|  |  |  |
| --- | --- | --- |
|  | **THADOMAL SHAHANI ENGINEERING COLLEGE** |  |
| **DEPARTMENT OF INFORMATION TECHNOLOGY** |

**Roll no:I-62**

**2.Built in Datatypes: LO1**

**1)Aim:**

Develop a Python program to manage a task list using lists and tuples, including adding, removing, updating, and sorting tasks.

**Theory:**

* Lists are flexible as they can be modified by adding, removing, or updating their elements.
* Tuples are unchangeable, making them ideal for storing constant or fixed information.
* Lists can dynamically resize and provide efficient indexing, making them particularly useful for managing tasks.
* Tuples ensure data consistency by preventing unintentional changes.
* The slicing and manipulation of data are seamless with lists, enhancing their versatility.
* Due to their immutable nature, tuples are valid as dictionary keys.
* By using lists and tuples together, task-related metadata and states can be organized effectively.
* Lists come equipped with various built-in methods, such as append(), remove(), sort(), and reverse(), which expand their functionality.

**Program:**

print("Welcome to your TO-DO-LIST manager!")

task\_list = []

def view\_tasks():

if not task\_list:

print("Your To-Do List is empty.")

else:

print("\nYour To-Do List:")

for i, (task, status) in enumerate(task\_list, 1):

print(f"{i}. {task} - {'Done' if status else 'Pending'}")

while True:

print("\n---MENU---")

print("1. Add New Task")

print("2. Delete a Task")

print("3. Update Task Status")

print("4. Sort Tasks")

print("5. View To-Do List")

print("6. Exit")

choice = int(input("Enter your choice: "))

match choice:

case 1:

task = str(input("Enter New Task: "))

task\_list.append((task, False))

print(f"Your task '{task}' has been added.")

case 2:

view\_tasks()

if task\_list:

task\_no = int(input("Enter the task number to remove: "))

if 1 <= task\_no <= len(task\_list):

removed\_task = task\_list.pop(task\_no - 1)

print(f"Task '{removed\_task[0]}' removed successfully!")

else:

print("Invalid task number!")

case 3:

view\_tasks()

if task\_list:

task\_no = int(input("Enter the task number to update status: "))

if 1 <= task\_no <= len(task\_list):

task, \_ = task\_list[task\_no - 1]

task\_list[task\_no - 1] = (task, True) # Mark as done

print(f"Task '{task}' marked as done!")

else:

print("Invalid task number!")

case 4:

task\_list.sort(key=lambda x: (x[1], x[0]))

print("Tasks sorted by status and name.")

case 5:

view\_tasks()

case 6:

print("Exiting!")

break

case \_:

print("Oops! This is not a valid option.")

**Output:**

Welcome to your TO-DO-LIST manager!

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 1

Enter New Task: Complete Python Assignments

Your task 'Complete Python Assignments' has been added.

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 1

Enter New Task: Complete Maths

Your task 'Complete Maths' has been added.

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 3

Your To-Do List:

1. Complete Python Assignments - Pending

2. Complete Maths - Pending

Enter the task number to update status: 2

Task 'Complete Maths' marked as done!

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 4

Tasks sorted by status and name.

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 5

Your To-Do List:

1. Complete Python Assignments - Pending

2. Complete Maths - Done

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 2

Your To-Do List:

1. Complete Python Assignments - Pending

2. Complete Maths - Done

Enter the task number to remove: 2

Task 'Complete Maths' removed successfully!

---MENU---

1. Add New Task

2. Delete a Task

3. Update Task Status

4. Sort Tasks

5. View To-Do List

6. Exit

Enter your choice: 6

Exiting!

**Conclusion:**

Using lists and tuples makes managing tasks easier. Lists let you change things around, while tuples keep important information safe and unchanged.

**2)Aim:**

Create a Python code to demonstrate the use of sets and perform set operations (union, intersection, difference) to manage student enrollments in multiple courses / appearing for multiple entrance exams like CET, JEE, NEET etc.

**Theory:**

* Sets are collections of unique items without any specific order.
* They enable operations such as union, intersection, and difference.
* The union operation merges enrollments from several courses.
* Intersection finds students shared between multiple courses.
* Difference identifies students specific to certain courses.
* Duplicate entries are automatically eliminated in sets.
* Methods like add(), remove(), and discard() allow easy changes.
* Sets excel at quickly testing membership and removing duplicates.

