Python Intensive Day 3: Expanded Data Structures and File Handling

---

Python Intensive Day 3: Expanded Data Structures and File Handling

---

Objective:

By the end of Day 3, students will have a deep understanding of Python's built-in data structures (lists, tuples, sets, dictionaries), file handling, and the ability to manipulate data effectively. Several mini-projects and hands-on exercises will be included to ensure comprehensive coverage of the topics.

1.1 Recap of Day 2 (15 minutes)

Goal:

Quickly review the key concepts covered in Day 2 to ensure everyone is prepared for new topics.

Topics to Review:

- Control flow (`if-else` statements, loops).
- Functions (defining, calling, and using parameters and return values).
- Practice exercises and mini-projects completed.

1.2 Introduction to Data Structures (20 minutes)

Why This is Important:

Data structures are essential for organizing and manipulating data efficiently in programming. Python's built-in data structures provide flexibility and powerful tools for handling different types of data.

1.3 Lists (60 minutes)

What is a List?:

- A **list** is a mutable, ordered collection of items, allowing for indexing, slicing, and modification.
- **Syntax**:

```
my_list = [1, 2, 3, 4, 5]
```

Basic List Operations:

- **Accessing Elements**:
  - Lists use zero-based indexing to access elements.

```
first_element = my_list[0]
last_element = my_list[-1]
```

- **Modifying Elements**:
  - Update a value by assigning a new one at a specific index.

```
my_list[1] = 10  # Changes second element to 10
```

- **Adding Elements**:
  - Use `append()` to add to the end, `insert()` to add at a specific index, or `extend()` to concatenate lists.

```
my_list.append(6)
my_list.insert(2, 7)  # Adds 7 at index 2
my_list.extend([8, 9])
```

- **Removing Elements**:
  - Use `remove()` to delete a specific value, `pop()` to remove by index, or `clear()` to empty the list.

```
my_list.remove(10)  # Removes the first occurrence of 10
popped_value = my_list.pop(3)  # Removes element at index 3
my_list.clear()  # Removes all elements
```

- **List Length**:
  - Use `len()` to get the number of elements.

```
length = len(my_list)
```

Project 1: To-Do List Manager (Mini-Project, 30 minutes)

Goal:

Create a simple to-do list manager where users can add, view, and remove tasks.

Steps:

1. Create an empty list to store tasks.
2. Allow the user to choose between adding, viewing, or removing tasks.
3. Implement functionalities to modify the to-do list based on user input.

Example Code:

```
to_do_list = []

def add_task(task):
```

```
    to_do_list.append(task)
    print(f"Task '{task}' added.")

def view_tasks():
    if to_do_list:
        print("Your tasks:")
        for i, task in enumerate(to_do_list, 1):
            print(f"{i}. {task}")
    else:
        print("No tasks in the list.")

def remove_task(task_number):
    if 0 < task_number <= len(to_do_list):
        removed = to_do_list.pop(task_number - 1)
        print(f"Task '{removed}' removed.")
    else:
        print("Invalid task number.")

while True:
    print("\n1. Add Task")
    print("2. View Tasks")
    print("3. Remove Task")
    print("4. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        task = input("Enter the task: ")
        add_task(task)
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        view_tasks()
        task_number = int(input("Enter the task number to remove: "))
        remove_task(task_number)
    elif choice == "4":
        break
    else:
        print("Invalid choice. Please try again.")
```

Challenge:

- Add an option to **mark tasks as completed** and show which tasks are done and which are pending.

