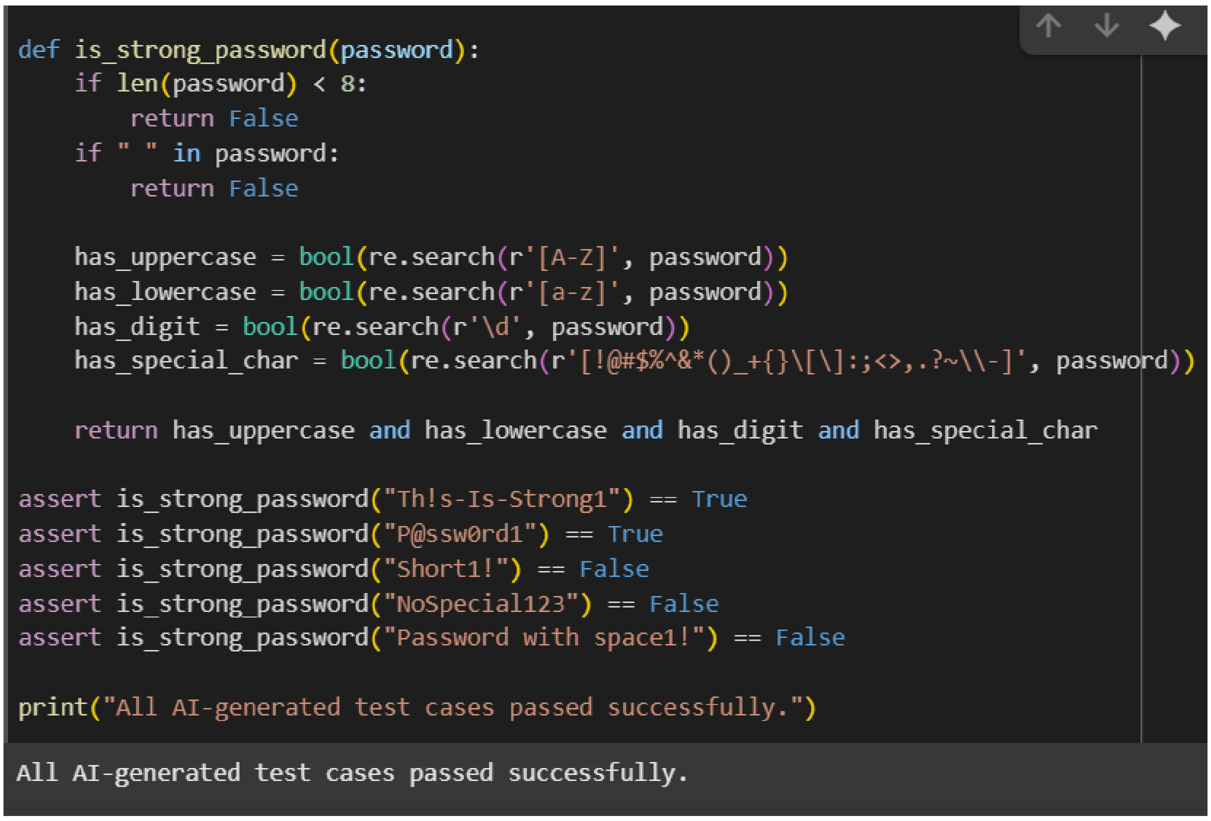
ASSIGNMENT 8

RAMIDI . SAI CHARAN

2403A52124

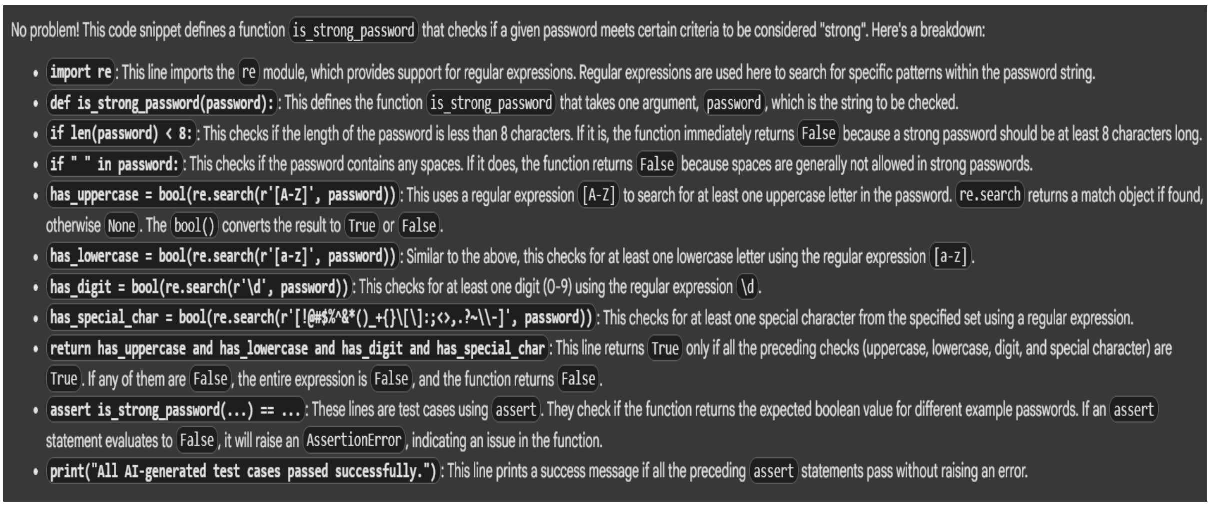
TASK 1 :



```python
def is_strong_password(password):
    if len(password) < 8:
        return False
    if " " in password:
        return False

    has_uppercase = bool(re.search(r'[A-Z]', password))
    has_lowercase = bool(re.search(r'[a-z]', password))
    has_digit = bool(re.search(r'\d', password))
    has_special_char = bool(re.search(r'[!@#$%^&*()_+{}\[\]:;<>,.?~\\-]', password))

    return has_uppercase and has_lowercase and has_digit and has_special_char

assert is_strong_password("Th!s-Is-Strong1") == True
assert is_strong_password("P@ssw0rd1") == True
assert is_strong_password("Short1!") == False
assert is_strong_password("NoSpecial123") == False
assert is_strong_password("Password with space1!") == False

print("All AI-generated test cases passed successfully.")
```

```
All AI-generated test cases passed successfully.
```

