Task no. 1

Write a Python function called **reverse_words** that takes a single string argument, which represents a sentence. The function should return a new string where the order of the words in the original sentence is reversed.

Example

```
# Example
input_sentence = "I want to work in Solutia"
reversed_sentence = reverse_words(input_sentence)
print(reversed_sentence)  # Expected output: "Solutia in work to want I"
```

Requirements

- Your function must be named **reverse_words**.
- It must accept one argument, a string. No other data type will be accepted.
- It must return a new string with the words in reversed order.
- Consider how to handle multiple spaces between words (ideally, they should be treated as single separators).

Bonus (optional)

- Can you handle punctuation attached to words correctly? For example, if the input is "Hello, world!", the output should ideally be "!world, Hello". (This is a bit more challenging and not strictly required for a basic assessment).
- Create another Python function called **reverse_and_swap_case** that does the same word reversal as reverse_words, but *also* swaps the case of each letter in the string (i.e., converts lowercase to uppercase and uppercase to lowercase).

Task no. 2

You are given a list of dictionaries, where each dictionary represents information about a book. Each dictionary has the keys "**title**" (string) and "**price**" (float).

Write a Python function called **calculate_average_price** that takes this list of book dictionaries as input and returns the average price of all the books in the list. If the input list is empty, the function should return 0.

Example

```python
# Example
books = [
    {"title": "The Hitchhiker's Guide to the Galaxy", "price": 12.50},
    {"title": "Pride and Prejudice", "price": 9.99},
    {"title": "To Kill a Mockingbird", "price": 15.00}
]
average = calculate_average_price(books)
print(f"The average price is: {average}")  # The average price is: 12.496666666666667

empty_books = []
average_empty = calculate_average_price(empty_books)
print(f"The average price for an empty list is: {average_empty}") #mThe average price for an empty list is: 0
```

Requirements

1. Your function must be named **calculate_average_price**.
2. It must accept one argument, a list of dictionaries.
3. Each dictionary in the list is expected to have "title" (string) and "price" (float) keys.
4. The function should return a float representing the average price.
5. If the input list is empty, the function should return 0.

Task no. 3

Create a simple **CRUD** (Create, Read, Update, Delete) API using **FastAPI** to manage a collection of books. The application will store book data in a **JSON** file (**books.json**) instead of a database.

Data Format

The book data will be in the following JSON format, consistent with Assignment 2

```
[
  {
    "title": "The Hitchhiker's Guide to the Galaxy",
    "price": 12.5
  },
  {
    "title": "Pride and Prejudice",
    "price": 9.99
  },
  {
    "title": "To Kill a Mockingbird",
    "price": 15
  }
]
```

Requirements

Create a FastAPI application: Set up a basic FastAPI application.

Data Storage:

- Create a file named **books.json** in the same directory as your main Python file.
- This file will store the book data as a JSON array of book objects.
- If the file doesn't exist, create it and initialize it with an empty JSON array ([]).

CRUD Endpoints: Implement the following CRUD endpoints:

- **Create (POST /books/):**
  - Accept a book object in JSON format in the request body.
  - Add the new book to the books.json file.
  - Return the newly created book object with a 201 Created status code.
- **Read All (GET /books/):**

    - Return all books from the books.json file as a JSON array.
    - Return a 200 OK status code.
- **Read One (GET /books/{title}):**
    - Accept a book title as a path parameter.
    - Search for the book with the matching title in the books.json file.
    - If the book is found, return the book object with a 200 OK status code.
    - If the book is not found, return a 404 Not Found error with an appropriate message (e.g., "Book not found").
- **Update (PUT /books/{title}):**
    - Accept a book title as a path parameter.
    - Accept a book object in JSON format in the request body.
    - Update the book with the matching title in the books.json file with the new data from the request body.
    - If the book is found and updated, return the updated book object with a 200 OK status code.
    - If the book is not found, return a 404 Not Found error with an appropriate message.
- **Delete (DELETE /books/{title}):**
    - Accept a book title as a path parameter.
    - Delete the book with the matching title from the books.json file.
    - If the book is found and deleted, return a 204 No Content status code.
    - If the book is not found, return a 404 Not Found error with an appropriate message.

Technical Requirements:

1. Use **FastAPI** for the API framework.
2. Use Python's **json** module to read from and write to the **books.json** file.
3. Handle errors appropriately (e.g., file not found, book not found).
4. Ensure your code is well-structured, readable, and follows Python best practices.
5. Use descriptive variable and function names.
6. Include comments to explain your code.

Task no. 4

Design an Entity-Relationship Diagram (ERD) for a **Configuration Management Database** (CMDB). The diagram should represent the essential entities and relationships needed to manage an IT infrastructure, including **users, computers, installed software, and physical locations.** You can use a drawing tool, a database design software, or draw it by hand and scan it.

Scenario

An organization wants to implement a CMDB to track and manage its IT assets. The CMDB needs to store information about the following:

- **Users:** Employees who use the computers. Each user has a unique ID, name, email, and department.
- **Computers:** The hardware devices (laptops, desktops) used by the users. Each computer has a unique ID, hostname, serial number, operating system, and model.
- **Installed Software:** The software applications installed on each computer. Each software installation includes the software name, version, and installation date.
- **Physical Locations:** The physical locations where the computers are located. Each location has a unique ID, name, building, and room number.

Requirements

1. **Entities:** Identify the main entities in the scenario (e.g., User, Computer, Software, Location).
2. **Attributes:** Determine the relevant attributes for each entity. Specify appropriate data types for each attribute.
3. **Relationships:** Define the relationships between the entities (e.g., a user uses a computer, a computer is located in a location).
4. **Cardinality:** Specify the cardinality of each relationship (e.g., one user can use multiple computers, a computer is used by one user).
5. **Primary Keys:** Choose a primary key for each entity.
6. **Foreign Keys:** Identify any foreign keys needed to establish the relationships between tables.