Artificial Intelligence Diploma

Python Session 2

NeuroTech

Agenda :

1. List
2. Tuple
3. Set
4. Dictionary
5. Comparison

List

List :

- In Python, a list is a built-in data structure that represents an ordered collection of elements. Lists are used to store multiple items in a single variable, and each item in the list is called an element. Lists are versatile and can hold elements of different data types, including integers, floats, strings, and even other lists.

- Here's how you can create a list in Python:

```
# Creating an empty list
empty_list = []

# Creating a list with elements
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'orange']
mixed_list = [1, 'hello', True, 3.14]
```

List Access Elements :

- In Python, you can access elements in a list using indexing. Lists are ordered collections of elements, and each element is associated with an index. The index starts from 0 for the first element and increments by 1 for each subsequent element.

- Here's how you can access elements in a list:

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

print(numbers[0])   # Output: 1
print(numbers[2])   # Output: 3
```

- You can also use negative indices to access elements from the end of the list:

```
numbers = [10, 20, 30, 40, 50]

print(numbers[-1])  # Output: 50 (last element)
print(numbers[-3])  # Output: 30 (third element from the end)
```

List Access Elements :

- If you try to access an index that is out of the range of the list, Python will raise an '**IndexError' .**

```
numbers = [10, 20, 30, 40, 50]

# Trying to access an index that is out of range
print(numbers[5])   # Raises IndexError: list index out of range
```

Slicing

- List slicing in Python allows you to extract a portion of a list, creating a new list with the specified elements. It uses a colon ‘ : ’ to specify the range of elements to include in the new list. The general syntax for list slicing is:

- new_list = original_list[start_index:end_index:step]

  •**start_index**: The index of the first element to include in the new list (inclusive).
  •**end_index**: The index of the first element to exclude from the new list (exclusive). The slicing stops just before this index.
  •**step**: Optional argument that specifies the step size. It determines how many elements to skip while slicing. The default step value is **1**.

Slicing

- Here are some examples of list slicing:

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Slicing from index 2 to index 5 (exclusive)
subset1 = numbers[2:5]
print(subset1)  # Output: [2, 3, 4]

# Slicing from the beginning to index 6 (exclusive)
subset2 = numbers[:6]
print(subset2)  # Output: [0, 1, 2, 3, 4, 5]

# Slicing from index 5 to the end
subset3 = numbers[5:]
print(subset3)  # Output: [5, 6, 7, 8, 9]

# Slicing with a step of 2 (includes every second element)
subset4 = numbers[1:9:2]
print(subset4)  # Output: [1, 3, 5, 7]

# Reverse the list using slicing (step -1)
reversed_list = numbers[::-1]
print(reversed_list)  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Slicing

- Keep in mind that list slicing returns a new list, so modifying the sliced list will not affect the original list.
- You can also use negative indices for slicing, which allows you to slice from the end of the list:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Slicing from the second-to-last element to the fourth-to-last element
subset5 = numbers[-2:-4:-1]
print(subset5)  # Output: [8, 7]
```

- List slicing is a powerful feature that enables you to work with a subset of elements in a list and perform various operations efficiently.

List Functions

- In Python, lists are a powerful data structure that offers various built-in functions and methods to manipulate and analyze data stored in lists. Here are some of the most commonly used list functions:

1) **len()**: Returns the number of elements in a list.

```
numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # Output: 5
```

List Functions

2) **max()**: Returns the largest element in a list.

```
numbers = [10, 5, 25, 15]
print(max(numbers))  # Output: 25
```

3) **min()**: Returns the smallest element in a list.

```
numbers = [10, 5, 25, 15]
print(min(numbers))  # Output: 5
```

List Functions

4) **sum()**: Returns the sum of all elements in a list.

```
numbers = [1, 2, 3, 4, 5]
print(sum(numbers))  # Output: 15
```

5) **sorted()**: Returns a new sorted list without modifying the original list.

```
numbers = [5, 2, 8, 1, 3]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 2, 3, 5, 8]
print(numbers)         # Output: [5, 2, 8, 1, 3]
```

List Functions

Python come with a variety of built-in methods that allow for easy manipulation and modification of the list's elements. Let's take a closer look at some of the commonly used list methods:

1) **append()**: Adds an element to the end of the list.

```
fruits = ['apple', 'banana']
fruits.append('orange')
print(fruits)  # Output: ['apple', 'banana', 'orange']
```

List Functions

2) **extend()**: Extends the list by appending elements from another iterable (e.g., another list, tuple, or string).

```
fruits = ['apple', 'banana']
more_fruits = ['orange', 'grape']
fruits.extend(more_fruits)
print(fruits)  # Output: ['apple', 'banana', 'orange', 'grape']
```

3) **insert()**: Inserts an element at a specific index.

```
numbers = [1, 2, 3, 5]
numbers.insert(3, 4)
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

List Functions

4) **remove()**: Removes the first occurrence of a specified element from the list.