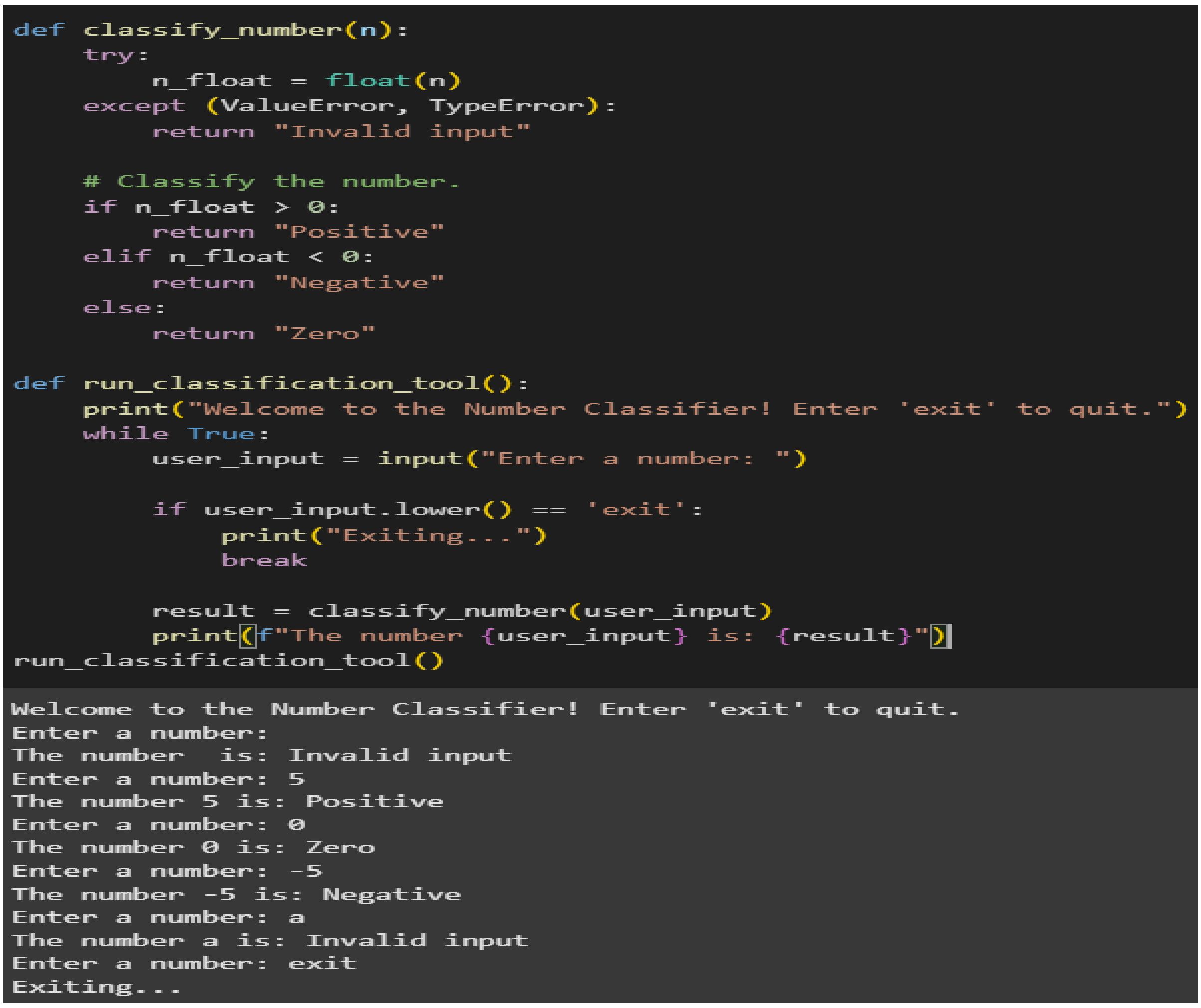
EXPLANATION :



No problem! This code snippet defines a function `is_strong_password` that checks if a given password meets certain criteria to be considered "strong". Here's a breakdown:

- **`import re`**: This line imports the `re` module, which provides support for regular expressions. Regular expressions are used here to search for specific patterns within the password string.
- **`def is_strong_password(password):`**: This defines the function `is_strong_password` that takes one argument, `password`, which is the string to be checked.
- **`if len(password) < 8:`**: This checks if the length of the password is less than 8 characters. If it is, the function immediately returns `False` because a strong password should be at least 8 characters long.
- **`if " " in password:`**: This checks if the password contains any spaces. If it does, the function returns `False` because spaces are generally not allowed in strong passwords.
- **`has_uppercase = bool(re.search(r'[A-Z]', password))`**: This uses a regular expression `[A-Z]` to search for at least one uppercase letter in the password. `re.search` returns a match object if found, otherwise `None`. The `bool()` converts the result to `True` or `False`.
- **`has_lowercase = bool(re.search(r'[a-z]', password))`**: Similar to the above, this checks for at least one lowercase letter using the regular expression `[a-z]`.
- **`has_digit = bool(re.search(r'\d', password))`**: This checks for at least one digit (0-9) using the regular expression `\d`.
- **`has_special_char = bool(re.search(r'[!@#$%^&*()_+{}\[\]:;<>,.?~\\-]', password))`**: This checks for at least one special character from the specified set using a regular expression.
- **`return has_uppercase and has_lowercase and has_digit and has_special_char`**: This line returns `True` only if all the preceding checks (uppercase, lowercase, digit, and special character) are `True`. If any of them are `False`, the entire expression is `False`, and the function returns `False`.
- **`assert is_strong_password(...) == ...`**: These lines are test cases using `assert`. They check if the function returns the expected boolean value for different example passwords. If an `assert` statement evaluates to `False`, it will raise an `AssertionError`, indicating an issue in the function.
- **`print("All AI-generated test cases passed successfully.")`**: This line prints a success message if all the preceding `assert` statements pass without raising an error.

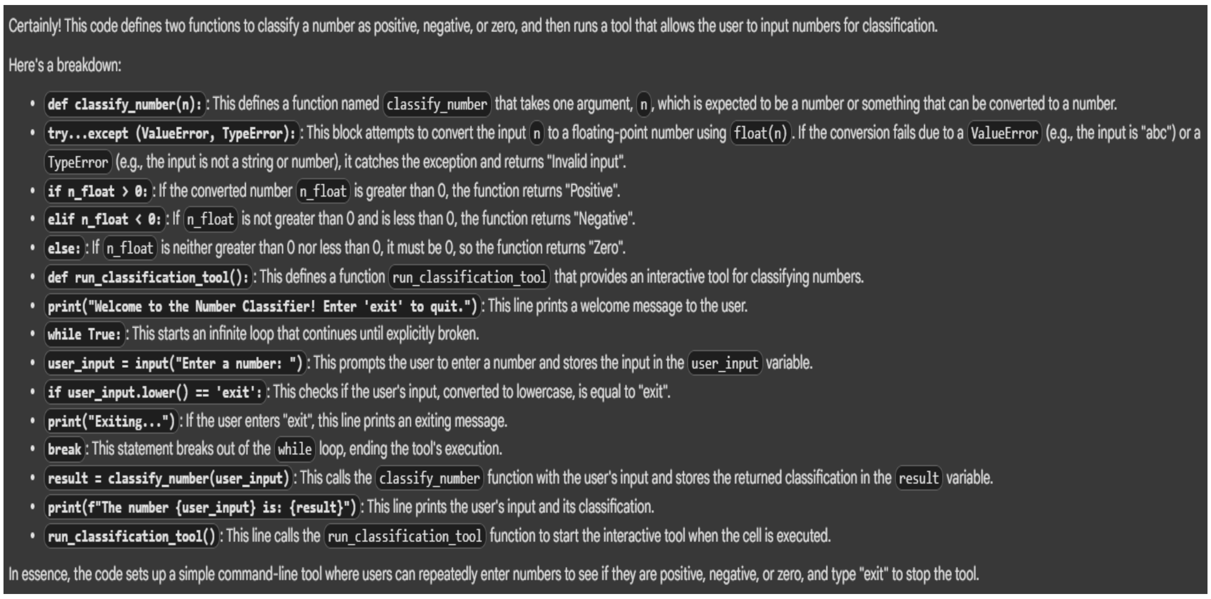
TASK 2 :



```python
def classify_number(n):
    try:
        n_float = float(n)
    except (ValueError, TypeError):
        return "Invalid input"

    # Classify the number.
    if n_float > 0:
        return "Positive"
    elif n_float < 0:
        return "Negative"
    else:
        return "Zero"

def run_classification_tool():
    print("Welcome to the Number Classifier! Enter 'exit' to quit.")
    while True:
        user_input = input("Enter a number: ")

        if user_input.lower() == 'exit':
            print("Exiting...")
            break

        result = classify_number(user_input)
        print(f"The number {user_input} is: {result}")
run_classification_tool()
```

```
Welcome to the Number Classifier! Enter 'exit' to quit.
Enter a number: 
The number  is: Invalid input
Enter a number: 5
The number 5 is: Positive
Enter a number: 0
The number 0 is: Zero
Enter a number: -5
The number -5 is: Negative
Enter a number: a
The number a is: Invalid input
Enter a number: exit
Exiting...
```

