



CSE 312
Artificial Intelligence Lab
Lab Report on Basic Python Syntax

Submitted by :

Name : MD. Rahamtulla Ashik

ID: 043220005101007

Batch: 52

Section: 5A

Task No. 01:

Experiment Name / Title: Implementation of the Ancient Game of Nimm using Python

AIM: To design and implement a Python program that simulates the two-player game "Nimm," where players take turns removing stones from a pile. The goal is to observe the application of loops, conditional statements, and input validation in a simple interactive game.

DESCRIPTION:

Nimm is an ancient strategy game where two players take turns removing stones from a pile. The pile begins with 20 stones. Each player, on their turn, must remove either 1 or 2 stones. The game continues until there are no stones left. The player forced to take the last stone **loses**.

The game tests the player's strategic thinking, as they must plan their moves to avoid being the one who takes the last stone. This assignment aims to implement this game logic using Python, ensuring input validation and alternating turns for the two players.

ALGORITHM:

Initialize the game:

- Start with 20 stones in the pile.

While loop for the game:

- Keep the game running while the number of stones is greater than 0.

Alternate turns between Player 1 and Player 2:

- For each turn, prompt the current player to choose to remove either 1 or 2 stones.
- Ensure the input is valid (1 or 2).
- Update the pile by subtracting the number of stones chosen by the current player.

Check for the end condition:

- After each turn, check if the number of stones is 0.
- If so, declare the current player as the loser since they were forced to take the last stone.

Switch players:

- After each valid move, switch to the other player.

End of game:

- Once the game ends, output the winner and loser.

Optional (Play Again):

- Ask the players if they want to play again.

INPUT

```
def nimm_game():
    # Initialize the game with 20 stones = 20
    # Set the current player to Player 1 current_player =
    1

    # Start the game loop while stones > 0:
    print(f"\nStones remaining: {stones}") try:
        # Prompt the current player to take 1 or 2 stones
        take = int(input(f"Player {current_player}, take 1 or 2 stones:
"))
    except ValueError:
        print("Invalid input. Please enter 1 or 2.") continue

    # Ensure the player can only take 1 or 2 stones if take not in
    [1, 2]:
        print("Invalid move! You can only take 1 or 2 stones.") continue

    # Ensure the player doesn't take more stones than are available if take > stones:
    print(f"There are only {stones} stones left. You can't take
{take}.")
    continue

    # Subtract the number of stones taken from the pile stones -= take

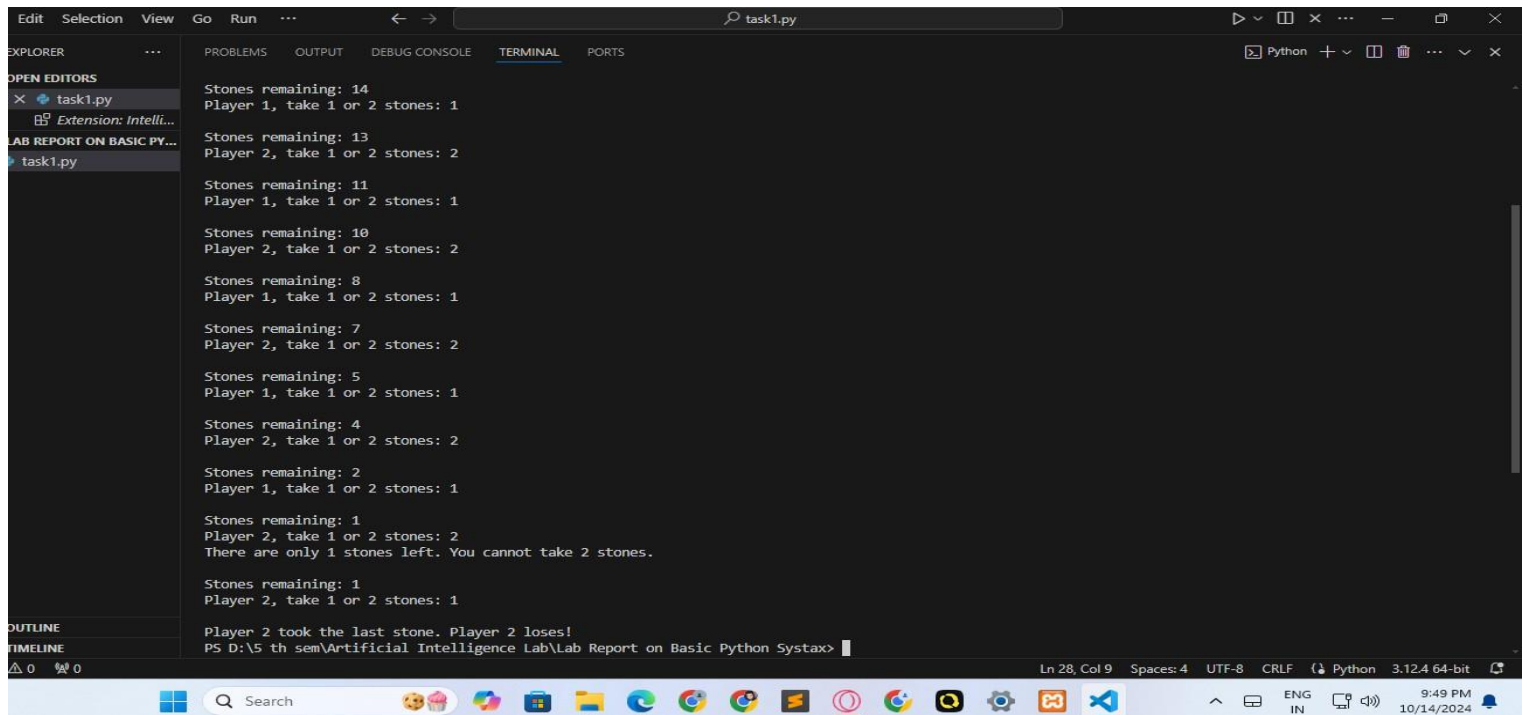
    # If no stones are left, the game ends if stones == 0:

        print(f"\nPlayer {current_player} took the last stone. Player
{current_player} loses!")
        break

    # Switch to the other player
    current_player = 2 if current_player == 1 else 1

# Run the game nimm_game()
```

