Introduction to Programming and Problem Solving

Week 12: Tuples, Sets and Dictionaries

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

Content

- Introduction to Tuples
- Tuples Indexing and Slicing
- Dictionary
- The Methods of Dictionary Class
- Traversing Dictionaries
- Set
- Set operations etc

Introduction to tuples

- **Tuples** in Python are ordered, immutable collections of elements.
- They are similar to lists, but with one crucial difference: once you create a tuple, you cannot change its content.
- This immutability makes tuples suitable for situations where you want to store a collection of elements that should not be modified.
- In this discussion, we'll cover the characteristics of tuples and provide project code examples to illustrate their practical use.

Introduction to tuples+

Key Characteristics of Tuples

- **Ordered**: Like lists, tuples maintain the order of elements as they are defined.
- **Immutable**: Once you create a tuple, you cannot change, add, or remove elements.
- **Heterogeneous**: Tuples can contain elements of different data types, such as numbers, strings, or even other tuples.

Creating a Tuple:

- Tuples are created using parentheses **()** or with a comma-separated list of values.

```
1.  ##Creating a tuple using parentheses
2. my_tuple = (1, 2, 3)
3. # Creating a tuple using a comma-separated list
4. colors = 'red', 'green', 'blue'
```

Introduction to tuples++

```
• my_tuple1 = (1, 2, 3, 4, 5)
• my_tuple2 = ("apple", "orange", "banana")
• my_tuple3 = ()  # Empty tuple
• my_tuple4 = (4,)  # Single-element tuple
```

Accessing Elements:

- You can access tuple elements by their index, just like with lists.
- print(my_tuple[0]) # Accessing the first element (index 0)

Common uses of tuples:

- Tuples are immutable, which makes them useful for creating data structures that should not be changed.
- Tuples are often used to store related data, such as coordinates or dates.
- Tuples can be used to return multiple values from a function.

Inbuilt Functions for Tuples

- Tuples in Python are similar to lists but with one key difference: they are immutable, meaning their elements cannot be modified once defined. Python provides a variety of built-in functions and methods for working with tuples. Here are some common inbuilt functions for tuples:

- **len()**: Returns the number of elements in a tuple.

- my_tuple = (1, 2, 3, 4, 5) length = len(my_tuple) # length will be 5

- **max()**:Returns the maximum value from a tuple (only applicable if all elements are of the same type and can be compared).

- my_tuple = (10, 20, 5, 30)

- max_value = max(my_tuple) # max_value will be 30

Inbuilt Functions for Tuples+

- **min()**: Returns the minimum value from a tuple (similar to **max()**).
- my_tuple = (10, 20, 5, 30)
- min_value = min(my_tuple) # min_value will be 5
- **sum()**: Returns the sum of all elements in a tuple (only applicable if all elements are numeric).
- my_tuple = (1, 2, 3, 4, 5) total = sum(my_tuple) # total will be 15
- **sorted()**: Returns a sorted list of the elements in a tuple. The original tuple remains unchanged.
- my_tuple = (5, 2, 8, 1, 3)
- sorted_tuple = sorted(my_tuple)
- # sorted_tuple will be [1, 2, 3, 5, 8]

Inbuilt Functions for Tuples++

- **tuple()**: Converts a sequence (e.g., a list or a string) into a tuple.
- my_list = [1, 2, 3, 4, 5]
- my_tuple = tuple(my_list) # my_tuple will be (1, 2, 3, 4, 5)
- **count()**: Returns the number of occurrences of a specified element in a tuple.
- my_tuple = (1, 2, 2, 3, 4, 2)
- count = my_tuple.count(2)
- # count will be 3
- **index()**: Returns the index of the first occurrence of a specified element in a tuple.
- my_tuple = (10, 20, 30, 40, 50)
- index = my_tuple.index(30) # index will be 2

Passing Variable Length Arguments to Tuples

- You can pass variable-length arguments to a function using ***args** and then convert these arguments into a tuple.
- The ***args** syntax allows you to pass an arbitrary number of positional arguments to a function, which will be collected into a tuple.
- How you can use ***args** to pass variable-length arguments to a function and work with them as a tuple:

```
def example_function(*args):
    for arg in args:
        print(arg)
# Call the function with multiple arguments
example_function(1, 2, 3, 4, 5)
```

Passing Variable Length Arguments to Tuples+

➢ You can also explicitly pass a tuple to a function and use the * operator to unpack it into individual arguments.

