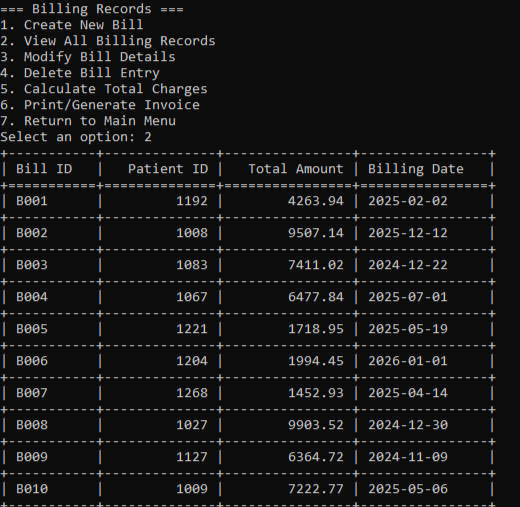
AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Billing Records**





A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Export Records**

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.