OUTPUT



```
task1.py

Stones remaining: 14
Player 1, take 1 or 2 stones: 1

Stones remaining: 13
Player 2, take 1 or 2 stones: 2

Stones remaining: 11
Player 1, take 1 or 2 stones: 1

Stones remaining: 10
Player 2, take 1 or 2 stones: 2

Stones remaining: 8
Player 1, take 1 or 2 stones: 1

Stones remaining: 7
Player 2, take 1 or 2 stones: 2

Stones remaining: 5
Player 1, take 1 or 2 stones: 1

Stones remaining: 4
Player 2, take 1 or 2 stones: 2

Stones remaining: 2
Player 1, take 1 or 2 stones: 1

Stones remaining: 1
Player 2, take 1 or 2 stones: 2
There are only 1 stones left. You cannot take 2 stones.

Stones remaining: 1
Player 2, take 1 or 2 stones: 1

Player 2 took the last stone. Player 2 loses!
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>
```

DISCUSSION

This experiment demonstrates the importance of loops, conditionals, and input validation in game development. The program ensures the players can only take valid actions (1 or 2 stones), and it prevents invalid input such as letters or numbers outside the valid range. The game alternates turns using a simple player-switching mechanism (`current_player`), and it updates the number of stones dynamically based on each player's move. This game is a basic implementation of strategy and ensures players stay engaged by avoiding the last move that leads to losing the game. It also incorporates error handling to improve the user experience.

CONCLUSION

In this assignment, we successfully implemented the game of Nimm in Python. The program allowed two players to take turns removing stones from a pile and followed the game's rule that the player who takes the last stone loses. The game effectively used loops, conditionals, and input handling to manage the game flow and ensure a smooth player experience.

Task No. 02:

Experiment Name / Title: Implementation of the High-Low Guessing Game using Python

AIM: To design and implement a Python program that simulates the "High-Low" game. In this game, the computer picks a random secret number, and the player attempts to guess it. After each guess, the computer provides feedback on whether the guess was too high or too low. The goal is to create an interactive guessing game that continues until the player guesses correctly.

DESCRIPTION:

In the **High-Low** game, the computer randomly selects a secret number between 1 and 100. The player's task is to guess the number, and after each guess, the computer will respond with feedback indicating whether the guessed number is too high or too low. The player continues guessing until they find the correct number.

This assignment aims to demonstrate the use of random number generation, loops, conditionals, and input validation in Python.

ALGORITHM:

Random Number Generation:

- Use Python's `random` library to generate a random number between 1 and 100.

Game Loop:

- The game continues until the player correctly guesses the secret number.
- Prompt the player to guess the number.
- Ensure the player input is valid (it must be a number).

Provide Feedback:

- After each guess, compare the guessed number with the secret number:
 - If the guess is lower than the secret number, print "Too low!".
 - If the guess is higher than the secret number, print "Too high!".
 - If the guess is correct, print a congratulatory message and end the game.

Input Validation:

- If the player inputs anything other than a number, display an error message and prompt the player to guess again.