```
fruits = ['apple', 'banana', 'orange']
fruits.remove('banana')
print(fruits)  # Output: ['apple', 'orange']
```

5) **pop()**: Removes and returns the element at the specified index. If no index is provided, it removes the last element.

```
fruits = ['apple', 'banana', 'orange']
popped_fruit = fruits.pop(1)
print(popped_fruit)  # Output: 'banana'
print(fruits)        # Output: ['apple', 'orange']
```

List Functions

6) **count()**: Returns the number of occurrences of a specific element in the list.

```
numbers = [1, 2, 2, 3, 2, 4, 2, 5]
count_2 = numbers.count(2)
print(count_2)  # Output: 4
```

7) **index()**: Returns the index of the first occurrence of a specific element in the list.

```
fruits = ['apple', 'banana', 'orange']
index_orange = fruits.index('orange')
print(index_orange)  # Output: 2
```

List Copy

- There are two primary methods to create a copy of a list: "shallow copy" and "deep copy." The difference lies in how they handle nested objects (e.g., lists within lists or other mutable objects). Let's explore both methods:

1. **Shallow Copy:**
   A shallow copy creates a new list, but it only copies the references to the elements of the original list. If the original list contains nested objects (e.g., other lists), the references to those objects will be copied to the new list, but the nested objects themselves will still be shared between the original and copied lists. Any changes made to the nested objects will affect both the original and copied lists.

- To create a shallow copy, you can use the slicing technique or the **copy()** method.

List Copy

Using slicing:

```
original_list = [1, 2, [3, 4]]
shallow_copy = original_list[:]
```

Using The **‘copy’** method :

```
import copy

original_list = [1, 2, [3, 4]]
shallow_copy = copy.copy(original_list)
```

List Copy

2. Deep copy

A deep copy, on the other hand, creates a completely independent copy of the original list, including all the nested objects. It creates new copies of all the nested objects within the original list, ensuring that the copied list is entirely separate from the original list.
To create a deep copy, you can use the **deepcopy()** function from the **copy** module.

```
import copy

original_list = [1, 2, [3, 4]]
deep_copy = copy.deepcopy(original_list)
```

Tuple

Tuple :

- A tuple is an ordered, immutable collection of elements. Tuples are very similar to lists, but the main difference is that tuples cannot be changed after creation, making them immutable data structures. Once you create a tuple, you cannot add, remove, or modify elements in it. Tuples are defined using parentheses **()**.

- Here's how you can create a tuple in Python:

```
# Creating an empty tuple
empty_tuple = ()

# Creating a tuple with elements
fruits = ('apple', 'banana', 'orange')

# Tuple with different data types
mixed_tuple = (1, 'hello', True, 3.14)
```

Tuple :

- Even though tuples are immutable, you can perform various operations on them, such as indexing, slicing, and iterating:

```
fruits = ('apple', 'banana', 'orange')

# Accessing individual elements using indexing
print(fruits[0])  # Output: 'apple'
print(fruits[1])  # Output: 'banana'

# Slicing a tuple
print(fruits[1:])  # Output: ('banana', 'orange')

# Iterating over a tuple
for fruit in fruits:
    print(fruit)
# Output:
# 'apple'
# 'banana'
# 'orange'
```

Tuple :

- Tuples are particularly useful when you want to store a fixed collection of items that should not be modified. They are commonly used in scenarios where the data needs to remain constant, like representing coordinates, dates, or records with fixed fields.

- It's essential to note that tuples are immutable, so you cannot modify their elements or their size after creation. If you need a collection that can be modified, use a list instead. However, the immutability of tuples also makes them suitable for certain use cases where you want to ensure that the data remains constant and unchangeable throughout the program's execution.

Set

Set :

- A set is an unordered collection of unique elements. It is defined by enclosing the elements within curly braces **{}** or using the built-in **set()** function. Sets are similar to lists and tuples, but they do not allow duplicate values. When you create a set, it automatically removes any duplicate elements, ensuring that each element is unique.

- Here's how you can create a set in Python:

```
# Creating an empty set
empty_set = set()

# Creating a set with elements
fruits = {'apple', 'banana', 'orange'}

# Using set() function
colors = set(['red', 'green', 'blue'])
```

Set :

- sets are unordered, which means they do not maintain the order of elements as they are inserted. Therefore, you cannot access elements in a set using indexing. However, you can iterate through the set to access its elements:

```
fruits = {'apple', 'banana', 'orange'}

for fruit in fruits:
    print(fruit)
# Output: 'orange', 'apple', 'banana' (order may vary)
```

Set :

Sets are commonly used for various purposes, including:

1.Removing Duplicates: Since sets automatically remove duplicates, they are handy when you have a list with duplicate elements, and you want to get a unique collection of items.

2.Membership Testing: You can quickly check if an element exists in a set using the **in** keyword, which makes set membership testing very efficient.