Example:

```
def example_function(arg1, arg2, arg3):
    print(arg1)
    print(arg2)
    print(arg3) # Create a tuple and unpack it to pass as arguments
my_tuple = (1, 2, 3)
example_function(*my_tuple)
```

Passing Variable Length Arguments to Tuples++

- In this case, the ***my_tuple** syntax is used to unpack the elements of the tuple and pass them as separate arguments to the **example_function**.
- Both of these approaches allow you to work with variable-length arguments as if they were elements in a tuple within your function.
- The choice between them depends on whether you want to accept an arbitrary number of arguments as a tuple or explicitly pass a tuple and then unpack it.

Lists and Tuples

- Lists and Tuples are both used to store collections of items, but they have some key differences, primarily related to their mutability, syntax, and use cases.
- Comparison of lists and tuples:

Lists:

- **Mutable**: Lists are mutable, which means you can change, add, or remove elements after the list is created.
- **Syntax**: Lists are defined using square brackets **[]**.
- **Performance**: Lists can be less memory-efficient and slower for certain operations because of their mutability.

Lists and Tuples++

- **Use Case**: Lists are suitable for collections of items that might need to be modified, such as a list of to-do items, shopping lists, or data that is subject to change.,

Example of creating a list:

- my_list = [1, 2, 3, 4]

- my_list.append(5) # Modifying the list by adding an element

Lists and Tuples+++

Tuples:

- **Immutable**: Tuples are immutable, which means you cannot change their elements after they are created.
- **Syntax**: Tuples are defined using parentheses **()**, although they can be created without parentheses as well.
- **Performance**: Tuples are generally more memory-efficient and faster for certain operations due to their immutability.
- **Use Case**: Tuples are suitable for collections of items that should not be changed, such as coordinates, configuration settings, or function return values.

Tuple indexing and Slicing

- Indexing in tuples is similar to indexing in lists. You can access individual elements of a tuple by their index, which starts from 0.
- For example, to access the first element of a tuple, you would use [0].
- To access the last element of a tuple, you would use [-1].
- You can also use negative indices to access elements from the end of the tuple. For example, [-2] refers to the second-to-last element, and [-3] refers to the third-to-last element.

Tuple indexing and Slicing

- Slicing in tuples allows you to access a range of elements from a tuple.
- To slice a tuple, you use the colon (:). The syntax is [start:stop:step].
- start: The index of the first element to include in the slice.
- stop: The index of the first element to exclude from the slice.
- step: The number of elements to skip between each element in the slice.
- If you omit stop and step, Python will assume that you want to slice the tuple from the start to the end, with a step of 1.
- Here are some examples of slicing in tuples:

Tuple Slicing examples

- `my_tuple = (1, 2, 3, 4, 5)`
  `# Slice the first two elements:`
- `first_two = my_tuple[:2]`
  `# Slice the last two elements:`
- `last_two = my_tuple[-2:]`
  `# Slice the elements from 1 to 4, with a step of 2:`
- `middle_two = my_tuple[1:4:2]`
  `# Slice the entire tuple:`
- `all_elements = my_tuple[:]`

Code examples to demonstrate usage of tuple

- **Example 1: Storing Coordinates**

```
1. point = (3, 4)
2. # Accessing and using the coordinates
3. x, y = point
4. # Unpacking the tuple
5. distance = (x**2 + y**2)**0.5
6. print(f"The distance from the origin to the point {point} is {distance:.2f}
   units.")
```

- In this example, we create a tuple to store the coordinates of a point, access its elements, and calculate the distance from the origin using the Pythagorean theorem

Code examples to demonstrate usage of tuple+

➢ **Example 2: Representing Date**

```
1. #Create a tuple to represent a date
2. date = (2023, 11, 3)
3. # Accessing and displaying the date
4. year, month, day = date
5. print(f"Today's date is {year}-{month:02d}-{day:02d}.")
```

➢ This example uses a tuple to represent a date, allowing you to easily extract and display the year, month, and day components.

Code examples to demonstrate usage of tuple++

- **Example 5: Storing and managing contacts**

- Tuples are commonly used to store and manage structured data in Python.

- A real-world example of using tuples is to create a basic contact list where each contact is represented as a tuple containing information like name, phone number, and email address.