INTPUT :

```
import random

def high_low_game():
    # The computer picks a random number between 1 and 100
    secret_number = random.randint(1, 100)

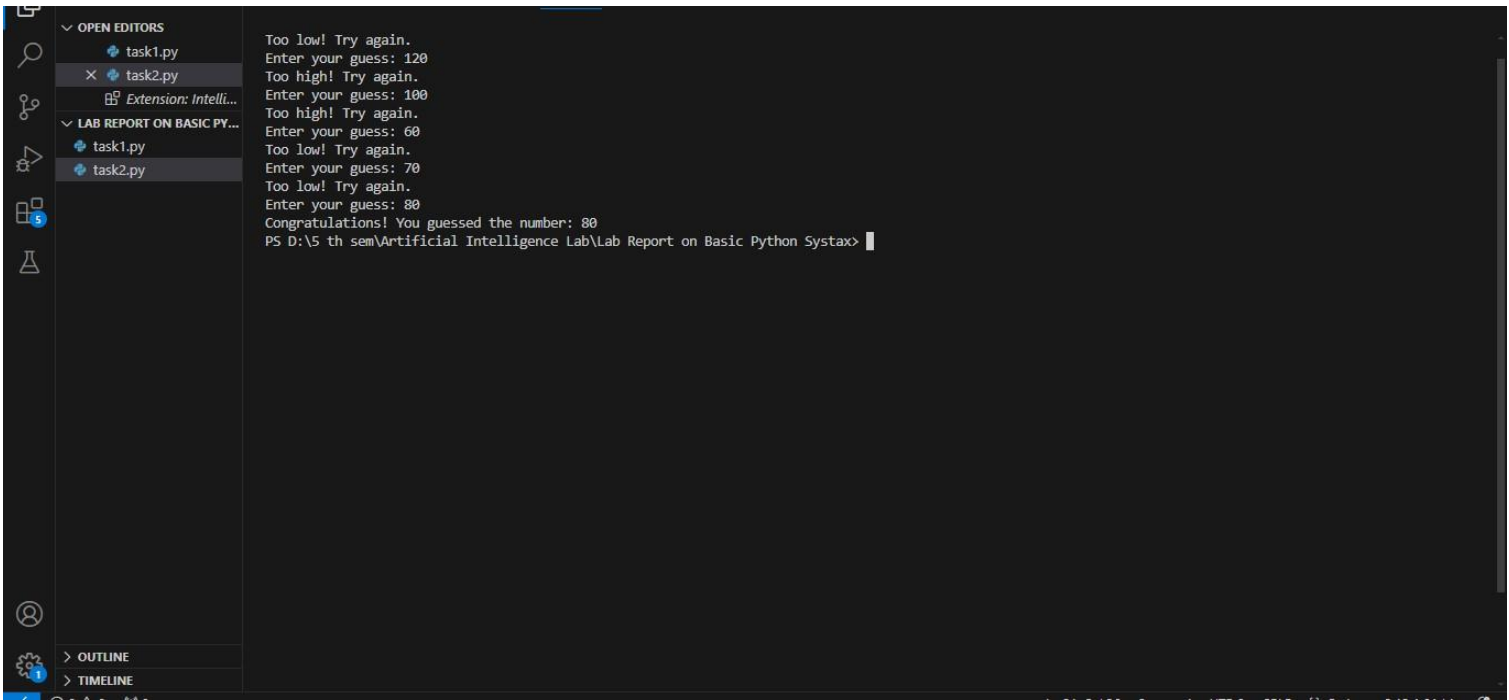
    print("Welcome to the High-Low Game!")
    print("I have picked a number between 1 and 100. Try to guess it!")

    while True:
        try:
            # Ask the player to guess the number
            guess = int(input("Enter your guess: "))
        except ValueError:
            # Handle invalid input
            print("Invalid input. Please enter a valid number.")
            continue

        # Provide feedback based on the player's guess
        if guess < secret_number:
            print("Too low! Try again.")
        elif guess > secret_number:
            print("Too high! Try again.")
        else:
            print(f"Congratulations! You guessed the number: {secret_number}")
            break

# Run the High-Low game
high_low_game()
```

OUTPUT



DISCUSSION

In this task, the program simulates a guessing game where the computer randomly selects a number and the player attempts to guess it. The program provides feedback in real-time, informing the player if their guess is too high or too low. The loop continues until the player guesses correctly, showcasing the use of conditional statements for feedback and a `while` loop to ensure the game keeps running until the winning condition is met. The program also incorporates **input validation** by ensuring that the player only enters valid numerical inputs, preventing the program from crashing due to invalid data types like letters or symbols.

CONCLUSION

The High-Low guessing game was successfully implemented in Python. The program selects a random secret number, prompts the player for guesses, and provides feedback based on the guessed number. The use of loops, conditionals, and input validation ensured that the game runs smoothly and handles incorrect inputs effectively. This assignment demonstrated how to create an interactive guessing game using Python.

Task No. 03:

Experiment Name / Title: Implementation of a Positive Affirmation Typing Program using Python

AIM: To design and implement a Python program that prompts the user to type a positive affirmation. The user must keep typing the affirmation correctly until they get it right. This program emphasizes input validation and repetition as a means to instill positive thinking.

DESCRIPTION:

In this task, the program will prompt the user to type a specific positive affirmation: **"I am capable of doing anything I put my mind to."** The goal of the program is to help reinforce a positive mindset by making the user repeat the affirmation until it is typed correctly.

The task showcases how to:

- Use loops to repeatedly ask for user input.
- Use conditional statements to validate whether the user's input matches the expected affirmation.
- Provide feedback to the user, encouraging them to try again if they make a mistake.

ALGORITHM:

Display the Affirmation:

- The program displays a message prompting the user to type the affirmation.

User Input:

- The user types the affirmation.

Check Input:

- The program compares the user's input with the correct affirmation.

Provide Feedback:

- If the input is incorrect, display a message and prompt the user to try again.
- If the input is correct, display a success message and end the program.

Repeat Until Correct:

- Continue prompting the user until they correctly type the affirmation.

