PYTHON

Lecture – 11

Recap

- Control Statements (If-elif-else)
- Loop Statements (while & for)
- String
- List

Contents

- ***Collection/ Sequence* Data Types**: *Built-in data types in Python used to store collections of data*
  - List
  - Tuple
  - Set
  - Dictionary
- **Array**

All Types at a glance

- **List**
  - A list is a ***mutable, ordered, duplicates, heterogeneous*** collection of elements.
  - Defined using **square brackets** []
  - Example: `my_list=[10, 20, 30, 23]`
- **Tuple**
  - A tuple is an immutable, ordered collection of elements.
  - Defined using **parentheses** ()
  - Example: `my_tuple = (1, 2, 3, 4)`
- **SET**
  - A set is a ***mutable***, ***unordered*** collection of ***unique*** elements.
  - Defined using **curly braces {} or the set() function**
  - Example: `my_set={10, 20, 30, 23}`

All Types at a glance

- **Dictionary**
  - A dictionary is a ***mutable***, ***unordered*** collection of kay-value pairs.
  - Defined using **curly braces {} with key-value pairs** separated by a colon : (e.g., {key: value}).
  - Example: `my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}`
- **Array**
  - An array is a mutable, ordered collection of elements, where all elements are of the same data type (unlike a list, which can contain different types).
  - Need to import the array module, and arrays are created using array.array()
  - Example:

```
import array
my_array = array.array('i', [1, 2, 3, 4])
```

Features of Sequence Data types

- **Mutable**: You can modify elements after the list is created (e.g., *add, remove, or change* elements).
- **Ordered**: Items are stored in a specific order (insertion order or preserve order).
- **Duplicates**: You can have duplicate elements.
- **Heterogeneous**: A list can store different data types (e.g., *integers, strings, other lists*).

List, Tuple, Set, Dictionay and Array at a Glance

Feature	List	Tuple	Set	Dictionary	Array (from array module)
Mutable	Yes	No	Yes	Yes	Yes (for array elements)
Ordered	Yes (preserves insertion order)	Yes (preserves order)	No (unordered collection)	Yes (insertion order since Python 3.7)	Yes (preserves order)
Allows Duplicates	Yes	Yes	No	Keys: No, Values: Yes	Yes
Heterogeneous	Yes (can store any data types)	Yes (can store any data types)	Yes (can store any data types)	Keys: No, Values: Yes	No (stores elements of a single data type)

Tuple

- A **tuple** is a collection data type in Python that is used to store multiple items in a single variable.
- ***Faster than List*:** Tuples are generally faster than lists because of their immutability.
- ***Fixed Data:*** If you want to store data that should not change throughout the program, tuples are ideal.
- ***Hashable:*** Tuples can be used as keys in dictionaries because they are hashable, while lists cannot.

Tuple

- Creating Tuples:

```
# Example of a tuple
my_tuple = (1, 2, 3)
print(my_tuple)  # Output: (1, 2, 3)

# Tuple without parentheses
my_tuple = 1, 2, 3
print(my_tuple)  # Output: (1, 2, 3)
```

- Accessing Elements in Tuple:

```
my_tuple = (10, 20, 30, 40, 50)
# Accessing the first element
print(my_tuple[0])  # Output: 10
# Accessing the last element
print(my_tuple[-1])  # Output: 50
```

Tuple (cont..)

- Looping a Tuple

```
my_tuple = ('apple', 'banana', 'cherry')
for fruit in my_tuple:
    print(fruit)
```

- Tuple Slicing

```
my_tuple = (10, 20, 30, 40, 50)
# Slicing from index 1 to 3 (excluding 4)
print(my_tuple[1:4])  # Output: (20, 30, 40)
# Slicing from the beginning to index 3
print(my_tuple[:3])  # Output: (10, 20, 30)
# Slicing from index 2 to the end
print(my_tuple[2:])  # Output: (30, 40, 50)
```

Tuple (cont..)