```
fruits = {'apple', 'banana', 'orange'}

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

Set Operation:

Sets in Python support various set operations that allow you to perform common mathematical operations on sets. Here are some of the commonly used set operations:

1) **Union (|)**: Combines two sets and returns a new set containing all the unique elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

Set Operation:

2) Intersection (&): Returns a new set containing only the elements that are present in both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

intersection_set = set1 & set2
print(intersection_set)  # Output: {3}
```

3) Difference (-): Returns a new set containing the elements that are present in the first set but not in the second set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}
```

Set Operation:

4) Symmetric Difference (^): Returns a new set containing the elements that are present in either of the sets, but not in both.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

symmetric_difference_set = set1 ^ set2
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

5) Subset (<=): Checks if one set is a subset of another set. It returns **True** if all elements of the first set are present in the second set.

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}

print(set1 <= set2)  # Output: True
```

Set Operation:

6) Proper Subset (<): Checks if one set is a proper subset of another set. It returns **True** if all elements of the first set are present in the second set, and the second set has additional elements.

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}

print(set1 < set2)  # Output: True
```

7) Superset (>=): Checks if one set is a superset of another set. It returns **True** if all elements of the second set are present in the first set.

```
set1 = {1, 2, 3, 4}
set2 = {1, 2}

print(set1 >= set2)  # Output: True
```

Dictionary

Dictionary :

- A dictionary is a built-in data structure that represents an unordered collection of key-value pairs. Dictionaries are also known as "maps," "hashmaps," or "associative arrays" in other programming languages. They are designed to efficiently store and retrieve data using a unique key associated with each value.

- The key in a dictionary must be immutable, such as a string, number, or tuple, as it is used to hash and uniquely identify the corresponding value. The values in a dictionary can be of any data type, including strings, numbers, lists, tuples, other dictionaries, and more.

- Dictionaries are defined using curly braces **{}** and consist of key-value pairs separated by colons **:**. Each key-value pair represents a mapping where the key is unique, and it is used to access its corresponding value efficiently.

Dictionary :

- Here's the general syntax for creating a dictionary in Python:

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with key-value pairs
dictionary = {
    'key1': value1,
    'key2': value2,
    'key3': value3,
    ...
}
```

Dictionary :

- To access values in a dictionary, you can use the keys:

```
dictionary = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A',
    'is_enrolled': True
}

print(dictionary['name'])         # Output: 'John Doe'
print(dictionary['age'])          # Output: 25
print(dictionary['is_enrolled'])  # Output: True
```

Dictionary :

- Dictionaries are mutable, which means you can add, modify, or remove key-value pairs after creation:

```
dictionary = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

# Adding a new key-value pair
dictionary['school'] = 'ABC High School'

# Modifying a value
dictionary['grade'] = 'B'

# Removing a key-value pair
del dictionary['age']

print(dictionary)
# Output: {'name': 'John Doe', 'grade': 'B', 'school': 'ABC High School'}
```

Dictionary Functions:

- Dictionaries come with a set of built-in functions and methods that allow you to perform various operations and manipulations on dictionaries. Here are some of the commonly used dictionary functions:

1) **len()**: Returns the number of key-value pairs in the dictionary.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

print(len(student))  # Output: 3
```

2) **dict()**: Creates a new dictionary from an iterable of key-value pairs or from a mapping object.

```
# Creating a dictionary from a list of tuples
data = [('name', 'John Doe'), ('age', 25), ('grade', 'A')]
student = dict(data)

print(student)  # Output: {'name': 'John Doe', 'age': 25, 'grade': 'A'}
```

Dictionary Functions:

3) **sorted()**: Returns a new sorted list of keys in the dictionary.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

sorted_keys = sorted(student)
print(sorted_keys)  # Output: ['age', 'grade', 'name']
```

4) **keys()**: Returns a view object that contains the keys of the dictionary.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

keys_view = student.keys()
print(keys_view)  # Output: dict_keys(['name', 'age', 'grade'])
```

Dictionary Functions:

5) **values()**: Returns a view object that contains the values of the dictionary.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

values_view = student.values()
print(values_view)  # Output: dict_values(['John Doe', 25, 'A'])
```

6) **items()**: Returns a view object that contains the key-value pairs of the dictionary as tuples.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

items_view = student.items()
print(items_view)  # Output: dict_items([('name', 'John Doe'), ('age', 25),
```

Dictionary Functions:

7) **get()**: Retrieves the value associated with the given key. If the key is not found, it returns a specified default value (or **None** if not provided).