- Here's a simplified example of a contact list project using tuples:

Storing and managing contacts project

```
contacts = []
# Function to add a contact to the list
def add_contact(name, phone, email):
    contact = (name, phone, email)
    contacts.append(contact)
    print(f"Contact '{name}' added successfully.")
# Function to display all contacts
def display_contacts():
    print("\nContact List:")
    for i, contact in enumerate(contacts, start=1):
        name, phone, email = contact
        print(f"{i}. Name: {name}, Phone: {phone}, Email: {email}")
# Main program loop
```

Storing and managing contacts project+

```
while True:
    print("\nContact List Project")
    print("1. Add Contact")
    print("2. Display Contacts")
    print("3. Quit")
    choice = input("Select an option (1/2/3): ")
    if choice == "1":
        name = input("Enter the name: ")
        phone = input("Enter the phone number: ")
        email = input("Enter the email address: ")
        add_contact(name, phone, email)
    elif choice == "2":
        display_contacts()
    elif choice == "3":
        break
    else:
        print("Invalid choice. Please select 1, 2, or 3.")
        print("Contact List Project ended. Goodbye!")
```

Explanation of the above code

- We start with an empty list called contacts to store contact information.
- We define two functions: add_contact and display_contacts.
- The add_contact function takes a name, phone number, and email address, creates a tuple representing a contact, and appends it to the contacts list. It provides feedback on successful addition.
- The display_contacts function iterates through the list of contacts, extracting and displaying the information for each contact.
- The main program loop presents a simple menu for adding contacts, displaying contacts, or quitting the program. The loop continues until the user chooses to quit.
- Input validation is included to handle invalid menu choices.

The zip() Function

- The **zip()** function in Python is a built-in function that is used to combine two or more iterables (e.g., lists, tuples, or strings) into a single iterable.
- It pairs elements from each input iterable in the order they appear and creates an iterator of tuples
- Basic syntax of the zip function: zip(iterable1, iterable2, ...)
- The **zip()** function continues until the shortest input iterable is exhausted. If the input iterables are of different lengths, the resulting iterator will contain tuples up to the length of the shortest iterable

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)

for item in zipped:
    print(item)
```

Output:

(1, 'a')

(2, 'b')

(3, 'c')

The zip() Function++

➢ Creating Lists or Dictionaries from zip():

```
1. keys = ['a', 'b', 'c']
2. values = [1, 2, 3]
3. zipped_dict = dict(zip(keys, values))
4. print(zipped_dict)
5. # Output: {'a': 1, 'b': 2, 'c': 3}
```

➢ You can also use the **zip()** function with more than two input iterables. For instance, if you have three lists, you can combine them like this:

```
• list1 = [1, 2, 3]
• list2 = ['a', 'b', 'c']
• list3 = [10, 20, 30]
• combined_data = zip(list1, list2, list3)
• result = list(combined_data)
• print(result)
• Output:[(1, 'a', 10), (2, 'b', 20), (3, 'c', 30)]
```

The Inverse zip(*) Function

- Python does not provide a direct built-in function for "unzipping" or obtaining the inverse of **zip()**, but you can achieve the same result by using the ***** operator (extended unpacking) and the **zip()** function in combination with list comprehensions or other techniques.

- The idea is to take a sequence of tuples and separate the elements into separate lists or sequences

```
1. data = [(1, 'a'), (2, 'b'), (3, 'c')]
2. # Unzip the data into separate lists
3. first_elements, second_elements = zip(*data)
4. # Convert the tuples to lists (optional)
5. first_list = list(first_elements)
6. second_list = list(second_elements)
7. print(first_list)
8. print(second_list)
```

The Inverse zip(*) Function

- In this example, we first unzip the list of tuples **data** into separate tuples **first_elements** and **second_elements** using **zip(*data)**. Then, if you want to convert them to lists, you can use the **list()** constructor as shown in the example.

- The ***** operator is used to unpack the list of tuples, effectively reversing the process of zipping. You can do this with any number of tuples and elements.

Dictionaries

- A dictionary is a built-in data structure used to store collections of key-value pairs. Each key is associated with a value, creating a mapping that allows you to efficiently access and manipulate data.
- Dictionaries are commonly used for tasks that require fast and organized data retrieval.

Key Characteristics:

- **Unordered**: Dictionaries do not guarantee any specific order for key-value pairs. In other words, the order in which you insert items is not necessarily the order in which you'll retrieve them.

Dictionaries+

- **Unique Keys**: Each key within a dictionary must be unique. Duplicate keys are not allowed.
- **Mutable**: You can add, modify, or remove key-value pairs in a dictionary.
- **Keys are Immutable**: Keys must be of an immutable data type (e.g., strings, numbers, or tuples).
- **Arbitrary Values**: Values associated with keys can be of any data type (e.g., strings, numbers, lists, other dictionaries, or more complex objects).
- In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.[2]

Creating and Accessing Dictionaries

➢ **Creating a Dictionary:** You can create a dictionary by enclosing key-value pairs within curly braces or using the **dict()** constructor.