EXPLANATION :



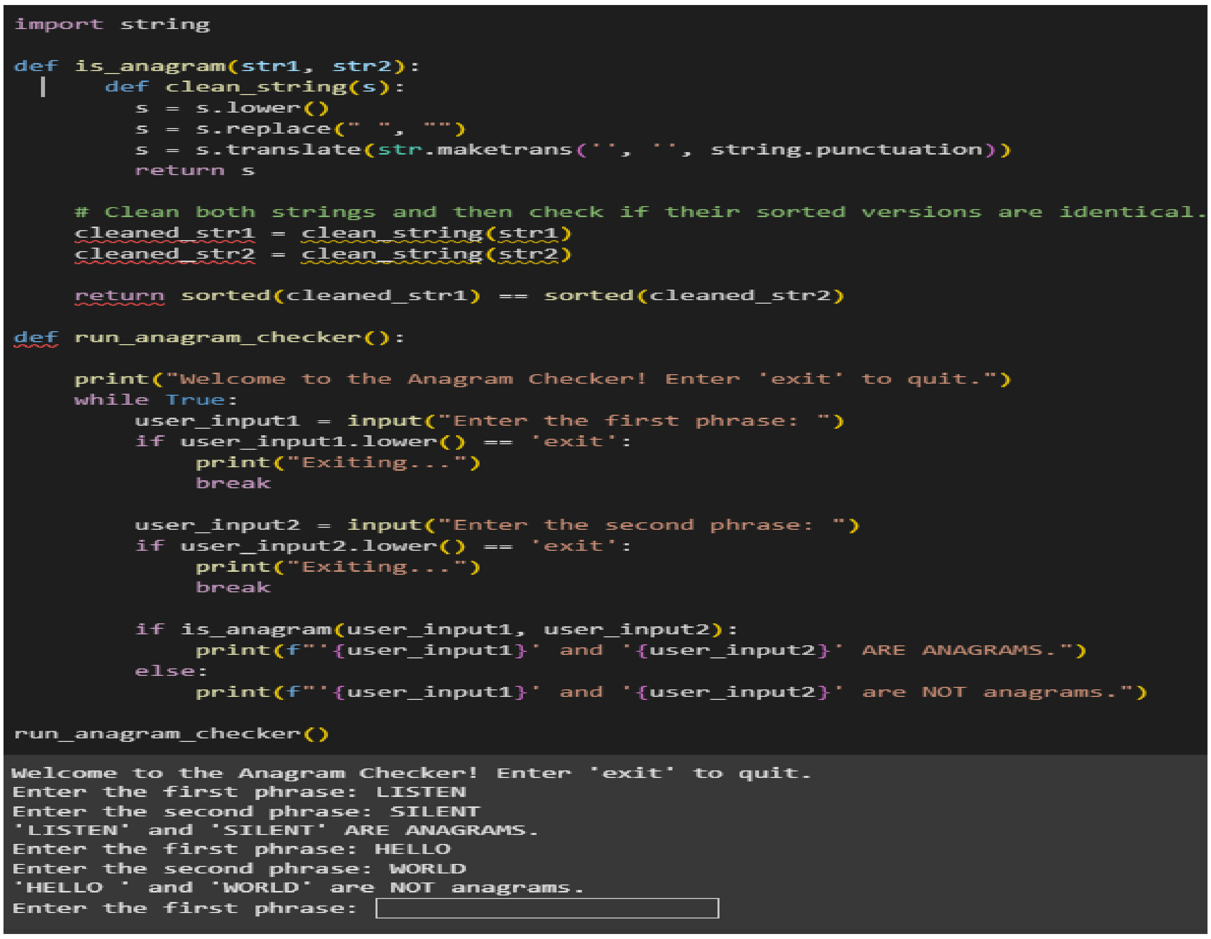
Certainly! This code defines two functions to classify a number as positive, negative, or zero, and then runs a tool that allows the user to input numbers for classification.

Here's a breakdown:

- **`def classify_number(n):`**: This defines a function named `classify_number` that takes one argument, `n`, which is expected to be a number or something that can be converted to a number.
- **`try...except (ValueError, TypeError):`**: This block attempts to convert the input `n` to a floating-point number using `float(n)`. If the conversion fails due to a `ValueError` (e.g., the input is "abc") or a `TypeError` (e.g., the input is not a string or number), it catches the exception and returns "Invalid input".
- **`if n_float > 0:`**: If the converted number `n_float` is greater than 0, the function returns "Positive".
- **`elif n_float < 0:`**: If `n_float` is not greater than 0 and is less than 0, the function returns "Negative".
- **`else:`**: If `n_float` is neither greater than 0 nor less than 0, it must be 0, so the function returns "Zero".
- **`def run_classification_tool():`**: This defines a function `run_classification_tool` that provides an interactive tool for classifying numbers.
- **`print("Welcome to the Number Classifier! Enter 'exit' to quit.")`**: This line prints a welcome message to the user.
- **`while True:`**: This starts an infinite loop that continues until explicitly broken.
- **`user_input = input("Enter a number: ")`**: This prompts the user to enter a number and stores the input in the `user_input` variable.
- **`if user_input.lower() == 'exit':`**: This checks if the user's input, converted to lowercase, is equal to "exit".
- **`print("Exiting...")`**: If the user enters "exit", this line prints an exiting message.
- **`break`**: This statement breaks out of the `while` loop, ending the tool's execution.
- **`result = classify_number(user_input)`**: This calls the `classify_number` function with the user's input and stores the returned classification in the `result` variable.
- **`print(f"The number {user_input} is: {result}")`**: This line prints the user's input and its classification.
- **`run_classification_tool()`**: This line calls the `run_classification_tool` function to start the interactive tool when the cell is executed.

In essence, the code sets up a simple command-line tool where users can repeatedly enter numbers to see if they are positive, negative, or zero, and type "exit" to stop the tool.

