Python Programming Guide - UNIT IV: Data Structures and File Handling

1. Lists

Lists are ordered, mutable collections that can contain elements of different types. They're one of the most versatile data structures in Python.

Creating and Accessing Lists

```
        # Creating lists
        fruits = ['apple', 'banana', 'cherry']
        numbers = [1, 2, 3, 4, 5]
        mixed = [1, 'hello', 3.14, True, None]
        empty_list = []

        # Accessing elements (0-indexed)
        print(fruits[0])           # Output: apple
        print(fruits[-1])          # Output: cherry (last element)
        print(fruits[1:3])         # Output: ['banana', 'cherry'] (slicing)
        print(fruits[:2])          # Output: ['apple', 'banana']
        print(fruits[1:])          # Output: ['banana', 'cherry']
        print(fruits[::2])         # Output: ['apple', 'cherry'] (every 2nd
element)
        print(fruits[::-1])        # Output: ['cherry', 'banana', 'apple']
(reversed)

        # List length
        print(len(fruits))         # Output: 3

        # Checking membership
        print('apple' in fruits)   # Output: True
        print('grape' in fruits)   # Output: False
```

List Operations and Methods

```
        # 1. Modifying lists
        fruits = ['apple', 'banana', 'cherry']

        # Append - add single element to end
        fruits.append('date')
```

```
        print(fruits)             # Output: ['apple', 'banana', 'cherry',
'date']

        # Extend - add multiple elements
        fruits.extend(['elderberry', 'fig'])
        print(fruits)             # Output: ['apple', 'banana', 'cherry',
'date', 'elderberry', 'fig']

        # Insert - add element at specific position
        fruits.insert(1, 'blueberry')
        print(fruits)             # Output: ['apple', 'blueberry',
'banana', 'cherry', ...]

        # Remove - remove first occurrence of value
        fruits.remove('blueberry')
        print(fruits)             # Output: ['apple', 'banana', 'cherry',
'date', ...]

        # Pop - remove and return element at index (default: last)
        last = fruits.pop()
        print(f"Removed: {last}")   # Output: Removed: fig
        print(fruits)

        # Clear - remove all elements
        # fruits.clear()

        # 2. Finding and counting
        numbers = [1, 2, 3, 2, 4, 2, 5]

        # Count occurrences
        print(numbers.count(2))    # Output: 3

        # Index of first occurrence
        print(numbers.index(4))    # Output: 4

        # 3. Sorting
        numbers = [5, 2, 8, 1, 9]
        numbers.sort()             # Sort in place (ascending)
        print(numbers)             # Output: [1, 2, 5, 8, 9]

        numbers.sort(reverse=True) # Sort descending
        print(numbers)             # Output: [9, 8, 5, 2, 1]

        # Reverse
        numbers.reverse()
        print(numbers)             # Output: [1, 2, 5, 8, 9]

        # 4. List comprehension
        # Create new list based on existing list
        squares = [x**2 for x in range(1, 6)]
        print(squares)             # Output: [1, 4, 9, 16, 25]

        # With condition
        evens = [x for x in range(10) if x % 2 == 0]
        print(evens)               # Output: [0, 2, 4, 6, 8]

        # Nested comprehension
        matrix = [[i*j for j in range(1, 4)] for i in range(1, 4)]
        print(matrix)              # Output: [[1, 2, 3], [2, 4, 6], [3, 6,
9]]
```

```
# 5. Copying lists
original = [1, 2, 3]
shallow_copy = original.copy()
deep_copy = original[:]

original.append(4)
print(f"Original: {original}")       # [1, 2, 3, 4]
print(f"Shallow copy: {shallow_copy}") # [1, 2, 3]
print(f"Deep copy: {deep_copy}")     # [1, 2, 3]
```