```
# Empty dictionary
empty_dict = {}
# Dictionary with initial key-value pairs
person = { 'name': 'John', 'age': 30, 'city': 'New York' }
```

Accessing Values:

```
# Accessing values using keys
name = person['name']
# Retrieves the value associated with the 'name' key.
# Using the get() method (useful for avoiding KeyError)
age = person.get('age')
```

Modifying and Adding Key-Value Pairs

```
person = { 'name': 'John', 'age': 30, 'city': 'New York' }
person['age'] = 31
# Adding a new key-value pair
person['job'] = 'Engineer'
```

5. Removing Key-Value Pairs

6. # Removing a key-value pair

```
del person['age’]
```

Iterating Through a Dictionary

```
for key in person.keys():
    print(key)
    # Iterating through values
for value in person.values():
    print(value)
    # Iterating through key-value pairs
for key, value in person.items():
    print(key, value)
```

The Methods of Dictionary Class

➢ Common methods and operations associated with the dictionary class

- **get(key, default=None):** Returns the value for the given key, or the specified default value if the key is not found.

```
my_dict = {'a': 1, 'b': 2}
value = my_dict.get('a')  # Returns 1
```

- **pop(key, default=None):** Removes the key and returns its value, or the specified default value if the key is not found.

```
my_dict = {'a': 1, 'b': 2}
value = my_dict.pop('a')  # Removes 'a' and returns its value: 1
```

The Methods of Dictionary Class+

- **popitem():** Removes and returns a key-value pair as a tuple. The pair is removed in LIFO (Last-In-First-Out) order.

```
my_dict = {'a': 1, 'b': 2}
my_dict.update({'c': 3})  # Adds 'c': 3 to the dictionary
```

- **update(iterable):** Updates the dictionary with key-value pairs from another dictionary or an iterable.

```
my_dict = {'a': 1, 'b': 2}
my_dict.update({'c': 3})  # Adds 'c': 3 to the dictionary
```

- **clear():** Removes all key-value pairs from the dictionary.

```
my_dict = {'a': 1, 'b': 2}
my_dict.clear()  # Empties the dictionary: {}
```

The Methods of Dictionary Class++

- **copy():** Returns a shallow copy of the dictionary.

```
original = {'a': 1, 'b': 2}
new_dict = original.copy()
```

- **keys():** Returns a view object of all keys in the dictionary.

```
my_dict = {'a': 1, 'b': 2}
all_keys = my_dict.keys()
# Output: dict_keys(['a', 'b'])
```

The Methods of Dictionary Class+++

- **values():** Returns a view object of all values in the dictionary.

```
1. my_dict = {'a': 1, 'b': 2}
2. all_values = my_dict.values()
3. # Output: dict_values([1, 2])
```

- **items():** Returns a view object of all key-value pairs in the dictionary.

```
1. my_dict = {'a': 1, 'b': 2}
2. all_items = my_dict.items()
3. # Output: dict_items([('a', 1), ('b', 2)])
```

Real world application of dictionaries

```
1. # Create a simple English-French dictionary
2. english_to_french = { 'apple': 'pomme', 'banana': 'banane', 'grape':
   'raisin' }
3. # Translate a word
4. word = input("Enter an English word: ")
5. if word in english_to_french:
6.     print(f"The French translation of '{word}' is '{english_to-
   french[word]}'.")
7. else: print(f"Translation not found for '{word}'.")
```

Real world application of dictionaries+

- **Example 2: Tracking Inventory**