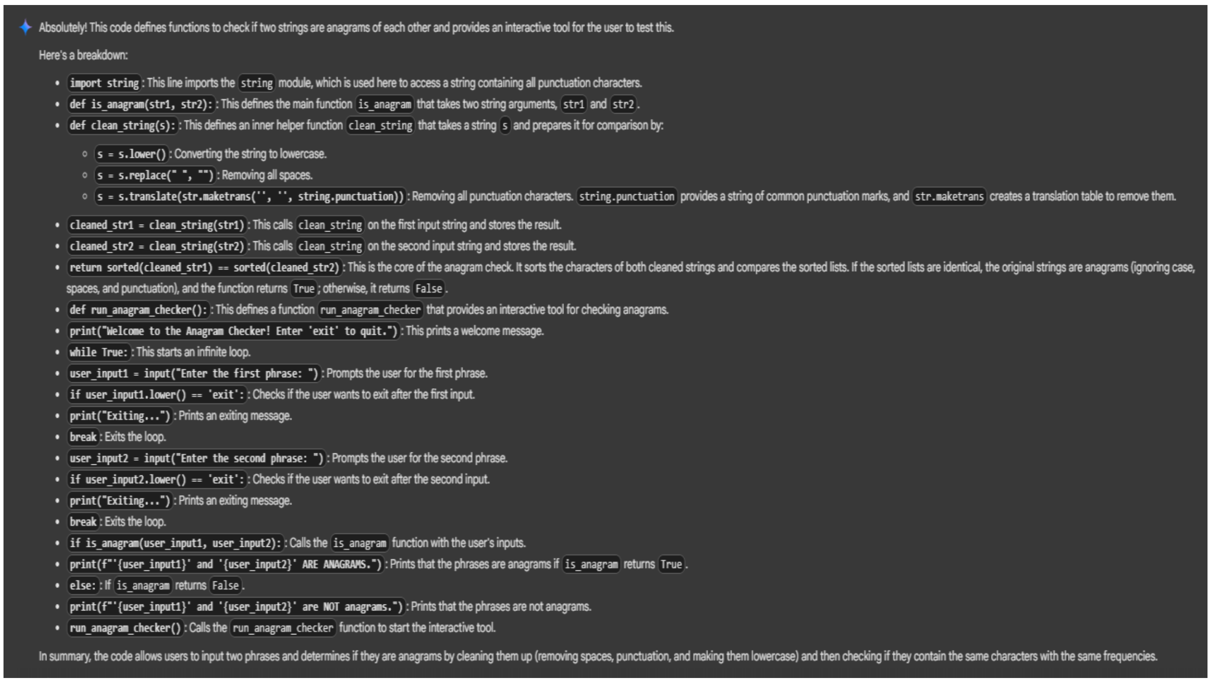
TASK 3 :



```python
import string

def is_anagram(str1, str2):
    def clean_string(s):
        s = s.lower()
        s = s.replace(" ", "")
        s = s.translate(str.maketrans('', '', string.punctuation))
        return s

    # Clean both strings and then check if their sorted versions are identical.
    cleaned_str1 = clean_string(str1)
    cleaned_str2 = clean_string(str2)

    return sorted(cleaned_str1) == sorted(cleaned_str2)

def run_anagram_checker():

    print("Welcome to the Anagram Checker! Enter 'exit' to quit.")
    while True:
        user_input1 = input("Enter the first phrase: ")
        if user_input1.lower() == 'exit':
            print("Exiting...")
            break

        user_input2 = input("Enter the second phrase: ")
        if user_input2.lower() == 'exit':
            print("Exiting...")
            break

        if is_anagram(user_input1, user_input2):
            print(f"'{user_input1}' and '{user_input2}' ARE ANAGRAMS.")
        else:
            print(f"'{user_input1}' and '{user_input2}' are NOT anagrams.")

run_anagram_checker()
```

```
Welcome to the Anagram Checker! Enter 'exit' to quit.
Enter the first phrase: LISTEN
Enter the second phrase: SILENT
'LISTEN' and 'SILENT' ARE ANAGRAMS.
Enter the first phrase: HELLO
Enter the second phrase: WORLD
'HELLO ' and 'WORLD' are NOT anagrams.
Enter the first phrase:
```

EXPLANATION :



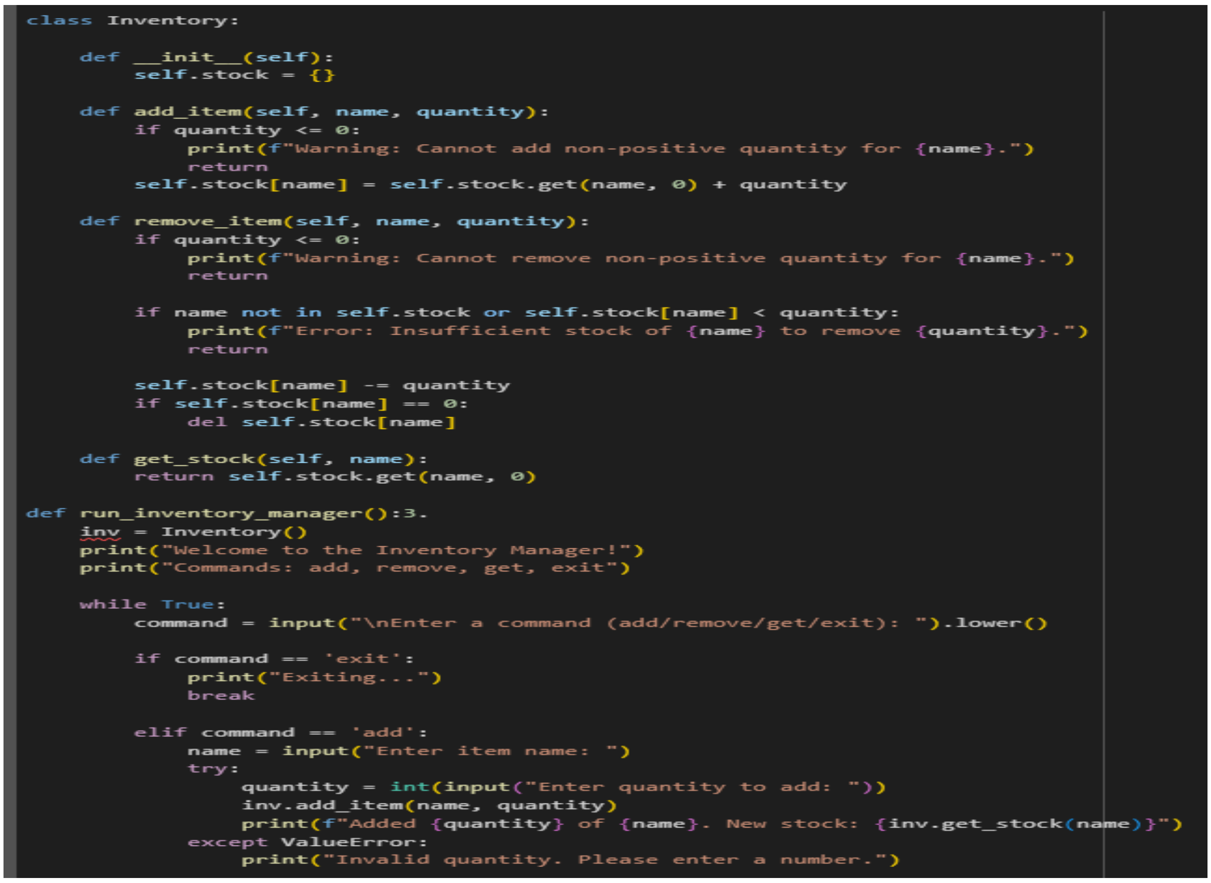
Absolutely! This code defines functions to check if two strings are anagrams of each other and provides an interactive tool for the user to test this.