- **Add, Update Tuple:** To update a tuple first convert it into a list then perfom operations and then revert to tuple.

```
my_tuple = ("apple", "banana", "cherry")
my_list = list(my_tuple)
my_list[1] = "kiwi"
my_tuple = tuple(my_list)
print(my_tuple) #Output: ("apple", "kiwi", "cherry")
```

- **Tuple Unpacking**: Values from a tuple can assign in variable.

```
my_tuple = (10, 20, 30)
#(a, b, c) = my_tuple # parentheses can also be used
a, b, c = my_tuple
print(a)  # Output: 10
print(b)  # Output: 20
print(c)  # Output: 30
```

Tuple (cont..)

- **Join or concat tuples:** (using +)

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3) #Output: ('a', 'b', 'c', 1, 2, 3)
```

- **Multiply Tuple**: (using *)

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
#output: ('apple', 'banana', 'cherry', 'apple', 'banana',
'cherry')
```

Exercises on Tuple

- 11.1 Take few numbers from user input in a tuple and find the summation of those numbers without sum function.
- 11.2 Find the sum of odd number from a tuple.

Exercises on Tuple

- Take few numbers from user input in a tuple and find the summation of those numbers without sum function.

```
# Take numbes and sum using a tuple
numbers = tuple(map(int, input("Enter numbers(Space
separated): ").split()))
sum=0
for num in numbers:
    sum+=num
print(f"Sum: {sum}")
```

SET

- **Sets** in Python are a built-in data structure that represent an unordered collection of unique elements.
- Sets are often used to *remove duplicates* from a list.
- Useful for problems involving sets of data, such as in *data science and algorithms.*
- Fast *membership testing* is useful in various algorithms and data processing. [if num in numbers]

Declearing SET

```
# Declare a set using curly braces
my_set1 = {1, 2, 3, 4, 5}
print(my_set1) #Output: {1, 2, 3, 4, 5}
# Declare a set using the set() function
my_set2 = set([7, 8, 9, 10, 11])
print(my_set2)  #Output: {7, 8, 9, 10, 11}
# Correct way to declare an empty set
empty_set = set()
print(empty_set) #Output: set()
#set don't allow duplicates
my_set3 = {1, 2, 2, 3, 4, 4, 5}
print(my_set3) #Output: {1, 2, 3, 4, 5}
#mixed set - it allows multiple data types
mix_set = {1, "Hello", (2, 3)}
print(mix_set) # Output: {1, (2, 3), 'Hello'}
```

Accessing SETs

- **Sets** are unordered collections, which means they do not support ***indexing, slicing, or direct access*** to individual elements like lists or tuples.
- **A set can be accessed by following ways:**
  - **Iterating over the set** using a loop.
  - **Checking for membership** (i.e., whether an element exists in the set→ in , not in).
  - **Converting the set to a list or tuple** to access elements by index.

- Example:

```
#accessing sets
my_set = {1, 2, 3, 4, 5}
# Looping through a set to access elements
for element in my_set:
    print(element) #output: all elements
#membership test
if 5 in my_set:
    print("5 is in the set") # Output: 5 is in the set
```

User Input SETs

- ```
  #User Input in Set
  user_input = input("Enter elements separated by spaces: ")
  my_set = set(user_input.split())
  print("The set is:", my_set)

  # User input using loop
  my_set = set()
  n = int(input("How many elements do you want to add to the
  set?: "))
  for i in range(n):
      element = input(f"Enter element {i+1}: ")
      my_set.add(element)
  print("The final set is:", my_set)
  ```
  ```

Add SET Items