1.4 Advanced List Operations: Slicing, Sorting, and Comprehensions (45 minutes)

List Slicing:

- **Syntax**:

```
sliced_list = my_list[start:stop:step]
```

- **Examples**:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
print(numbers[2:6])  # Outputs: [3, 4, 5, 6]
print(numbers[:4])   # Outputs: [1, 2, 3, 4]
print(numbers[::2])  # Outputs: [1, 3, 5, 7]
```

Sorting Lists:

- Use `sort()` to sort a list in-place or `sorted()` to return a sorted list.
- **Syntax**:

```
my_list.sort()  # Ascending order
my_list.sort(reverse=True)  # Descending order
```

List Comprehensions:

- A concise way to create lists based on existing lists.
- **Syntax**:

```
new_list = [expression for item in iterable]
```

Example:

```
squares = [x**2 for x in range(1, 11)]
```

Project 2: Number Filter (Mini-Project, 20 minutes)

Goal:

Create a program that filters out even numbers from a list and returns a new list with only odd numbers using list comprehension.

Steps:

1. Create a list of numbers from 1 to 50.
2. Use list comprehension to filter out even numbers.
3. Print the new list.

Example Code:

```
numbers = list(range(1, 51))
odd_numbers = [num for num in numbers if num % 2 != 0]
print(odd_numbers)
```

Challenge:

- Modify the program to filter out numbers that are multiples of 3.

1.5 Tuples (30 minutes)

What is a Tuple?:

- A tuple is an immutable, ordered collection of elements. Once a tuple is created, it cannot be modified.
- **Syntax**:

```
my_tuple = (1, 2, 3)
```

Common Tuple Operations:

- **Accessing Elements**:
  - Use indexing to access tuple elements.

```
first_element = my_tuple[0]
```

- **Unpacking Tuples**:
  - Assign each element of a tuple to a variable.

```
a, b, c = my_tuple
```

- **Tuples in Functions**:
  - Use tuples to return multiple values from a function.

```
def get_min_max(numbers):
    return min(numbers), max(numbers)

minimum, maximum = get_min_max([5, 10, 3, 8])
```

Practice Exercise:

1. Create a function that takes a list of numbers and returns a tuple containing the smallest and largest numbers.

1.6 Sets (45 minutes)

What is a Set?:

- A set is an unordered collection of unique items. It does not allow duplicates and is useful for membership tests and set operations.
- **Syntax**:

```
my_set = {1, 2, 3, 4}
```

Set Operations:

- **Adding Elements**:

```
my_set.add(5)
```

- **Removing Elements**:

```
my_set.remove(3)
```

- **Set Operations**: Union, intersection, and difference.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)  # {1, 2, 3, 4, 5}
intersection_set = set1.intersection(set2)  # {3}
difference_set = set1.difference(set2)  # {1, 2}
```

Project 3: Set Operations (Mini-Project, 25 minutes)

Goal:

Write a program that performs set operations to find common and unique elements between two sets of numbers.

Steps:

1. Create two sets with some overlapping elements.
2. Find the union, intersection, and difference.
3. Print the results.

Example Code:

```
set1 = {1,

2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

union_set = set1.union(set2)
intersection_set = set1.intersection(set2)
difference_set = set1.difference(set2)

print(f"Union: {union_set}")
print(f"Intersection: {intersection_set}")
print(f"Difference: {difference_set}")
```

Challenge:

- Allow the user to input numbers to create their own sets.

1.7 Dictionaries (60 minutes)

What is a Dictionary?:

- A dictionary is a collection of key-value pairs where each key maps to a value.
- **Syntax**:

```
my_dict = {"name": "Alice", "age": 25}
```

Dictionary Operations:

- **Adding/Modifying**:

```
my_dict["city"] = "New York"
```

- **Removing**:

```
del my_dict["age"]
```

- **Iteration**:

```
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Project 4: Student Grades Tracker (Mini-Project, 35 minutes)

Goal:

Create a program that tracks students' grades. It should allow adding, updating, and viewing grades.

Steps:

1. Use a dictionary to store student names as keys and grades as values.
2. Implement functionalities to add, update, view, and remove grades.

1.8 File Handling (60 minutes)

Reading and Writing Text Files:

- Use `open()`, `read()`, `write()`, `with`.

Project 5: Persistent To-Do List (Mini-Project, 45 minutes)

Goal:

Modify the to-do list to save tasks in a file and load them on startup.

Day 3 Summary

This detailed day plan covers all data structures, file handling, and integrates **7 projects** to ensure comprehensive learning.