Here's a breakdown:

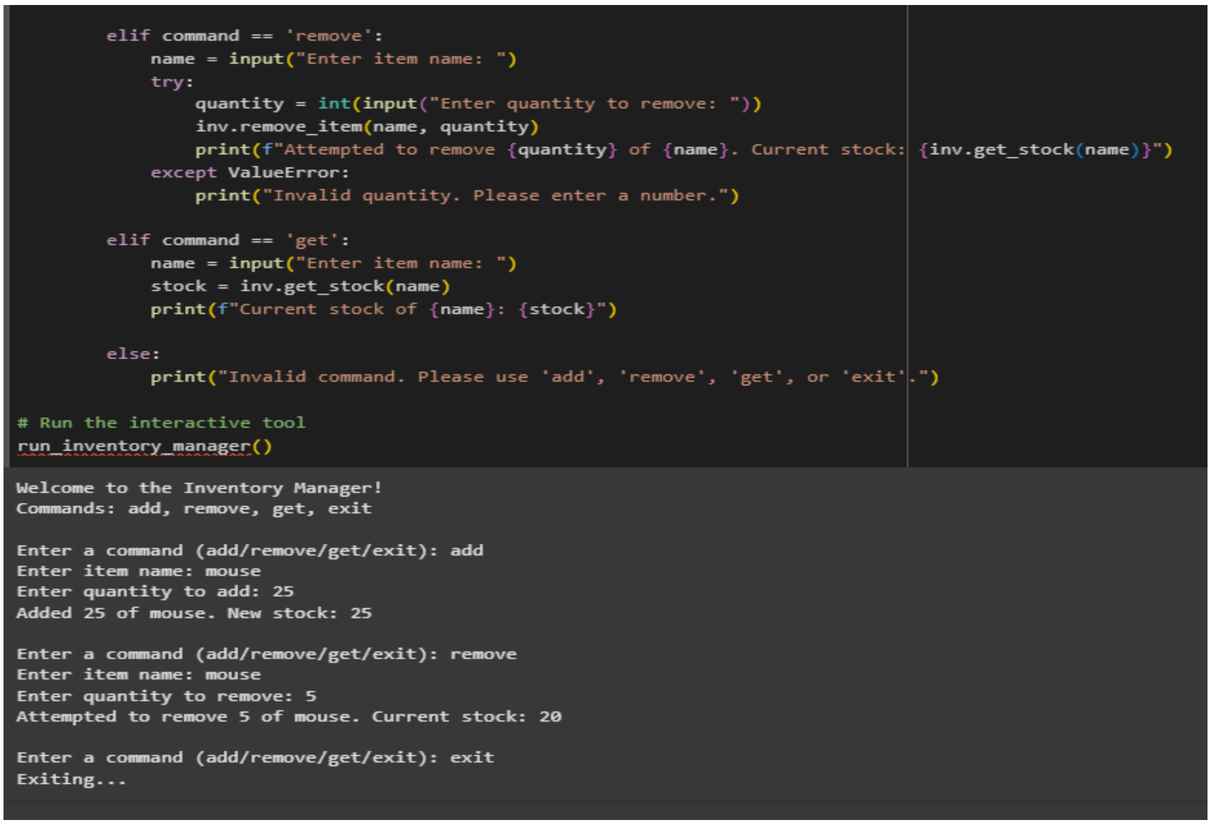
- `import string` : This line imports the `string` module, which is used here to access a string containing all punctuation characters.
- `def is_anagram(str1, str2):` : This defines the main function `is_anagram` that takes two string arguments, `str1` and `str2`.
- `def clean_string(s):` : This defines an inner helper function `clean_string` that takes a string `s` and prepares it for comparison by:
    - `s = s.lower()` : Converting the string to lowercase.
    - `s = s.replace(" ", "")` : Removing all spaces.
    - `s = s.translate(str.maketrans('', '', string.punctuation))` : Removing all punctuation characters. `string.punctuation` provides a string of common punctuation marks, and `str.maketrans` creates a translation table to remove them.
- `cleaned_str1 = clean_string(str1)` : This calls `clean_string` on the first input string and stores the result.
- `cleaned_str2 = clean_string(str2)` : This calls `clean_string` on the second input string and stores the result.
- `return sorted(cleaned_str1) == sorted(cleaned_str2)` : This is the core of the anagram check. It sorts the characters of both cleaned strings and compares the sorted lists. If the sorted lists are identical, the original strings are anagrams (ignoring case, spaces, and punctuation), and the function returns `True` ; otherwise, it returns `False`.
- `def run_anagram_checker():` : This defines a function `run_anagram_checker` that provides an interactive tool for checking anagrams.
- `print("Welcome to the Anagram Checker! Enter 'exit' to quit.")` : This prints a welcome message.
- `while True:` : This starts an infinite loop.
- `user_input1 = input("Enter the first phrase: ")` : Prompts the user for the first phrase.
- `if user_input1.lower() == 'exit':` : Checks if the user wants to exit after the first input.
- `print("Exiting...")` : Prints an exiting message.
- `break` : Exits the loop.
- `user_input2 = input("Enter the second phrase: ")` : Prompts the user for the second phrase.
- `if user_input2.lower() == 'exit':` : Checks if the user wants to exit after the second input.
- `print("Exiting...")` : Prints an exiting message.
- `break` : Exits the loop.
- `if is_anagram(user_input1, user_input2):` : Calls the `is_anagram` function with the user's inputs.
- `print(f"'{user_input1}' and '{user_input2}' ARE ANAGRAMS.")` : Prints that the phrases are anagrams if `is_anagram` returns `True`.
- `else:` : If `is_anagram` returns `False`.
- `print(f"'{user_input1}' and '{user_input2}' are NOT anagrams.")` : Prints that the phrases are not anagrams.
- `run_anagram_checker()` : Calls the `run_anagram_checker` function to start the interactive tool.

In summary, the code allows users to input two phrases and determines if they are anagrams by cleaning them up (removing spaces, punctuation, and making them lowercase) and then checking if they contain the same characters with the same frequencies.