INPUT:

```
def affirmation_program():
    correct_affirmation = "I am capable of doing anything I put my
mind to."

    print("Welcome! Let's reinforce a positive mindset.")

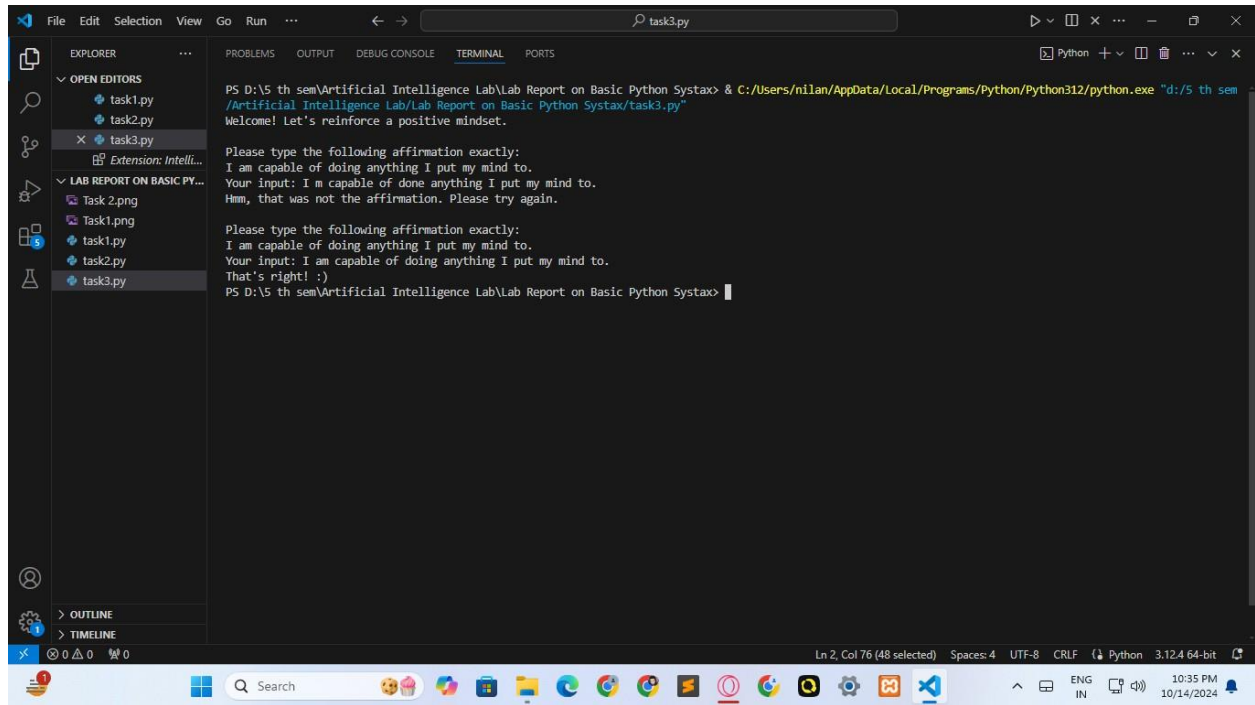
    while True:

        # Prompt the user to type the affirmation
        print("\nPlease type the following affirmation exactly:")
        print(correct_affirmation)
        user_input = input("Your input: ")

        # Check if the user typed the affirmation correctly
        if user_input == correct_affirmation:
            print("That's right! :)")
            break
        else:
            print("Hmm, that was not the affirmation. Please try
again.")

# Run the affirmation program
affirmation_program()
```

OUTPUT



```
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax> & C:/Users/nilan/AppData/Local/Programs/Python/Python312/python.exe "d:/5 th sem /Artificial Intelligence Lab/Lab Report on Basic Python Systax/task3.py"
Welcome! Let's reinforce a positive mindset.

Please type the following affirmation exactly:
I am capable of doing anything I put my mind to.
Your input: I m capable of done anything I put my mind to.
Hmm, that was not the affirmation. Please try again.

Please type the following affirmation exactly:
I am capable of doing anything I put my mind to.
Your input: I am capable of doing anything I put my mind to.
That's right! :)
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax> |
```

DISCUSSION

The positive affirmation program continuously prompts the user until they type the correct affirmation, reinforcing a positive mindset through repetition. This simple program demonstrates the following programming concepts:

- **Loops:** The `while` loop ensures that the program keeps asking for the correct input until the user gets it right.
- **Input validation:** The program checks the user's input against the correct affirmation and provides feedback.
- **Conditionals:** The program uses an `if-else` structure to determine whether the user's input matches the expected affirmation.

This kind of program could be used as a motivational tool, encouraging users to engage with positive self-talk and build confidence over time.

CONCLUSION

In this assignment, we successfully implemented a Python program that prompts the user to type a positive affirmation and checks if they type it correctly. This task involved the use of loops, input validation, and conditionals to provide feedback and repeat the process until the user typed the correct phrase. By focusing on positive affirmations, the program not only demonstrates basic programming skills but also serves as a motivational tool.