Real-World Application: Student Grade Management

```
class StudentGradeManager:
    """Manage student grades using lists."""

    def __init__(self):
        self.students = []

    def add_student(self, name, grade):
        """Add a new student."""
        self.students.append({'name': name, 'grade': grade})

    def get_top_students(self, n=5):
        """Get top N students by grade."""
        sorted_students = sorted(self.students, key=lambda s:
s['grade'], reverse=True)
        return sorted_students[:n]

    def get_average_grade(self):
        """Calculate average grade."""
        if not self.students:
            return 0
        total = sum(s['grade'] for s in self.students)
        return total / len(self.students)

    def get_students_above_threshold(self, threshold):
        """Get students with grade above threshold."""
        return [s for s in self.students if s['grade'] >= threshold]

    def display_statistics(self):
        """Display grade statistics."""
        if not self.students:
            print("No students in database")
            return

        grades = [s['grade'] for s in self.students]
        print(f"\n{'='*50}")
        print("GRADE STATISTICS")
        print(f"{'='*50}")
        print(f"Total Students: {len(self.students)}")
        print(f"Average Grade: {self.get_average_grade():.2f}")
        print(f"Highest Grade: {max(grades)}")
        print(f"Lowest Grade: {min(grades)}")
        print(f"{'='*50}")

# Usage
manager = StudentGradeManager()
students_data = [
    ("Alice", 95), ("Bob", 87), ("Charlie", 92),
    ("Diana", 78), ("Eve", 88), ("Frank", 85)
```

```
]

for name, grade in students_data:
    manager.add_student(name, grade)

manager.display_statistics()

print("\nTop 3 Students:")
for student in manager.get_top_students(3):
    print(f"  {student['name']}: {student['grade']}")

print("\nStudents with grade >= 85:")
for student in manager.get_students_above_threshold(85):
    print(f"  {student['name']}: {student['grade']}")
```

---

2. Tuples

Tuples are ordered, immutable collections. Once created, they cannot be modified.

```
# Creating tuples
colors = ('red', 'green', 'blue')
single_tuple = (42,)           # Single element needs comma
empty_tuple = ()
numbers = (1, 2, 3, 4, 5)

# Accessing elements (same as lists)
print(colors[0])               # Output: red
print(colors[-1])              # Output: blue
print(colors[1:3])             # Output: ('green', 'blue')

# Tuple length
print(len(colors))             # Output: 3

# Membership
print('red' in colors)         # Output: True

# Tuple unpacking
r, g, b = colors
print(f"Red: {r}, Green: {g}, Blue: {b}")

# Tuple methods
t = (1, 2, 3, 2, 1)
print(t.count(2))              # Output: 2 (count occurrences)
print(t.index(3))              # Output: 2 (find index)

# Converting to tuple
list_to_tuple = tuple([1, 2, 3])
print(list_to_tuple)           # Output: (1, 2, 3)

# Tuple comprehension (generates generator)
squares = tuple(x**2 for x in range(5))
print(squares)                 # Output: (0, 1, 4, 9, 16)

# Immutability
try:
    colors[0] = 'yellow'       # This will raise TypeError
except TypeError:
```

```
        print("Cannot modify tuple")

    # Returning multiple values (tuple unpacking)
    def get_coordinates():
        return (10, 20, 30)

    x, y, z = get_coordinates()
    print(f"Coordinates: ({x}, {y}, {z})")

    # Nested tuples
    matrix = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
    print(matrix[0][1])            # Output: 2

    # Tuple concatenation
    t1 = (1, 2)
    t2 = (3, 4)
    combined = t1 + t2             # Output: (1, 2, 3, 4)

    # Tuple repetition
    repeated = (1, 2) * 3          # Output: (1, 2, 1, 2, 1, 2)
```

Real-World Application: Coordinate and Location Management

```
    class LocationTracker:
        """Track locations using tuples."""

        def __init__(self):
            self.locations = []  # List of tuples: (name, latitude,
longitude, altitude)

        def add_location(self, name, latitude, longitude, altitude=0):
            """Add a location as a tuple."""
            self.locations.append((name, latitude, longitude, altitude))

        def distance_between_points(self, lat1, lon1, lat2, lon2):
            """Simple distance calculation (Haversine would be
better)."""
            import math
            return math.sqrt((lat2 - lat1)**2 + (lon2 - lon1)**2)

        def get_location_info(self, name):
            """Get information about a location."""
            for location in self.locations:
                loc_name, lat, lon, alt = location
                if loc_name == name:
                    return {
                        'name': loc_name,
                        'latitude': lat,
                        'longitude': lon,
                        'altitude': alt
                    }
            return None

        def find_nearest_location(self, latitude, longitude):
            """Find nearest location to given coordinates."""
            if not self.locations:
                return None

            nearest = None
            min_distance = float('inf')
```

```
                for location in self.locations:
                    _, lat, lon, _ = location
                    dist = self.distance_between_points(latitude, longitude,
lat, lon)
                    if dist < min_distance:
                        min_distance = dist
                        nearest = location

                return nearest

        # Usage
        tracker = LocationTracker()
        tracker.add_location("Office", 40.7128, -74.0060, 100)
        tracker.add_location("Home", 40.7589, -73.9851, 50)
        tracker.add_location("Gym", 40.7614, -73.9776, 80)

        nearest = tracker.find_nearest_location(40.750, -73.990)
        print(f"Nearest location: {nearest[0]} at {nearest[1]},
{nearest[2]}")
```