TASK 4 :



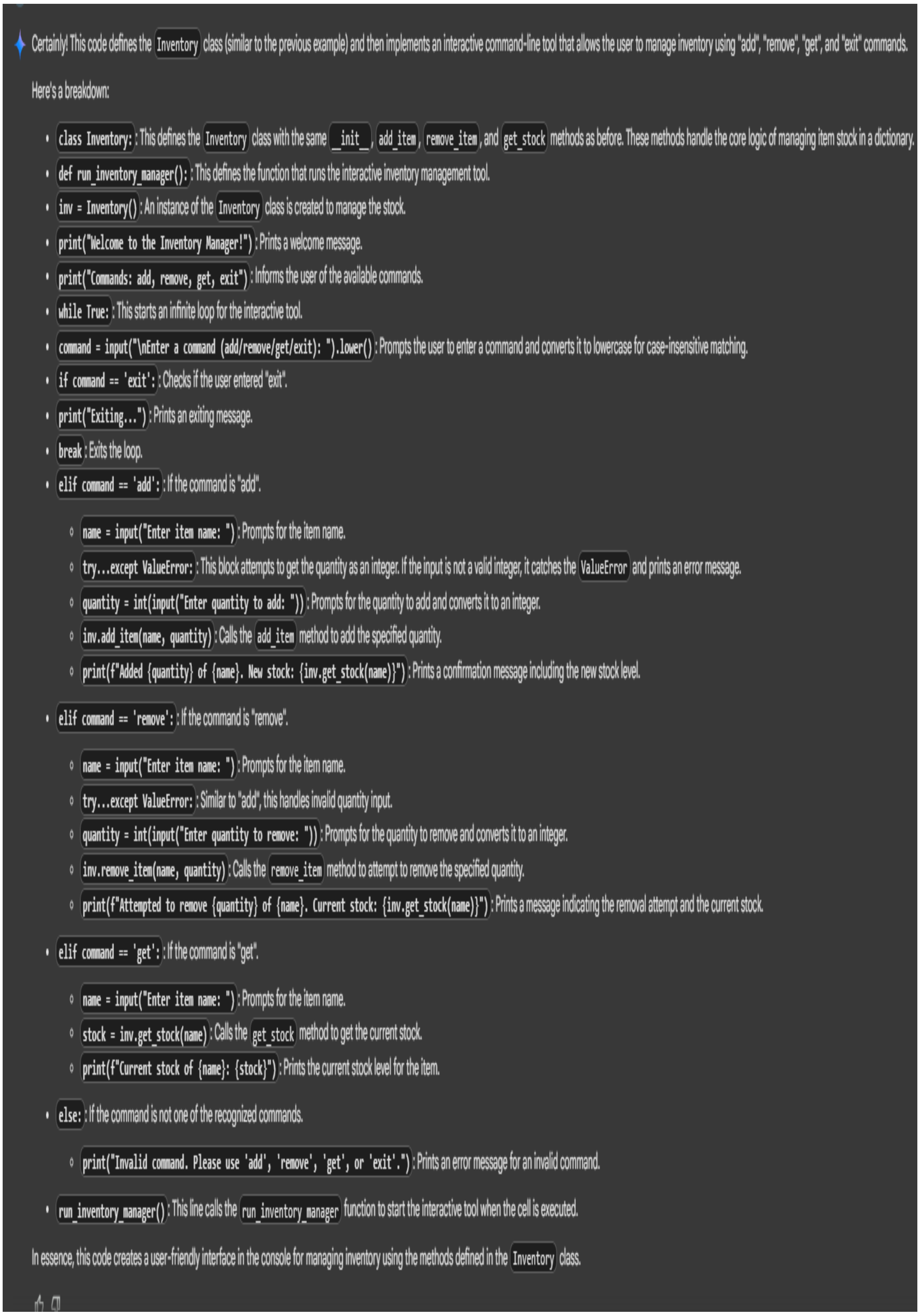
```python
class Inventory:

    def __init__(self):
        self.stock = {}

    def add_item(self, name, quantity):
        if quantity <= 0:
            print(f"Warning: Cannot add non-positive quantity for {name}.")
            return
        self.stock[name] = self.stock.get(name, 0) + quantity

    def remove_item(self, name, quantity):
        if quantity <= 0:
            print(f"Warning: Cannot remove non-positive quantity for {name}.")
            return

        if name not in self.stock or self.stock[name] < quantity:
            print(f"Error: Insufficient stock of {name} to remove {quantity}.")
            return

        self.stock[name] -= quantity
        if self.stock[name] == 0:
            del self.stock[name]

    def get_stock(self, name):
        return self.stock.get(name, 0)

def run_inventory_manager():3.
    inv = Inventory()
    print("Welcome to the Inventory Manager!")
    print("Commands: add, remove, get, exit")

    while True:
        command = input("\nEnter a command (add/remove/get/exit): ").lower()

        if command == 'exit':
            print("Exiting...")
            break

        elif command == 'add':
            name = input("Enter item name: ")
            try:
                quantity = int(input("Enter quantity to add: "))
                inv.add_item(name, quantity)
                print(f"Added {quantity} of {name}. New stock: {inv.get_stock(name)}")
            except ValueError:
                print("Invalid quantity. Please enter a number.")
```



```python
        elif command == 'remove':
            name = input("Enter item name: ")
            try:
                quantity = int(input("Enter quantity to remove: "))
                inv.remove_item(name, quantity)
                print(f"Attempted to remove {quantity} of {name}. Current stock: {inv.get_stock(name)}")
            except ValueError:
                print("Invalid quantity. Please enter a number.")

        elif command == 'get':
            name = input("Enter item name: ")
            stock = inv.get_stock(name)
            print(f"Current stock of {name}: {stock}")

        else:
            print("Invalid command. Please use 'add', 'remove', 'get', or 'exit'.")

# Run the interactive tool
run_inventory_manager()
```

```
Welcome to the Inventory Manager!
Commands: add, remove, get, exit

Enter a command (add/remove/get/exit): add
Enter item name: mouse
Enter quantity to add: 25
Added 25 of mouse. New stock: 25

Enter a command (add/remove/get/exit): remove
Enter item name: mouse
Enter quantity to remove: 5
Attempted to remove 5 of mouse. Current stock: 20

Enter a command (add/remove/get/exit): exit
Exiting...
```

EXPLANATION :