Task No. 04:

Experiment Name / Title: Backpack Item Management Simulation using Python

AIM: To design and implement a Python program that simulates managing items in a backpack using a list. The program allows the user to add, remove, view, check, and sort items in the backpack, demonstrating the use of list methods.

DESCRIPTION:

In this task, the user is on an adventure and needs to manage the contents of their backpack. The program will allow the following actions:

- **Add an item:** Add an item to the backpack if it's not already present.
- **Remove an item:** Remove an item if it exists in the backpack.
- **View the contents:** Display all the items currently in the backpack.
- **Check for an item:** Verify if a specific item is in the backpack.
- **Sort the items:** Display the items in the backpack in sorted order.

This assignment helps the user understand how to manipulate lists in Python using methods such as `.append()`, `.remove()`, `.sort()`, and `in`.

ALGORITHM:

- Initialize an empty list to represent the backpack.
- Display a menu of options for the user:
 - Add an item.
 - Remove an item.
 - View the contents of the backpack.
 - Check if an item is in the backpack.
 - Sort the items in the backpack.
- Process user input based on the selected option:
 - Add: If the item is not in the list, append it to the backpack.
 - Remove: If the item exists in the backpack, remove it.
 - View: Display all the items in the backpack.
 - Check: Check if an item exists in the backpack.
 - Sort: Sort the items in alphabetical order and display them.
- Continue offering the menu until the user decides to exit.

INPUT :

```
def backpack_manager():
    backpack = [] # Initialize an empty backpack

    while True:
        print("\n--- Backpack Manager ---") print("1. Add an item")
        print("2. Remove an item") print("3. View the
        contents") print("4. Check for an item") print("5.
        Sort the items") print("6. Exit")

        choice = input("Choose an option (1-6): ") if choice == '1':
            item = input("Enter the item to add: ") if item not in
            backpack:
                backpack.append(item)
                print(f"'{item}' has been added to the backpack.") else:
                print(f"'{item}' is already in the backpack.")

            elif choice == '2':
                item = input("Enter the item to remove: ") if item in backpack:
                    backpack.remove(item)
                    print(f"'{item}' has been removed from the
                    backpack.")
                else:
                    print(f"'{item}' is not in the backpack.")

            elif choice == '3': if backpack:
                print("The backpack contains the following items:") for item in backpack:
                    print(f"- {item}")
                else:
                    print("The backpack is empty.")

            elif choice == '4':
                item = input("Enter the item to check: ") if item in backpack:
                    print(f"'{item}' is in the backpack.") else:
                    print(f"'{item}' is not in the backpack.")
```

```

elif choice == '5': if backpack:

    backpack.sort()
    print("The backpack contents have been sorted:") for item in backpack:

        print(f"- {item}")

else:

    print("The backpack is empty, nothing to sort.")

elif choice == '6':

    print("Exiting the Backpack Manager. Goodbye!") break

else:

    print("Invalid choice, please select a valid option.")

# Run the backpack manager program backpack_manager()

```

OUTPUT

```

PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax> & C:/Users/nilan/AppData/Local/Programs/Python/Python312/python.exe "d:/5 th sem/Artificial Intelligence Lab/Lab Report on Basic Python Systax/task4.py"

--- Backpack Manager ---
1. Add an item
2. Remove an item
3. View the contents
4. Check for an item
5. Sort the items
6. Exit
Choose an option (1-6): 1
Enter the item to add: mango
'mango' has been added to the backpack.

--- Backpack Manager ---
1. Add an item
2. Remove an item
3. View the contents
4. Check for an item
5. Sort the items
6. Exit
Choose an option (1-6): 3
The backpack contains the following items:
- mango

--- Backpack Manager ---
1. Add an item
2. Remove an item
3. View the contents
4. Check for an item
5. Sort the items
6. Exit
Choose an option (1-6): 6
Exiting the Backpack Manager. Goodbye!
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>

```

DISCUSSION

This program provides a simulation of managing a backpack's contents during an adventure. It effectively uses list operations to add, remove, display, and sort items, giving the user a comprehensive understanding of basic list manipulation in Python. The options are displayed as a menu, allowing the user to interact with the backpack multiple times until they choose to exit.

The following key concepts are applied in this program:

- **List Methods:** `.append()`, `.remove()`, `.sort()`, and `in` are used to manipulate and query the contents of the backpack.
- **Loops and Conditionals:** The program uses a `while` loop to repeatedly display the menu until the user exits and employs conditional statements to handle user choices.

CONCLUSION

In this assignment, we successfully implemented a Python program that manages items in a backpack. The program allows the user to add, remove, view, check, and sort items, using list methods to handle the contents of the backpack. This program demonstrates the effective use of list operations in Python and helps users understand how to manage collections of data interactively.

Task No. 05:

Experiment Name / Title: Wizard's Spellbook Management using Python Dictionary

AIM: To design and implement a Python program that manages a wizard's spellbook using a dictionary. The program allows the user to add, update, remove, view, and find the most powerful spell in the spellbook.