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

print(student.get('name'))         # Output: 'John Doe'
print(student.get('address'))      # Output: None
print(student.get('address', 'N/A'))  # Output: 'N/A'
```

8) **pop()**: Removes and returns the value associated with the given key. If the key is not found, it raises a **KeyError** (unless a default value is provided).

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

name = student.pop('name')
print(name)        # Output: 'John Doe'
print(student)     # Output: {'age': 25, 'grade': 'A'}
```

Dictionary Functions:

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

```
student = {
    'name': 'John Doe',
    'age': 25
}

student_copy = student.copy()
print(student_copy)    # Output: {'name': 'John Doe', 'age': 25}
```

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

```
student = {
    'name': 'John Doe',
    'age': 25
}

# Update with a dictionary
student.update({'grade': 'A'})

# Update with an iterable of key-value pairs
student.update([('address', '123 Main St'), ('city', 'New York')])
```

Dictionary Functions:

11) **clear()**: Removes all key-value pairs from the dictionary, making it empty.

```
student = {
    'name': 'John Doe',
    'age': 25,
    'grade': 'A'
}

student.clear()
print(student)    # Output: {}
```

- Dictionaries are widely used in Python due to their efficiency and flexibility in mapping data with meaningful keys. They are particularly useful when you need to store and retrieve data based on specific identifiers or labels, making them an essential data structure in Python programming.

Comparison

Comparison

- Comparing lists, sets, tuples, and dictionaries in Python involves understanding their key characteristics, use cases, and differences. Let's compare them based on various aspects:

1. Mutability:

- **Lists:** Mutable (can be changed after creation).
- **Sets:** Mutable (can be changed after creation).
- Tuples: Immutable (cannot be changed after creation).
- **Dictionaries:** Mutable (can be changed after creation).

2. Syntax:

- **Lists:** Enclosed in square brackets **[ ]**.
- **Sets:** Enclosed in curly braces **{ }**.
- **Tuples:** Enclosed in parentheses **( )**.
- **Dictionaries:** Key-value pairs enclosed in curly braces **{ }**.

Comparison

3. **Ordering**:
   - **Lists:** Ordered (maintain the order of elements as they are inserted).
   - **Sets:** Unordered (do not maintain the order of elements).
   - **Tuples:** Ordered (maintain the order of elements as they are inserted).
   - **Dictionaries:** Unordered (do not maintain the order of key-value pairs).

4. **Duplicates**:
   - **Lists:** Allow duplicates.
   - **Sets:** Do not allow duplicates (automatically remove duplicates).
   - **Tuples:** Allow duplicates.
   - **Dictionaries:** Keys must be unique; values can be duplicated.

Comparison

5. Indexing and Slicing:

- **Lists:** Support indexing and slicing.
- **Sets:** Do not support indexing or slicing, as they are unordered.
- **Tuples:** Support indexing and slicing.
- **Dictionaries:** Use keys to access values; they do not support slicing.

6. Performance:

- **Lists:** Generally perform well for most operations, but their performance might degrade with large datasets due to linear search for some operations (e.g., checking membership).
- **Sets:** Perform very well for membership tests and set operations like union and intersection.
- **Tuples:** Offer better performance compared to lists, especially for data that should not change.
- **Dictionaries:** Provide fast lookups using keys.

Comparison

7. Use Cases:

•**Lists:** Suitable for ordered collections of items, especially when the items might need to be modified (add, remove, or change).
•**Sets:** Ideal for collections that require unique elements or need to perform set operations like union, intersection, and difference.
•**Tuples:** Useful when you want to create a collection of elements that should remain constant throughout the program's execution.
•**Dictionaries:** Perfect for mapping key-value pairs, especially when you need to access values based on specific keys.

- In summary, the choice between lists, sets, tuples, or dictionaries depends on the specific requirements of your program. Use lists when you need an ordered, mutable collection of items. Use sets for unique and unordered collections or set operations. Use tuples when you want an ordered, immutable collection, and use dictionaries when you need to map keys to values for efficient lookups and data retrieval.

Any Question ?

NeuroTech

Thanks

NeuroTech