---

3. Sets

Sets are unordered, mutable collections with no duplicate elements. They're useful for membership testing and removing duplicates.

```
        # Creating sets
        colors = {'red', 'green', 'blue'}
        numbers = {1, 2, 3, 4, 5}
        empty_set = set()            # Note: {} creates dict, not set
        mixed = {1, 'hello', (1, 2)} # Can contain hashable types (not
lists)

        print(colors)                # Output: {'red', 'blue', 'green'}
(order may vary)
        print(len(colors))           # Output: 3

        # Membership testing
        print('red' in colors)       # Output: True
        print('yellow' in colors)    # Output: False

        # Adding elements
        colors.add('yellow')
        print(colors)

        # Removing elements
        colors.discard('yellow')     # Won't raise error if not found
        print(colors)

        colors.remove('red')         # Raises KeyError if not found
        print(colors)

        # Set operations
        set1 = {1, 2, 3, 4}
        set2 = {3, 4, 5, 6}

        # Union - all elements from both sets
        union = set1 | set2
        print(f"Union: {union}")     # Output: {1, 2, 3, 4, 5, 6}
```

```
        # Intersection - elements common to both
        intersection = set1 & set2
        print(f"Intersection: {intersection}")  # Output: {3, 4}

        # Difference - elements in set1 but not in set2
        difference = set1 - set2
        print(f"Difference: {difference}")      # Output: {1, 2}

        # Symmetric difference - elements in either but not both
        sym_diff = set1 ^ set2
        print(f"Symmetric Difference: {sym_diff}")  # Output: {1, 2, 5, 6}

        # Set methods
        set1.update({5, 6})          # Add multiple elements
        print(set1)

        set1.pop()                    # Remove arbitrary element
        print(set1)

        set1.clear()                  # Remove all elements

        # Removing duplicates using set
        numbers_with_duplicates = [1, 2, 2, 3, 3, 3, 4, 5, 5]
        unique_numbers = list(set(numbers_with_duplicates))
        print(f"Unique: {sorted(unique_numbers)}")  # Output: [1, 2, 3, 4,
5]

        # Set comprehension
        squares = {x**2 for x in range(6)}
        print(squares)               # Output: {0, 1, 4, 9, 16, 25}
```

Real-World Application: Unique Visitor Tracking

```
        class VisitorTracker:
            """Track unique visitors to a website."""

            def __init__(self):
                self.visitors = set()
                self.daily_visitors = {}

            def add_visitor(self, visitor_id, date=None):
                """Add a unique visitor."""
                self.visitors.add(visitor_id)

                if date:
                    if date not in self.daily_visitors:
                        self.daily_visitors[date] = set()
                    self.daily_visitors[date].add(visitor_id)

            def get_total_unique_visitors(self):
                """Get total number of unique visitors."""
                return len(self.visitors)

            def get_returning_visitors(self, date1, date2):
                """Get visitors who visited on both dates."""
                if date1 in self.daily_visitors and date2 in
self.daily_visitors:
                    return self.daily_visitors[date1] &
self.daily_visitors[date2]
                return set()
```

```
            def get_new_visitors(self, date, previous_dates):
                """Get visitors new to the website."""
                if date not in self.daily_visitors:
                    return set()

                all_previous = set()
                for prev_date in previous_dates:
                    if prev_date in self.daily_visitors:
                        all_previous.update(self.daily_visitors[prev_date])

                return self.daily_visitors[date] - all_previous

            def get_popular_visit_time(self):
                """Get date with most visitors."""
                if not self.daily_visitors:
                    return None
                return max(self.daily_visitors.items(), key=lambda x:
len(x[1]))

        # Usage
        tracker = VisitorTracker()

        # Day 1 visitors
        for visitor in ['user1', 'user2', 'user3', 'user4', 'user5']:
            tracker.add_visitor(visitor, '2026-02-01')

        # Day 2 visitors (some returning, some new)
        for visitor in ['user2', 'user3', 'user6', 'user7']:
            tracker.add_visitor(visitor, '2026-02-02')

        print(f"Total unique visitors:
{tracker.get_total_unique_visitors()}")

        returning = tracker.get_returning_visitors('2026-02-01', '2026-02-
02')
        print(f"Returning visitors: {returning}")

        new_visitors = tracker.get_new_visitors('2026-02-02', ['2026-02-
01'])
        print(f"New visitors on Feb 2: {new_visitors}")

        busiest = tracker.get_popular_visit_time()
        print(f"Busiest day: {busiest[0]} with {len(busiest[1])} visitors")
```