DESCRIPTION:

In this task, the user is a wizard tasked with managing their spellbook, which contains spells and their associated power levels. The program uses a dictionary where the keys represent spell names, and the values represent the spell's power levels (as integers). The program allows the following actions:

- **Add a new spell:** Add a spell with a corresponding power level to the spellbook.
- **Update a spell's power:** Update the power level of an existing spell.
- **Remove a spell:** Remove a spell from the spellbook.
- **View all spells:** Display all the spells in the spellbook along with their power levels.
- **Find the most powerful spell:** Display the spell with the highest power level.

The program demonstrates the use of dictionary methods such as `.update()`, `.pop()`, and `max()` to manage the spellbook.

ALGORITHM:

- Initialize an empty dictionary to represent the spellbook.
- Display a menu of options:
 - Add a new spell.
 - Update a spell's power.
 - Remove a spell.
 - View all spells.
 - Find the most powerful spell.
- Process user input based on the selected option:
 - Add: Add the spell and its power level to the dictionary.
 - Update: Modify the power level of an existing spell.
 - Remove: Remove the spell from the dictionary.
 - View: Display all the spells and their power levels.
 - Find the most powerful spell: Display the spell with the maximum power level.
- Repeat the process until the user chooses to exit.

INPUT :

```
def spellbook_manager():  
    spellbook = {} # Initialize an empty spellbook  
  
    while True:  
        print("\n--- Wizard's Spellbook Manager ---") print("1. Add a new  
spell")  
        print("2. Update a spell's power") print("3. Remove a  
spell") print("4. View all spells")  
        print("5. Find the most powerful spell") print("6. Exit")  
  
        choice = input("Choose an option (1-6): ") if choice == '1':  
            spell = input("Enter the spell name: ")  
            power = int(input(f"Enter the power level for  
'{spell}': "))  
            spellbook.update({spell: power})  
            print(f"'{spell}' has been added to the spellbook with power level {power}.")  
  
        elif choice == '2':  
            spell = input("Enter the spell name to update: ") if spell in spellbook:  
                new_power = int(input(f"Enter the new power level for '{spell}': "))  
                spellbook[spell] = new_power
```

```

        print(f"'{spell}' has been updated to power level
{new_power}.")
    else:
        print(f"'{spell}' is not in the spellbook.")

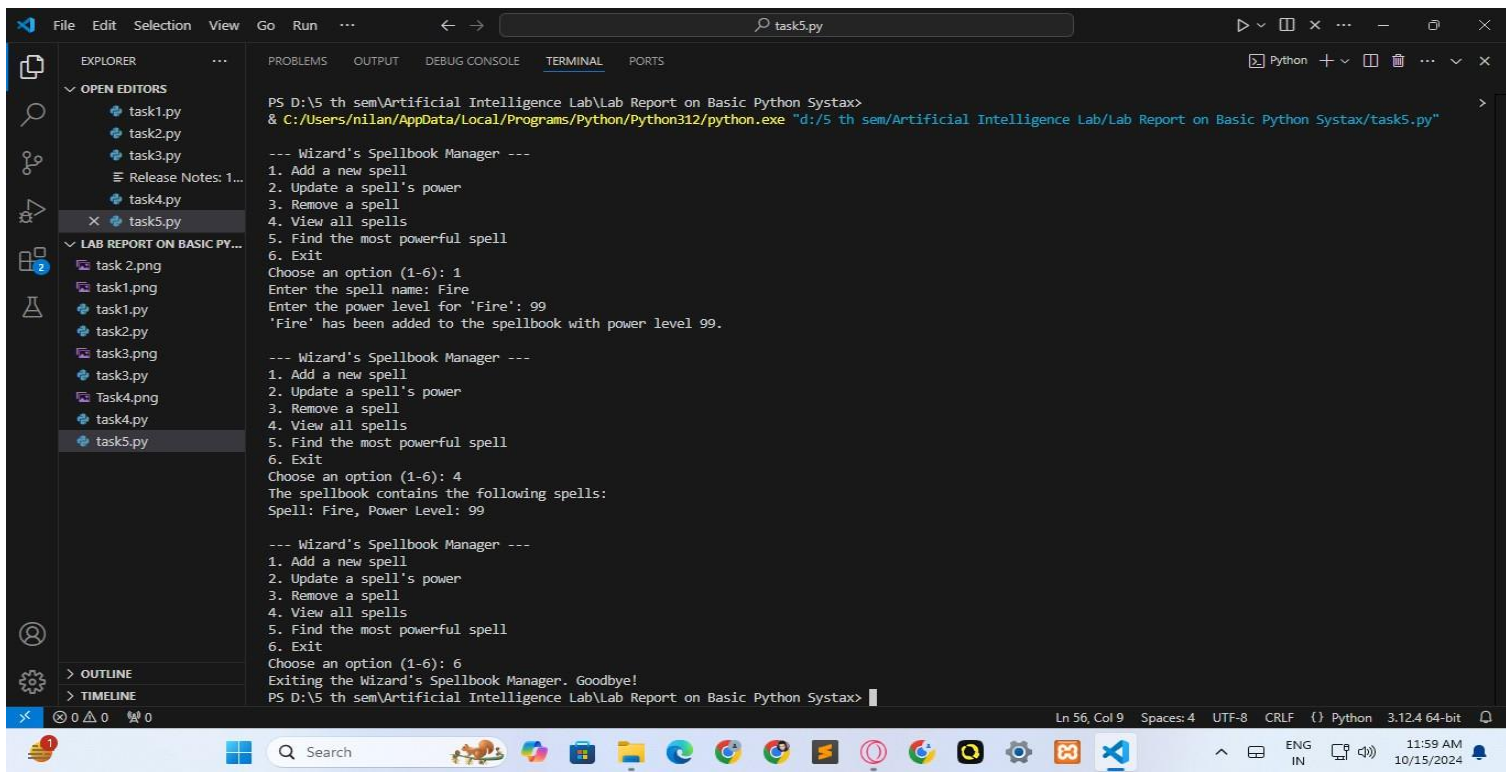
elif choice == '3':
    spell = input("Enter the spell name to remove: ") if spell in spellbook:
        spellbook.pop(spell)
        print(f"'{spell}' has been removed from the
spellbook.")
    else:
        print(f"'{spell}' is not in the spellbook.")

elif choice == '4': if spellbook:
        print("The spellbook contains the following
spells:")
        for spell, power in spellbook.items(): print(f"Spell: {spell}, Power
        Level: {power}")
    else:
        print("The spellbook is empty.")

elif choice == '5': if spellbook:
        most_powerful = max(spellbook, key=spellbook.get)
        print(f"The most powerful spell is a      power
        level of
        '{most_powerful}'      with
        {spellbook[most_powerful]}.")
        else: print("The spellbook is empty.")
    elif choice == '6':
        print("Exiting      the      Wizard's      Spellbook      Manager.
        Goodbye!")
        break
    else:
        print("Invalid choice, please select a valid option.")
spellbook_manager()