```
1. inventory = { 'apple': 100, 'banana': 150, 'orange': 75 }
2. # Update inventory after a sale
3. product = input("Enter the product name: ")
4. quantity_sold = int(input("Enter the quantity sold: "))
5. if product in inventory and inventory[product] >= quantity_sold:
6.     inventory[product] -= quantity_sold
7.     print(f'Sale recorded. Remaining {product} inventory:
   {inventory[product]}')
8. else:
9.     print('Product not found or insufficient inventory.')
```

Real world application of dictionaries++

- **Example 3: Storing User Profiles**

```
# Create a dictionary to store user profiles
user_profiles = {} # Collect user information
username = input("Enter your username: ")
age = int(input("Enter your age: "))
city = input("Enter your city: ")
# Create a user profile
user_profiles[username] = {'age': age, 'city': city}
# Display user profile
print(f"User profile for {username}: {user_profiles[username]}")
```

Real world application of dictionaries+++

Example 4: Aggregate and process data using dictionaries.

```
data = [
    {'user': 'Alice', 'points': 100},
    {'user': 'Bob', 'points': 150},
    {'user': 'Alice', 'points': 50}
]
user_points = {}
for entry in data:
    user = entry['user']
    points = entry['points']
    user_points[user] = user_points.get(user, 0) + points
```

Sets in Python

- A Set in Python programming is an unordered collection data type that is iterable, mutable and has no duplicate elements.[1]
- They are useful for tasks where you need to store a collection of items without duplicates or specific order.
- In this lecture, we'll explore the key characteristics of sets, explain their uses, and provide project code examples to illustrate their practical applications.

Key Characteristics:

- **Unordered**: Sets do not maintain any specific order for elements. The order of items is not guaranteed.
- **Unique Elements**: Sets do not allow duplicate elements. Every item in a set is unique.
- **Mutable**: You can add and remove elements from a set.
- A set can contain different data types[3]

Sets in Python +

Creating a Set:

- Sets are created using curly braces **{}** or the **set()** constructor.

```
1. # Creating a set with curly braces
2. my_set = {1, 2, 3}
3. # Creating a set using the set() constructor
4. colors = set(['red', 'green', 'blue'])
```

- **Adding and Removing Elements**:

- You can add elements to a set using the **add()** method and remove elements using the **remove()** method.

```
• my_set.add(4) # Add an element
• my_set.remove(2) # Remove an element
```

Set operations

- Set operations in Python refer to various methods and operators used to perform tasks on sets. These operations help manipulate sets to find intersections, unions, differences, and more

- Union (| or union()): Combines elements from two sets, excluding duplicates.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
# Union of set1 and set2
```

- Intersection (& or intersection()): Finds common elements in two sets

```
intersection_set = set1.intersection(set2)
# Intersection of set1 and set2
```

Set operations+

- Difference (- or difference()): Finds elements in one set but not in the other.
- set1 = {1, 2, 3}
- set2 = {3, 4, 5}
- difference_set = set1 - set2 # {1, 2}
- Symmetric Difference (^ or symmetric_difference()): Finds elements in either set but not in both.
- symmetric_difference_set = set1 ^ set2 # {1, 2, 4, 5}

Set operations++

- **Set Comprehension**: Sets can be created using set comprehensions, which are similar to list comprehensions.
- my_set = {x for x in range(1, 6)}
- **Membership Testing**: You can check if an element is present in a set using the **in** operator.
- my_set = {1, 2, 3, 4, 5}
- is_present = 3 in my_set # True

Set operations+++

- **Length and Clearing**: You can get the length of a set using the **len()** function. To clear all elements from a set, you can use the **clear()** method.
- my_set = {1, 2, 3} length = len(my_set) # 3
- my_set.clear() # Removes all elements, my_set is now an empty set
- **Frozen Sets**: Python also provides a **frozenset** class, which is an immutable version of a set. Once created, the elements cannot be modified.
- frozen_set = frozenset([1, 2, 3])

Set Project examples

➢ **Example 1: Removing Duplicates from a List**

```
# Create a list with duplicate elements
fruits = ['apple', 'banana', 'apple', 'orange', 'banana']
# Convert the list to a set to remove duplicates
unique_fruits = set(fruits)
# Convert the set back to a list if needed
unique_fruits_list = list(unique_fruits)
print("Original list:", fruits)
print("List with duplicates removed:", unique_fruits_list)
```