---

4. Dictionaries

Dictionaries are unordered, mutable collections of key-value pairs. They're extremely useful for structured data.

```
        # Creating dictionaries
        person = {'name': 'Alice', 'age': 25, 'city': 'NYC'}
        student = dict(name='Bob', age=20, major='Computer Science')
        empty_dict = {}

        # Accessing values
        print(person['name'])              # Output: Alice
        print(person.get('age'))           # Output: 25 (safe access)
```

```
        print(person.get('phone', 'N/A'))  # Output: N/A (default value)

        # Modifying dictionaries
        person['age'] = 26                 # Update value
        person['email'] = 'alice@example.com'  # Add new key-value
        del person['city']                 # Delete key-value pair

        # Dictionary methods
        print(person.keys())               # View keys
        print(person.values())             # View values
        print(person.items())              # View key-value pairs

        person.update({'age': 27, 'phone': '123-456-7890'})  # Update
multiple

        # Dictionary comprehension
        squares = {x: x**2 for x in range(5)}
        print(squares)                     # Output: {0: 0, 1: 1, 2: 4, 3:
9, 4: 16}

        # Nested dictionaries
        company = {
            'name': 'TechCorp',
            'employees': {
                'e1': {'name': 'Alice', 'salary': 50000},
                'e2': {'name': 'Bob', 'salary': 45000}
            }
        }

        print(company['employees']['e1']['name'])  # Output: Alice

        # Iterating through dictionary
        for key, value in person.items():
            print(f"{key}: {value}")

        # Check if key exists
        if 'name' in person:
            print("Name exists")

        # Get and remove
        value = person.pop('email', 'Not found')
        print(f"Removed: {value}")

        # Get all values of same type
        scores = {'math': 85, 'english': 90, 'science': 88}
        total = sum(scores.values())
        average = total / len(scores)
        print(f"Average score: {average:.2f}")
```

Real-World Application: User Profile Management

```
        class UserProfileManager:
            """Manage user profiles using dictionaries."""

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

            def create_profile(self, user_id, name, email,
preferences=None):
                """Create a new user profile."""
                self.users[user_id] = {
```

```
            'name': name,
            'email': email,
            'created_date': '2026-02-03',
            'last_login': None,
            'preferences': preferences or {},
            'activity_log': []
        }

    def update_profile(self, user_id, **kwargs):
        """Update user profile information."""
        if user_id in self.users:
            for key, value in kwargs.items():
                if key in self.users[user_id]:
                    self.users[user_id][key] = value
            return True
        return False

    def log_activity(self, user_id, activity):
        """Log user activity."""
        if user_id in self.users:
            self.users[user_id]['activity_log'].append({
                'timestamp': '2026-02-03 12:00:00',
                'activity': activity
            })

    def get_user_summary(self, user_id):
        """Get summary of user profile."""
        if user_id in self.users:
            user = self.users[user_id]
            return {
                'id': user_id,
                'name': user['name'],
                'email': user['email'],
                'activities': len(user['activity_log']),
                'preferences_set': len(user['preferences'])
            }
        return None

    def find_users_by_email_domain(self, domain):
        """Find all users with email from specific domain."""
        matching_users = {}
        for user_id, user in self.users.items():
            if user['email'].endswith(f"@{domain}"):
                matching_users[user_id] = user['name']
        return matching_users

# Usage
manager = UserProfileManager()

manager.create_profile(
    'u1', 'Alice Johnson', 'alice@company.com',
    {'theme': 'dark', 'notifications': True}
)
manager.create_profile(
    'u2', 'Bob Smith', 'bob@company.com',
    {'theme': 'light', 'notifications': False}
)

manager.log_activity('u1', 'Viewed dashboard')
manager.log_activity('u1', 'Updated profile')
```

```
summary = manager.get_user_summary('u1')
print("User Summary:")
for key, value in summary.items():
    print(f"  {key}: {value}")

print("\nCompany email users:")
company_users = manager.find_users_by_email_domain('company.com')
for user_id, name in company_users.items():
    print(f"  {user_id}: {name}")
```