```


OUTPUT



```
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>
& C:/Users/nilan/AppData/Local/Programs/Python/Python312/python.exe "d:/5 th sem/Artificial Intelligence Lab/Lab Report on Basic Python Systax/task5.py"

--- Wizard's Spellbook Manager ---
1. Add a new spell
2. Update a spell's power
3. Remove a spell
4. View all spells
5. Find the most powerful spell
6. Exit
Choose an option (1-6): 1
Enter the spell name: Fire
Enter the power level for 'Fire': 99
'Fire' has been added to the spellbook with power level 99.

--- Wizard's Spellbook Manager ---
1. Add a new spell
2. Update a spell's power
3. Remove a spell
4. View all spells
5. Find the most powerful spell
6. Exit
Choose an option (1-6): 4
The spellbook contains the following spells:
Spell: Fire, Power Level: 99

--- Wizard's Spellbook Manager ---
1. Add a new spell
2. Update a spell's power
3. Remove a spell
4. View all spells
5. Find the most powerful spell
6. Exit
Choose an option (1-6): 6
Exiting the Wizard's Spellbook Manager. Goodbye!
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>
```

DISCUSSION

The wizard's spellbook management program allows the user to dynamically add, update, remove, view, and find the most powerful spell from a dictionary. This task provides a comprehensive demonstration of dictionary operations and key programming concepts such as:

- **Dictionaries:** Using key-value pairs to store and manage data (spells and their power levels).
- **Dictionary Methods:** Employing methods like `.update()`, `.pop()`, and `max()` to manipulate the contents of the dictionary.
- **Loops and Conditionals:** The program continuously offers the user a menu and processes input until they choose to exit, using `while` loops and `if-else` conditions.

This exercise helps reinforce the importance of dictionaries in Python for efficiently storing and retrieving data.

CONCLUSION

In this assignment, we successfully implemented a Python program that manages a wizard's spellbook using a dictionary. The program allows users to add, update, remove, view, and find the most powerful spell by making use of various dictionary methods. Through this task, the user gains valuable experience in managing key-value pairs in Python.

Task No. 06:

Experiment Name / Title: Finding Common Treasures Between Two Teams in a Treasure Hunt Event

AIM: To design a Python program that determines the common treasures collected by two teams participating in a treasure hunt event, using sets to manage the treasure collections of each team.

DESCRIPTION:

In a treasure hunt event, multiple teams collect unique treasures from different locations. Each team can gather multiple treasures, but no duplicates are allowed within their collection. Task 6 of this program is to find the treasures that are common between two teams. This functionality is essential for comparing the achievements of different teams and identifying any overlaps in their collections.

In Python, this can be efficiently handled using sets, which are unordered collections of unique elements. The intersection operation between two sets helps us determine the common elements (treasures) between two teams. The solution involves storing each team's treasure collection in a dictionary where the key is the team name and the value is a set of treasures.

ALGORITHM:

Initialize Data Structure:

- Create a dictionary where the keys are team names and the values are sets containing the treasures each team has collected.

Add Treasures:

- Allow the teams to add treasures to their collection using the `add_treasure()` function. Ensure that each treasure is unique for a team using Python sets.

Input Teams for Comparison:

- Input the names of two teams between which the common treasures are to be found. Check for Team

Existence:

- Verify that both teams exist in the dictionary. If one or both teams do not exist, print an error message.

Find Common Treasures:

- Use the `intersection` method for sets to find the common treasures between the two teams.