Set Project examples+

- **Example 3: Implementing a Simple Voting System**

```
1. # Create an empty set to store votes
2. votes = set()
3. # Simulate voters casting their votes
4. voters = ['Alice', 'Bob', 'Charlie', 'Alice', 'David', 'Alice']
5. for voter in voters:
6.     if voter not in votes:
7.         print(f"{voter} cast their vote.")
8.         votes.add(voter)
9.     else:
10.         print(f"{voter} has already voted. Their vote is not counted.")
11.         print("Votes received:", votes)
```

Set Project examples++

1. Example 4: Shopping cart

```
# Create an empty shopping list set
shopping_list = set()
# Function to add items to the shopping list
def add_to_shopping_list(item):
    if item in shopping_list:
        print(f"'{item}' is already in the shopping list.")
    else:
        shopping_list.add(item)
        print(
            f"'{item}' added to the shopping list."
        )  # Function to display the shopping list
def display_shopping_list():
    if not shopping_list:
        print("The shopping list is empty.")
    else:
        print("\nShopping List:")
```

Set Project examples+++

```
18)     for item in shopping_list:
19)         print(f"- {item}")
20)         # Main program loop
21)         while True:
22)             print("\nShopping List Project")
23)             print("1. Add Item to Shopping List")
24)             print("2. Display Shopping List")
25)             print("3. Quit")
26)             choice = input("Select an option (1/2/3): ")
27)             if choice == "1":
28)                 item = input("Enter the item to add to the shopping list: ")
29)                 add_to_shopping_list(item)
30)             elif choice == "2":
31)                 display_shopping_list()
32)             elif choice == "3":
33)                 break
34)             else:
35)                 print("Invalid choice. Please select 1, 2, or 3.")
36)                 print("Shopping List Project ended. Happy shopping!")
```

Explanation of our shopping cart coordaninator

- We start with an empty set called shopping_list to store unique items.
- We define two functions: add_to_shopping_list and display_shopping_list.
- The add_to_shopping_list function takes an item, checks if it's already in the set, and adds it to the shopping list if it's not already there. It provides feedback on the addition.
- The display_shopping_list function checks if the shopping list is empty and, if not, iterates through the set to display the items.
- The main program loop presents a simple menu for adding items, displaying the shopping list, or quitting the program. The loop continues until the user chooses to quit.
- Input validation is included to handle invalid menu choices.

Summary

- In wrapping up our week 12 Lecture, we've delved into the intriguing world of data structures and their functionalities. This week has been a deep dive into three essential components: sets, tuples, and dictionaries, each offering unique capabilities in managing and manipulating data.

Sets:

- Sets have empowered us with the ability to handle collections of unique elements, allowing for efficient operations such as finding intersections, unions, differences, and symmetric differences. Their characteristic feature of exclusivity among elements has proven invaluable in various data-handling scenarios, enhancing our ability to work with distinct entities and streamline processes.

Summary+

Tuples:

- Tuples, with their immutability and ordered sequence, have shown us the power of structured data representation. Their unchangeable nature makes them reliable containers for preserving data integrity, especially in scenarios where preserving the sequence and contents is critical, like in coordinates, configuration settings, or representing fixed attributes of an entity.

Dictionaries:

- Diving into dictionaries has been an exploration of key-value pairs, offering a versatile approach to managing data through associative arrays. The capability to access values via keys has unlocked a world of efficiency in handling diverse data types and structures. With its wide range of methods and functionalities, dictionaries have proven instrumental in applications ranging from data storage to web development.

Summary++

- Throughout this week's lecture, we've not only explored the fundamental aspects of these data structures but also understood their practical implications in real-world applications. These tools have expanded our capabilities in managing and organizing data, offering versatile solutions for an array of programming challenges.
- As we conclude this week's lecture, the understanding gained in sets, tuples, and dictionaries provides a robust foundation for navigating the diverse landscape of data structures and lays the groundwork for more sophisticated data handling in our ongoing journey in programming and problem-solving.

Reference

[1] (2023, May 18). *Sets in Python*. GeeksforGeeks. Retrieved November 9, 2023, from https://www.geeksforgeeks.org/sets-in-python/

[2] (n.d.). *Python Dictionary - javatpoint*. Www.Javatpoint.com. Retrieved November 11, 2023, from https://www.javatpoint.com/python-dictionary

[3] (n.d.). *Python sets*. W3schools. Retrieved November 11, 2023, from https://www.w3schools.com/python/python_sets.asp