---

5. Comparison of Data Structures

Feature	List	Tuple	Set	Dictionary
Ordered	Yes	Yes	No	No (3.7+ preserves insertion)
Mutable	Yes	No	Yes	Yes
Indexable	Yes	Yes	No	By key
Duplicates	Allowed	Allowed	Not allowed	Keys unique
Use Case	Flexible collection	Fixed data	Unique items	Key-value pairs
Speed	O(n) lookup	O(n) lookup	O(1) lookup	O(1) lookup

```
# Practical comparison
import time

# Creating test data
list_data = list(range(100000))
tuple_data = tuple(range(100000))
set_data = set(range(100000))
dict_data = {i: i for i in range(100000)}

# Lookup speed comparison
# Lists and tuples are O(n) - slow for large datasets
# Sets and dictionaries are O(1) - fast constant time

# Choose based on requirements:
# - List: ordered, mutable collection
# - Tuple: fixed, hashable, immutable collection
# - Set: unique items, fast membership testing
# - Dictionary: key-value associations
```

---

6. File Handling

File handling is essential for reading, writing, and manipulating files.

File Types

Python can work with various file types:

- **Text Files**: .txt, .csv, .json, .html, .xml
- **Binary Files**: .pdf, .jpg, .png, .exe, .zip
- **Code Files**: .py, .java, .cpp, .js

File Modes

```
# File modes when opening files:
# 'r'   - Read (default) - file must exist
# 'w'   - Write - creates new or overwrites existing
# 'a'   - Append - adds to end of file
# 'x'   - Create - fails if file exists
# 'b'   - Binary mode (combine with above, e.g., 'rb', 'wb')
# '+'   - Read and write (e.g., 'r+', 'w+')

# Examples:
# 'r'    - Read text
# 'rb'   - Read binary
# 'w'    - Write text
# 'wb'   - Write binary
# 'a'    - Append text
# 'ab'   - Append binary
# 'r+'   - Read and write text
```

Reading Files

```
# Method 1: read() - Read entire file
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)

# Method 2: readline() - Read one line
with open('data.txt', 'r') as file:
    line = file.readline()
    print(line)

# Method 3: readlines() - Read all lines as list
with open('data.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())

# Method 4: iterate through file
with open('data.txt', 'r') as file:
    for line in file:
        print(line.strip())

# Reading specific number of characters
with open('data.txt', 'r') as file:
    partial = file.read(100)  # Read first 100 characters
    print(partial)
```

Writing Files

```
# Write mode (overwrites existing content)
with open('output.txt', 'w') as file:
    file.write("Hello, World!\n")
    file.write("This is a new file.\n")
```

```
        # Append mode (adds to existing file)
        with open('output.txt', 'a') as file:
            file.write("Additional line.\n")

        # Write multiple lines
        lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
        with open('output.txt', 'w') as file:
            file.writelines(lines)

        # Using print() to write
        with open('output.txt', 'w') as file:
            print("Hello from print()", file=file)
            print("Another line", file=file)
```

Real-World Application: Log File Management

```
        import os
        from datetime import datetime

        class LogFileManager:
            """Manage application log files."""

            def __init__(self, log_file='application.log'):
                self.log_file = log_file
                self.ensure_file_exists()

            def ensure_file_exists(self):
                """Create log file if it doesn't exist."""
                if not os.path.exists(self.log_file):
                    with open(self.log_file, 'w') as f:
                        f.write("LOG FILE CREATED\n")

            def log_event(self, event_type, message, severity='INFO'):
                """Log an event with timestamp."""
                timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                log_entry = f"[{timestamp}] [{severity}] [{event_type}]
{message}\n"

                with open(self.log_file, 'a') as f:
                    f.write(log_entry)

            def read_logs(self, limit=None):
                """Read log entries."""
                try:
                    with open(self.log_file, 'r') as f:
                        lines = f.readlines()

                    if limit:
                        return lines[-limit:]  # Last N lines
                    return lines

                except FileNotFoundError:
                    return []

            def search_logs(self, keyword):
                """Search for entries containing keyword."""
                matching_lines = []
                try:
                    with open(self.log_file, 'r') as f:
                        for line in f:
                            if keyword.lower() in line.lower():
```

```
                        matching_lines.append(line.strip())
            except FileNotFoundError:
                pass

            return matching_lines

        def get_file_stats(self):
            """Get log file statistics."""
            try:
                file_size = os.path.getsize(self.log_file)
                with open(self.log_file, 'r') as f:
                    line_count = len(f.readlines())

                return {
                    'file_size': file_size,
                    'line_count': line_count,
                    'exists': True
                }
            except FileNotFoundError:
                return {'exists': False}

    # Usage
    # logger = LogFileManager()
    # logger.log_event('USER', 'User logged in', 'INFO')
    # logger.log_event('ERROR', 'Database connection failed', 'ERROR')
    # print(logger.read_logs(limit=5))
    # print(logger.search_logs('ERROR'))
    # print(logger.get_file_stats())
```