Output the Common Treasures:

- Display the common treasures to the user, or if there are no common treasures, inform the user that the teams do not share any.

```
INPUT:

# Dictionary to store each team's treasures (using duplicates)           sets to avoid
teams = {}

# 1. Add a treasure to a team's collection def
add_treasure(team_name, treasure):
    if team_name not in teams:  teams[team_name] = set()
    if treasure not in teams[team_name]:
        teams[team_name].add(treasure)
        print(f"Added treasure '{treasure}' to {team_name}'s
collection.")
    else:
        print(f"{team_name} already has the treasure
'{treasure}'.")

# 2. Remove a treasure from a team's collection def
remove_treasure(team_name, treasure):
    if team_name in teams and treasure in teams[team_name]:
        teams[team_name].remove(treasure)
        print(f"Removed treasure '{treasure}' from {team_name}'s collection.")
    else:
        print(f"{team_name} does not have the treasure
'{treasure}'.")

# 3. View all treasures of a team def
view_treasures(team_name):
    if team_name in teams:
        print(f"{team_name}'s treasures: {teams[team_name]}")
    else:
        print(f"{team_name} has no treasures.")

# 4. Find total number of treasures collected by a team def
total_treasures(team_name):
    if team_name in teams:
        print(f"{team_name} has collected {len(teams[team_name])} treasures.")
    else:
        print(f"{team_name} has no treasures.")
```

5. Check if a specific treasure is collected by a team def

```
check_treasure(team_name, treasure):
```

```
    if team_name in teams and treasure in teams[team_name]: print(f"Yes, {team_name} has
                                                                collected the treasure
'{treasure}'.")
```

```
    else:
```

```
        print(f"No, {team_name} has not collected the treasure '{treasure}'.")
```

6. Find common treasures between two teams def

```
common_treasures(team1, team2):
```

```
    if team1 in teams and team2 in teams:
```

```
        common = teams[team1].intersection(teams[team2]) print(f"Common treasures between
{team1} and {team2}:
```

```
{common}")
```

```
    else:
```

```
        print(f"One or both teams have no treasures.")
```

7. Find all treasures collected across all teams

```
def all_treasures(): all_treasures = set()
```

```
    for treasures in teams.values():
```

```
        all_treasures.update(treasures)
```

```
    print(f"All treasures collected by any team: {all_treasures}")
```

8. Find which team has collected the most treasures def

```
team_with_most_treasures():
```

```
    if not teams:
```

```
        print("No teams have collected any treasures.") return
```

```
        most_treasures_team = max(teams, key=lambda team: len(teams[team]))
        print(f"The team with the most treasures is
```

```
{most_treasures_team} with {len(teams[most_treasures_team])}
treasures.")
```

```

# Adding treasures
add_treasure("Team A", "Golden Crown") add_treasure("Team A", "Silver Sword")
add_treasure("Team B", "Silver Sword") add_treasure("Team B", "Emerald Ring")

# Viewing treasures view_treasures("Team A") view_treasures("Team B")

# Removing treasures
remove_treasure("Team A", "Golden Crown")

# Checking for a specific treasure check_treasure("Team A",
"Golden Crown")

# Finding common treasures between two teams
common_treasures("Team A", "Team B")

# Finding all treasures across all teams all_treasures()

# Finding the team with the most treasures
team_with_most_treasures()

```

OUTPUT

```

PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>
& C:/Users/nilan/AppData/Local/Programs/Python/Python312/python.exe "d:/5 th sem/Artificial Intelligence Lab/Lab Report on Basic Python Systax/task6.py"
Added treasure 'Golden Crown' to Team A's collection.
Added treasure 'Silver Sword' to Team A's collection.
Added treasure 'Silver Sword' to Team B's collection.
Added treasure 'Emerald Ring' to Team B's collection.
Team A's treasures: {'Golden Crown', 'Silver Sword'}
Team B's treasures: {'Silver Sword', 'Emerald Ring'}
Removed treasure 'Golden Crown' from Team A's collection.
No, Team A has not collected the treasure 'Golden Crown'.
Common treasures between Team A and Team B: {'Silver Sword'}
All treasures collected by any team: {'Silver Sword', 'Emerald Ring'}
The team with the most treasures is Team B with 2 treasures.
PS D:\5 th sem\Artificial Intelligence Lab\Lab Report on Basic Python Systax>

```

DISCUSSION

The program demonstrates how efficiently Python's `set` data structure can be used to find the common elements between two collections. This task is particularly useful in a scenario where teams in a treasure hunt event want to compare their achievements and discover overlapping treasures. The `intersection()` method is perfect for this situation as it allows fast comparison of the two sets.

This implementation assumes that the teams' data is stored properly in the dictionary and ensures that no duplicate treasures are stored within a team's collection, as sets inherently eliminate duplicates. The comparison between two teams is done in constant time concerning the number of treasures in each set.

CONCLUSION

This assignment was completed individually and any references or external resources, if used, were properly cited. The work represents my understanding and application of Python's set operations in solving a specific problem related to managing and comparing data in a treasure hunt event. No unauthorized collaboration or copying occurred during the completion of this task.