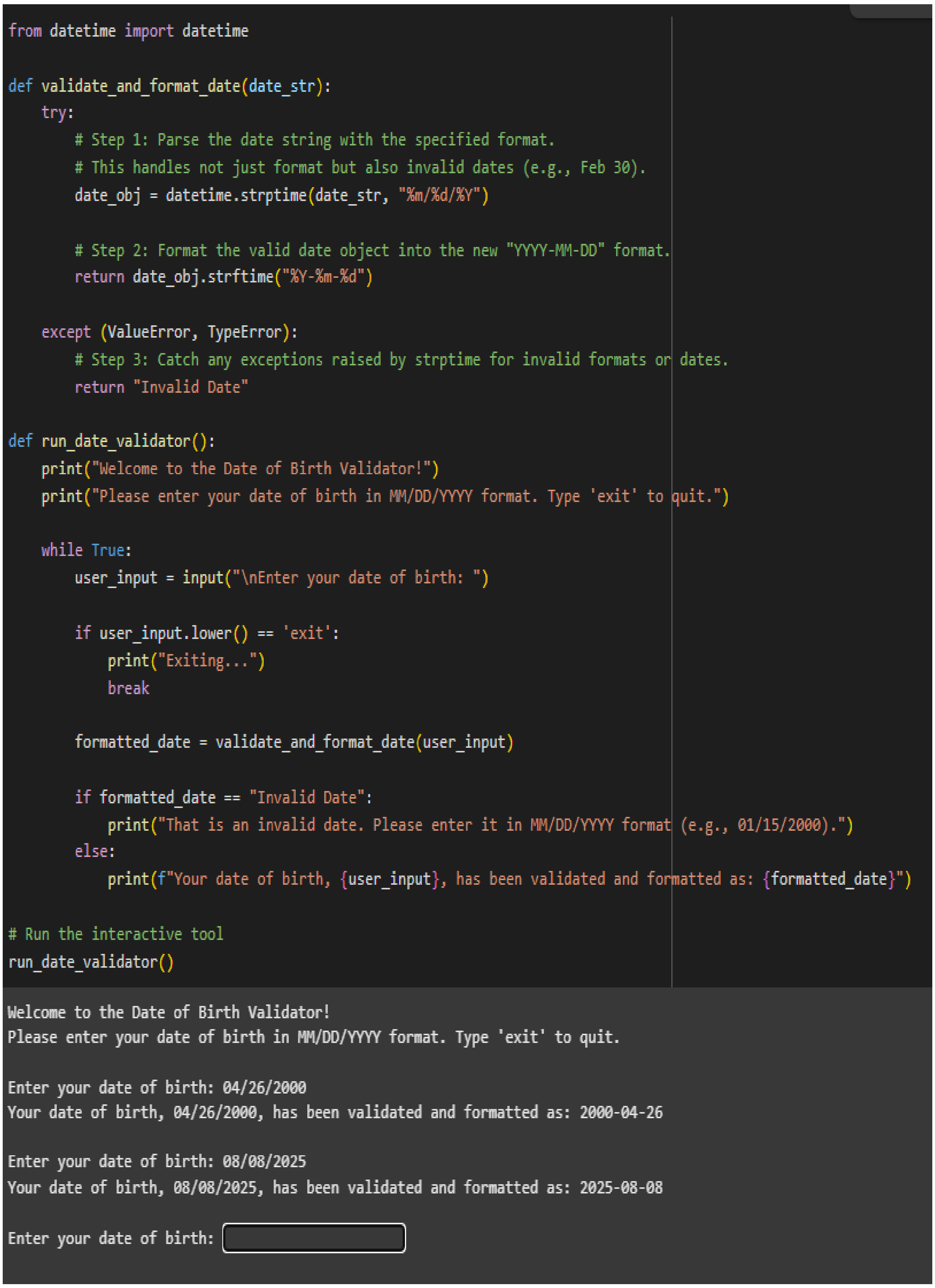
Certainly! This code defines the `Inventory` class (similar to the previous example) and then implements an interactive command-line tool that allows the user to manage inventory using "add", "remove", "get", and "exit" commands.

Here's a breakdown:

- `class Inventory:` : This defines the `Inventory` class with the same `__init__`, `add_item`, `remove_item`, and `get_stock` methods as before. These methods handle the core logic of managing item stock in a dictionary.
- `def run_inventory_manager():` : This defines the function that runs the interactive inventory management tool.
- `inv = Inventory()` : An instance of the `Inventory` class is created to manage the stock.
- `print("Welcome to the Inventory Manager!")` : Prints a welcome message.
- `print("Commands: add, remove, get, exit")` : Informs the user of the available commands.
- `while True:` : This starts an infinite loop for the interactive tool.
- `command = input("\nEnter a command (add/remove/get/exit): ").lower()` : Prompts the user to enter a command and converts it to lowercase for case-insensitive matching.
- `if command == 'exit':` : Checks if the user entered "exit".
- `print("Exiting...")` : Prints an exiting message.
- `break` : Exits the loop.
- `elif command == 'add':` : If the command is "add".
  - `name = input("Enter item name: ")` : Prompts for the item name.
  - `try...except ValueError:` : This block attempts to get the quantity as an integer. If the input is not a valid integer, it catches the `ValueError` and prints an error message.
  - `quantity = int(input("Enter quantity to add: "))` : Prompts for the quantity to add and converts it to an integer.
  - `inv.add_item(name, quantity)` : Calls the `add_item` method to add the specified quantity.
  - `print(f"Added {quantity} of {name}. New stock: {inv.get_stock(name)}")` : Prints a confirmation message including the new stock level.
- `elif command == 'remove':` : If the command is "remove".
  - `name = input("Enter item name: ")` : Prompts for the item name.
  - `try...except ValueError:` : Similar to "add", this handles invalid quantity input.
  - `quantity = int(input("Enter quantity to remove: "))` : Prompts for the quantity to remove and converts it to an integer.
  - `inv.remove_item(name, quantity)` : Calls the `remove_item` method to attempt to remove the specified quantity.
  - `print(f"Attempted to remove {quantity} of {name}. Current stock: {inv.get_stock(name)}")` : Prints a message indicating the removal attempt and the current stock.
- `elif command == 'get':` : If the command is "get".
  - `name = input("Enter item name: ")` : Prompts for the item name.
  - `stock = inv.get_stock(name)` : Calls the `get_stock` method to get the current stock.
  - `print(f"Current stock of {name}: {stock}")` : Prints the current stock level for the item.
- `else:` : If the command is not one of the recognized commands.
  - `print("Invalid command. Please use 'add', 'remove', 'get', or 'exit'.")` : Prints an error message for an invalid command.
- `run_inventory_manager()` : This line calls the `run_inventory_manager` function to start the interactive tool when the cell is executed.

In essence, this code creates a user-friendly interface in the console for managing inventory using the methods defined in the `Inventory` class.