```
#adding item in sets using add() function
set1 = {"apple", "banana", "cherry"}
set1.add("orange") # orange will be added to the set
print(set1) #Output: {'banana', 'apple', 'orange', 'cherry'}
#adding items from another set using update() function
set2={"pineapple", "mango"}
set1.update(set2)
print(set1) #Output: {'apple', 'mango', 'cherry', 'pineapple',
'banana'}
#update() can add any types--> list, tuple, dictionay
mylist=["papaya","kiwi"]
set1.update(mylist)
print(set1) #Output: {'apple', 'mango', 'cherry', 'pineapple',
'banana','papaya','kiwi'}
```

Remove SET Items

- **remove()**: Removes a *specified item* from the set. If the item is not found, it raises a *key error*. [`myset.remove(item)`]

- **discard()**: Removes a specified item from the set if it is present. If the item is not found, it does nothing and does not raise an error. [`myset.discard(item)`]

- **pop()**:Removes and returns an *arbitrary item* from the set. Raises a KeyError if the set is empty. [`myset.pop()`]

- **clear()**: Removes all items from the set, resulting in an empty set. [`myset.clear()`]

- **del**: The del keyword will delete the set completely. [del `myset`]

Remove SET Items(Examples)

```
thisset = {"apple", "banana", "cherry","orange"}
thisset.remove("banana")
print(thisset) #Output: {'apple', 'cherry','orange'}
#discard()
thisset.discard("apple")
print(thisset) #Output: { 'cherry','orange'}
#pop()
x=thisset.pop()
print(x) #Output: cherry
print(thisset) #output: {'orange'}
#clear()
thisset.clear()
print(thisset) #Output: set()
#del
del thisset
```

Join SET

- ***Union Sets:*** ***C***ombining all unique elements from two or more sets using the *union()* method or the | operator.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
combined_set1 = set1.union(set2)
combined_set2 = set1 | set2
print(combined_set1)  # Output: {1, 2, 3, 4, 5}
print(combined_set2)  # Output: {1, 2, 3, 4, 5}
```

- ***Intersection:*** To find the common elements between two sets, use the intersection() method or the & operator.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
common_elements1 = set1.intersection(set2)
common_elements2 = set1 & set2
print(common_elements1)  # Output: {3}
print(common_elements2)  # Output: {3}
```

Join SET

- ***Difference:*** To get elements in one set but not in another, use the *difference()* method or the - operator.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

- ***Symmetric Difference:*** To get elements that are in either of the sets but not in both, use the *symmetric_difference()* method or the ^ operator. It is reverse of union.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_diff = set1.symmetric_difference(set2)
print(symmetric_diff)  # Output: {1, 2, 4, 5}
```

Exercises on SET

11.3 Write a Python program that takes two sets of numbers as input and prints the common elements. If there are no common elements, print a message indicating that.

11.4 Write a Python program that takes two sets of numbers as input and prints the common elements. If there are no common elements, print a message indicating that. Use a loop and if-else conditions to check for common elements.

11.5 Write a program that takes a set of allowed usernames and checks if the input username is allowed or not. Use a loop to repeatedly ask for input until the user enters a valid username.

11.6 Write a Python program that takes a set of numbers and removes all even numbers from it using a loop and if-else condition.

Exercises on SET

11.3 Write a Python program that takes two sets of numbers as input and prints the common elements. If there are no common elements, print a message indicating that.

```
# Taking input from user for two sets of numbers
set1 = set(map(int, input("Enter numbers for the first set
separated by spaces: ").split()))
set2 = set(map(int, input("Enter numbers for the second set
separated by spaces: ").split()))
# Finding the common elements using function
common_elements = set1.intersection(set2)
# Checking if there are common elements
if common_elements:
    print(f"Common elements: {common_elements}")
else:
    print("No common elements found.")
```

Exercises on SET

11.5 Write a program that takes a set of allowed usernames and checks if the input username is allowed or not. Use a loop to repeatedly ask for input until the user enters a valid username.

```
# list of existing users
users = {"alice", "bob", "charlie", "david", "eve"}
# create an infinite loop until the user found and break if found
user
while True:
    username = input("Enter your username: ").lower()
    if username in users:
        print(f"Welcome, {username}! You are allowed access.")
        break  # Exit if a valid user is found
    else:
        print("Invalid username. Please try again.")
```