Working with CSV Files

```
    import csv

    # Writing CSV file
    data = [
        ['Name', 'Age', 'City'],
        ['Alice', 25, 'NYC'],
        ['Bob', 30, 'LA'],
        ['Charlie', 28, 'Chicago']
    ]

    # Method 1: Using writerows()
    with open('data.csv', 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerows(data)

    # Method 2: Using writerow()
    with open('data.csv', 'w', newline='') as f:
        writer = csv.writer(f)
        for row in data:
            writer.writerow(row)

    # Reading CSV file
    with open('data.csv', 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            print(row)

    # Reading as dictionaries
    with open('data.csv', 'r') as f:
```

```
    reader = csv.DictReader(f)
    for row in reader:
        print(row)  # {'Name': 'Alice', 'Age': '25', ...}
```

Working with JSON Files

```
import json

# Writing JSON file
person = {
    'name': 'Alice',
    'age': 25,
    'city': 'NYC',
    'hobbies': ['reading', 'gaming', 'swimming']
}

with open('person.json', 'w') as f:
    json.dump(person, f, indent=2)

# Reading JSON file
with open('person.json', 'r') as f:
    loaded_person = json.load(f)
    print(loaded_person)

# JSON string operations
json_string = json.dumps(person, indent=2)
print(json_string)

parsed = json.loads(json_string)
print(parsed)
```

Real-World Application: JSON Data Processing

```
import json
import os

class JSONDataManager:
    """Manage JSON data files."""

    @staticmethod
    def save_data(filename, data):
        """Save data to JSON file."""
        try:
            with open(filename, 'w') as f:
                json.dump(data, f, indent=2)
            return True
        except Exception as e:
            print(f"Error saving file: {e}")
            return False

    @staticmethod
    def load_data(filename):
        """Load data from JSON file."""
        try:
            with open(filename, 'r') as f:
                return json.load(f)
        except FileNotFoundError:
            return None
        except json.JSONDecodeError:
            return None
```

```
    @staticmethod
    def append_record(filename, record):
        """Append a record to JSON file (array)."""
        data = JSONDataManager.load_data(filename)
        if data is None:
            data = []

        if isinstance(data, list):
            data.append(record)
            JSONDataManager.save_data(filename, data)
            return True
        return False

    @staticmethod
    def update_record(filename, record_id, updated_data):
        """Update a record in JSON array."""
        data = JSONDataManager.load_data(filename)
        if data and isinstance(data, list):
            for i, record in enumerate(data):
                if record.get('id') == record_id:
                    record.update(updated_data)
                    JSONDataManager.save_data(filename, data)
                    return True
        return False

# Usage
# manager = JSONDataManager()
#
# # Save user data
# users = [
#     {'id': 1, 'name': 'Alice', 'email': 'alice@example.com'},
#     {'id': 2, 'name': 'Bob', 'email': 'bob@example.com'}
# ]
# manager.save_data('users.json', users)
#
# # Load and display
# loaded_users = manager.load_data('users.json')
# print(loaded_users)
#
# # Add new user
# manager.append_record('users.json', {'id': 3, 'name': 'Charlie',
'email': 'charlie@example.com'})
```

Summary of Unit IV

This unit covers all major data structures and file operations:

- **Lists**: Ordered, mutable, versatile collection
- **Tuples**: Ordered, immutable, fixed collections
- **Sets**: Unordered, unique elements, fast operations
- **Dictionaries**: Key-value pairs, fast lookups
- **File Handling**: Reading, writing, and manipulating files
- **File Formats**: Text, CSV, JSON, and binary files

Practice Exercises for Unit IV

1. Create a student management system using lists and dictionaries
2. Build a contact book using dictionaries
3. Write a program that reads and processes CSV data
4. Create a JSON data manager with CRUD operations
5. Build a file organizer that categorizes files by type

Next Unit: Unit V - Object-Oriented Programming and Exception Handling