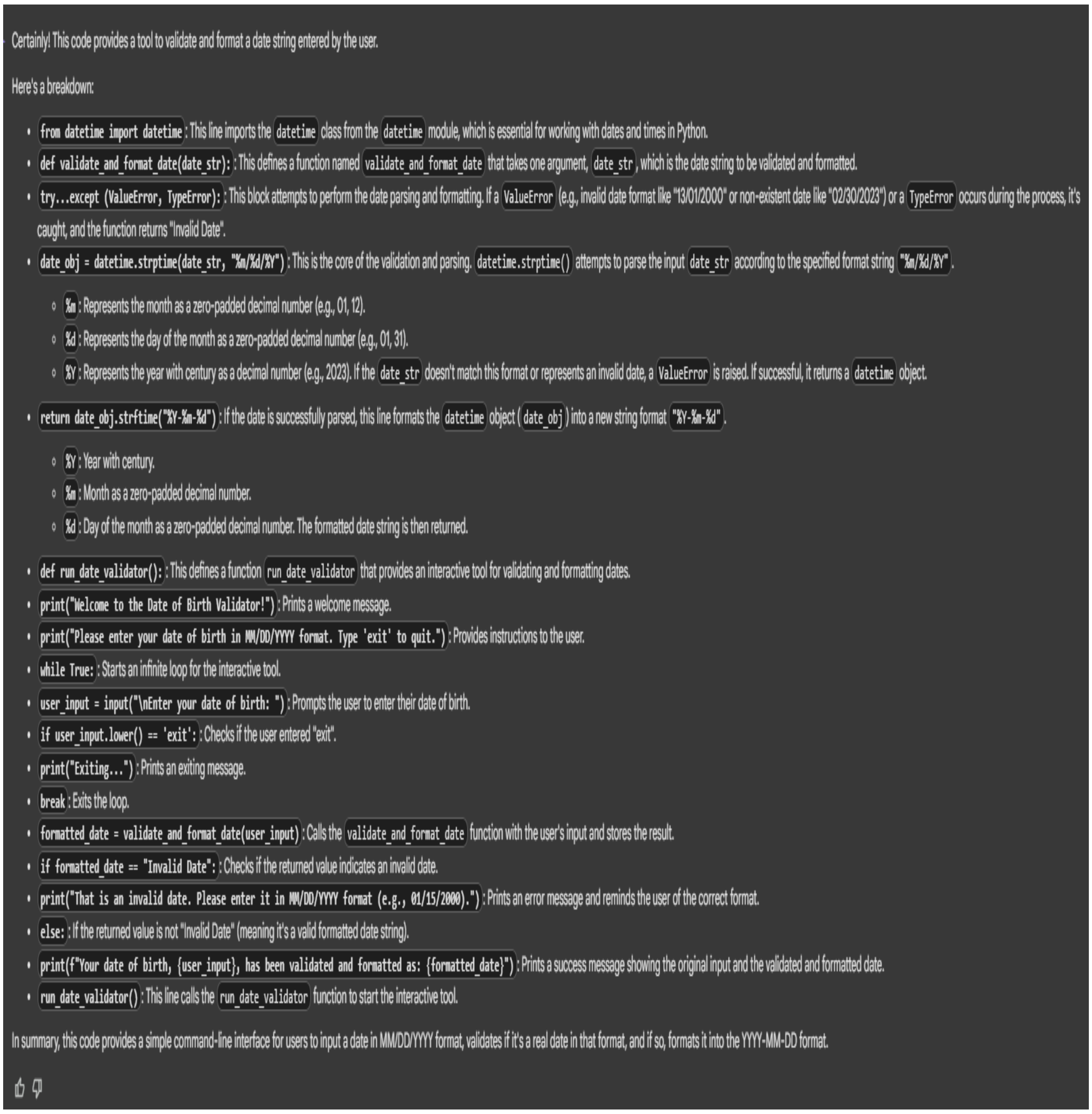
TASK 5 :



```python
from datetime import datetime

def validate_and_format_date(date_str):
    try:
        # Step 1: Parse the date string with the specified format.
        # This handles not just format but also invalid dates (e.g., Feb 30).
        date_obj = datetime.strptime(date_str, "%m/%d/%Y")

        # Step 2: Format the valid date object into the new "YYYY-MM-DD" format.
        return date_obj.strftime("%Y-%m-%d")

    except (ValueError, TypeError):
        # Step 3: Catch any exceptions raised by strptime for invalid formats or dates.
        return "Invalid Date"

def run_date_validator():
    print("Welcome to the Date of Birth Validator!")
    print("Please enter your date of birth in MM/DD/YYYY format. Type 'exit' to quit.")

    while True:
        user_input = input("\nEnter your date of birth: ")

        if user_input.lower() == 'exit':
            print("Exiting...")
            break

        formatted_date = validate_and_format_date(user_input)

        if formatted_date == "Invalid Date":
            print("That is an invalid date. Please enter it in MM/DD/YYYY format (e.g., 01/15/2000).")
        else:
            print(f"Your date of birth, {user_input}, has been validated and formatted as: {formatted_date}")

# Run the interactive tool
run_date_validator()
```

```
Welcome to the Date of Birth Validator!
Please enter your date of birth in MM/DD/YYYY format. Type 'exit' to quit.

Enter your date of birth: 04/26/2000
Your date of birth, 04/26/2000, has been validated and formatted as: 2000-04-26

Enter your date of birth: 08/08/2025
Your date of birth, 08/08/2025, has been validated and formatted as: 2025-08-08

Enter your date of birth:
```

EXPLANATION :



Certainly! This code provides a tool to validate and format a date string entered by the user.

Here's a breakdown:

- `from datetime import datetime` : This line imports the `datetime` class from the `datetime` module, which is essential for working with dates and times in Python.
- `def validate_and_format_date(date_str):` : This defines a function named `validate_and_format_date` that takes one argument, `date_str`, which is the date string to be validated and formatted.
- `try...except (ValueError, TypeError):` : This block attempts to perform the date parsing and formatting. If a `ValueError` (e.g., invalid date format like "13/01/2000" or non-existent date like "02/30/2023") or a `TypeError` occurs during the process, it's caught, and the function returns "Invalid Date".
- `date_obj = datetime.strptime(date_str, "%m/%d/%Y")` : This is the core of the validation and parsing. `datetime.strptime()` attempts to parse the input `date_str` according to the specified format string `"%m/%d/%Y"`.
    - `%m` : Represents the month as a zero-padded decimal number (e.g., 01, 12).
    - `%d` : Represents the day of the month as a zero-padded decimal number (e.g., 01, 31).
    - `%Y` : Represents the year with century as a decimal number (e.g., 2023). If the `date_str` doesn't match this format or represents an invalid date, a `ValueError` is raised. If successful, it returns a `datetime` object.
- `return date_obj.strftime("%Y-%m-%d")` : If the date is successfully parsed, this line formats the `datetime` object (`date_obj`) into a new string format `"%Y-%m-%d"`.
    - `%Y` : Year with century.
    - `%m` : Month as a zero-padded decimal number.
    - `%d` : Day of the month as a zero-padded decimal number. The formatted date string is then returned.
- `def run_date_validator():` : This defines a function `run_date_validator` that provides an interactive tool for validating and formatting dates.
- `print("Welcome to the Date of Birth Validator!")` : Prints a welcome message.
- `print("Please enter your date of birth in MM/DD/YYYY format. Type 'exit' to quit.")` : Provides instructions to the user.
- `while True:` : Starts an infinite loop for the interactive tool.
- `user_input = input("\nEnter your date of birth: ")` : Prompts the user to enter their date of birth.
- `if user_input.lower() == 'exit':` : Checks if the user entered "exit".
- `print("Exiting...")` : Prints an exiting message.
- `break` : Exits the loop.
- `formatted_date = validate_and_format_date(user_input)` : Calls the `validate_and_format_date` function with the user's input and stores the result.
- `if formatted_date == "Invalid Date":` : Checks if the returned value indicates an invalid date.
- `print("That is an invalid date. Please enter it in MM/DD/YYYY format (e.g., 01/15/2000).")` : Prints an error message and reminds the user of the correct format.
- `else:` : If the returned value is not "Invalid Date" (meaning it's a valid formatted date string).
- `print(f"Your date of birth, {user_input}, has been validated and formatted as: {formatted_date}")` : Prints a success message showing the original input and the validated and formatted date.
- `run_date_validator()` : This line calls the `run_date_validator` function to start the interactive tool.

In summary, this code provides a simple command-line interface for users to input a date in MM/DD/YYYY format, validates if it's a real date in that format, and if so, formats it into the YYYY-MM-DD format.