**Program:**

def input\_students(exam\_name):

n = int(input(f"Enter the number of students giving {exam\_name}:"))

student\_set = set()

for \_ in range(n):

student = input("Student Name:")

student\_set.add(student)

return student\_set

cet\_students = input\_students("CET")

jee\_students = input\_students("JEE")

neet\_students = input\_students("NEET")

all\_students = cet\_students | jee\_students | neet\_students

common\_students = cet\_students & jee\_students & neet\_students

cet\_only = cet\_students - (jee\_students | neet\_students)

jee\_only = jee\_students - (cet\_students | neet\_students)

neet\_only = neet\_students - (jee\_students | cet\_students)

jee\_neet\_common = jee\_students & neet\_students

jee\_cet\_common = jee\_students & cet\_students

neet\_cet\_common = neet\_students & cet\_students

print("\nAll unique students appearing for at least one exam:", all\_students)

print("\nStudents appearing for all three exams (CET, JEE, NEET):", common\_students)

print("\nStudents appearing only for CET:", cet\_only)

print("\nStudents appearing only for JEE:", jee\_only)

print("\nStudents appearing only for NEET:", neet\_only)

print("\nStudents appearing for both JEE and NEET:", jee\_neet\_common)

print("\nStudents appearing for both JEE and CET:", jee\_cet\_common)

print("\nStudents appearing for both NEET and CET:", neet\_cet\_common)

**Output:**

Enter the number of students giving CET:5

Student Name:Dishita

Student Name:Ankit

Student Name:Shaurya

Student Name:Vanshita

Student Name:Bhoomi

Enter the number of students giving JEE:5

Student Name:Swanand

Student Name:Dishita

Student Name:Manav

Student Name:Vanshita

Student Name:Aayush

Enter the number of students giving NEET:3

Student Name:G

Student Name:Suhani

Student Name:Dishita

All unique students appearing for at least one exam: {'G', 'Shaurya', 'Aayush ', 'Manav ', 'Bhoomi', 'Suhani ', 'Swanand ', 'Dishita ', 'Vanshita', 'Ankit'}

Students appearing for all three exams (CET, JEE, NEET): {'Dishita '}

Students appearing only for CET: {'Bhoomi', 'Shaurya', 'Ankit'}

Students appearing only for JEE: {'Swanand ', 'Aayush ', 'Manav '}

Students appearing only for NEET: {'G', 'Suhani '}

Students appearing for both JEE and NEET: {'Dishita '}

Students appearing for both JEE and CET: {'Dishita ', 'Vanshita'}

Students appearing for both NEET and CET: {'Dishita '}

**Conclusion:**

Set operations help manage student data easily. They quickly show shared students, unique ones, and those only in certain groups.

**3)Aim:**

Write a Python program to create, update, and manipulate a dictionary of student records, including their grades and attendance.

**Theory:**

* Dictionaries organize data into key-value pairs.
* They enable fast lookups and easy updates.
* Ideal for storing student details like grades and attendance.
* Support nested structures, making them great for complex data.
* Handle missing or optional data effortlessly using default values.
* Methods like get(), update(), and pop() allow versatile data handling.
* Their hashable nature ensures quick and efficient data access.

**Program:**

student\_records = {

"student1": {"name": "Dishita", "grades": [85, 92, 78], "attendance": 100},

"student2": {"name": "Vanshita", "grades": [88, 76, 95], "attendance": 100},

"student3": {"name": "Ankit", "grades": [90, 88, 93], "attendance": 95},

}

def add\_or\_update\_student(student\_id, name, grades, attendance):

student\_records[student\_id] = {"name": name, "grades": grades, "attendance": attendance}

def display\_records():

for student\_id, record in student\_records.items():

print(f"ID: {student\_id}, Name: {record['name']}, Grades: {record['grades']}, Attendance: {record['attendance']}")

display\_records()

print("\nAdding a new student...\n")

add\_or\_update\_student("student4", "Manav", [89, 92, 85], 90)

display\_records()

print("\nUpdating student2...\n")

add\_or\_update\_student("student2", "Shaurya", [30, 35, 40], 19)

display\_records()

**Output:**

ID: student1, Name: Dishita, Grades: [85, 92, 78], Attendance: 100

ID: student2, Name: Vanshita, Grades: [88, 76, 95], Attendance: 100

ID: student3, Name: Ankit, Grades: [90, 88, 93], Attendance: 95

Adding a new student...

ID: student1, Name: Dishita, Grades: [85, 92, 78], Attendance: 100

ID: student2, Name: Vanshita, Grades: [88, 76, 95], Attendance: 100

ID: student3, Name: Ankit, Grades: [90, 88, 93], Attendance: 95

ID: student4, Name: Manav, Grades: [89, 92, 85], Attendance: 90

Updating student2...

ID: student1, Name: Dishita, Grades: [85, 92, 78], Attendance: 100

ID: student2, Name: Shaurya, Grades: [30, 35, 40], Attendance: 19

ID: student3, Name: Ankit, Grades: [90, 88, 93], Attendance: 95

ID: student4, Name: Manav, Grades: [89, 92, 85], Attendance: 90

**Conclusion:**

Dictionaries in Python are like labeled storage boxes for student data. They let you quickly find, update, or add information, keeping everything neat and manageable, even with many students.