(Chapter- 05) Control Structures

If-else statements

It is used for conditional execution of code. It allows to execute certain blocks of code based on whether a condition evaluates to True or False. This is essential for making decisions in your programs.

```
age = 18

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Nested If-Else:

A **nested if-else** is an if-else statement placed inside another if or else, allowing multiple levels of condition checking.

```
id = "hemantraj"
pas = "hemant@123"

if id == input("Enter the user Id: "):
    if pas == input("Enter the password: "):
        print("Hurray! You have successfully logged in.")
    else:
        print("Oops! You entered the wrong password.")
else:
    print("Wrong user id: ")
```

Loops:

It allows a block of code to be executed repeatedly based on a condition. Loops are essential in automating repetitive tasks in programming.

Two primary types of loops in Python are **for loops** and **while loops.**

1. For Loop

It is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or any other iterable object. It executes a block of code for every item in the sequence.

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

Output:

1
2
3
4
5

Range Function:

The **range()** function is often used in for loops to generate a sequence of numbers.

```
for i in range(start, stop, step):
    # code block to execute
```

start: (optional) Starting value (default is 0).

stop: The value at which the loop stops (exclusive).

step: (optional) The increment between numbers (default is 1).

Nested For Loops:

It is a loop inside another loop, where the inner loop runs fully for each iteration of the outer loop and use for more complex iterations.

```
for i in range(3):  # Outer loop
    for j in range(3):  # Inner loop
        print("*", end=" ")
    print()  # Moves to the next line after inner loop finishes
```

Loop Control Statements:

Break: Exits the loop when a certain condition is met.	`for i in range(10):`
	`  if i == 5:`
	`  break`
	`  print(i)`
	Output: 0 1 2 3 4

Continue: Skips the current iteration and moves to the next.	`for i in range(5):`
	`  if i == 3:`
	`  continue`
	`  print(i)`
	Output: 0 1 2 4

Else with For Loop: The else block is executed when the loop finishes normally (i.e., without encountering a break statement).

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

Output:
0
1
2
Loop finished

2. While Loop

A while loop continues to execute a block of code as long as a given condition is True.

Syntax: `while condition:`

condition: The loop will keep running as long as this condition evaluates to True.

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Output:
1
2
3
4
5

Infinite While Loop:

If the condition in a while loop never becomes False, the loop will continue indefinitely, causing an infinite loop.

```
while True:
    print("This is an infinite loop")
```

Differences between For and While Loops

For Loop	While Loop
Used for iterating over a sequence or range.	Used when the number of iterations is unknown, and the loop depends on a condition.
Stops automatically when the sequence ends.	Stops when the condition becomes False.
Ideal for iterating through a known number of elements.	Ideal for conditions where you don't know how many iterations are required.

(Chapter- 06) Data Structures in Python

Data structures are essential for organizing and storing data efficiently, enabling quick access and modifications. Python provides several built-in data structures, which can be classified into:

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

Python Lists: A list is a collection of ordered, mutable elements in Python. Lists can hold any data type (integers, floats, strings, objects, etc.)

```python
my_list = [1, 'apple', 3.14, [1, 2, 3]]
```

List Index: Each item/element in a list has its own unique index. This index can be used to access any particular item from the list. The first item has index [0], second item has index [1], third item has index [2] and so on.

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
#          [0]      [1]     [2]      [3]      [4]
```

- **Positive Indexing:** Start from 0 to positive whole number.

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
#          [0]      [1]     [2]      [3]      [4]
print(colors[2])
print(colors[4])
print(colors[0])
```

Output:

```
Blue
Green
Red
```

- **Negative Indexing:** Similar to positive indexing, negative indexing is also used to access items, but from the end of the list. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on.

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
#          [-5]     [-4]    [-3]     [-2]      [-1]
print(colors[-1])
print(colors[-3])
print(colors[-5])
```

Output:

```
Green
Blue
Red
```

Slicing and Indexing: Slicing allows access to sub lists using the format **list[start: stop: step]**.

```python
lst = [1, 2, 3, 4, 5, 6]
print(lst[1:5])  # Output: [2, 3, 4, 5]
print(lst[::-1])  # Reverse the list: [6, 5, 4, 3, 2, 1]
```

List Operations and Methods:

- **Adding elements:**
  - **append():** Adds a single element.
  - **extend():** Adds multiple elements from another list or iterable.
  - **insert():** Inserts an element at a specific index

```
lst = [1, 2, 3]
lst.append(4)  # [1, 2, 3, 4]
lst.extend([5, 6])  # [1, 2, 3, 4, 5, 6]
lst.insert(1, 'a')  # [1, 'a', 2, 3, 4, 5, 6]
```

- **Removing elements:**
  - **remove():** Removes the first occurrence of the element.
  - **pop():** Removes and returns the element at the given index.
  - **clear():** Removes all elements from the list.

```
lst.remove(3)  # Removes the first '3' in the list.
lst.pop(0)  # Removes the first element and returns it.
lst.clear()  # Empties the list.
```

Introduction to Tuples

A **tuple** is a collection data type in Python, similar to a list. However, unlike lists, tuples are **immutable**, meaning once a tuple is created, its elements cannot be changed, modified, or updated.

Tuples are often used when you need to group multiple related values together and want to ensure that the values remain constant throughout the program.

Characteristics of Tuples:

- **Ordered**: The elements in a tuple maintain their order.
- **Immutable**: Once defined, elements in a tuple cannot be changed.
- **Allows Duplicates**: Tuples can have duplicate elements.
- **Heterogeneous**: A tuple can contain different data types (int, float, string, etc.).

1. Creating Tuples

Tuples are defined by placing a comma-separated sequence of values within parentheses ().

```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
```

Note: To create a tuple with only one element, a comma is required after the element; otherwise, it will not be recognized as a tuple.

2. Accessing Tuple Elements

You can access tuple elements using **indexing**, similar to lists. The index starts at 0 for the first element, 1 for the second, and so on.

Example:

```
my_tuple = ('apple', 'banana', 'cherry')
print(my_tuple[1])  # Output: banana
```

We can perform Slicing and indexing in Tuple also.

3. Tuple Methods

Tuples support only two built-in methods: count() and index().

- **count()**: Returns the number of times a specified value appears in the tuple.

```
my_tuple = (1, 2, 2, 3, 4)
print(my_tuple.count(2))  # Output: 2
```

- **index()**: Returns the index of the first occurrence of the specified value.

```
my_tuple = (1, 2, 3, 4)
print(my_tuple.index(3))  # Output: 2
```

4. Tuple Operations

Just like lists, tuples support a variety of operations, including concatenation, repetition, and membership testing.

- **Concatenation**: Use + to combine two tuples.

```
t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)  # Output: (1, 2, 3, 4)
```

- **Repetition**: Use * to repeat the elements in a tuple.

```
t = (1, 2)
print(t * 3)  # Output: (1, 2, 1, 2, 1, 2)
```

- **Membership Testing**: Use in to check if an element exists in a tuple.

```
my_tuple = (1, 2, 3, 4)
print(3 in my_tuple)  # Output: True
```

5. Tuples vs Lists: Key Differences

Feature	Tuple	List
Mutability	Immutable (Cannot change)	Mutable (Can change)
Syntax	Uses parentheses ()	Uses square brackets []
Methods	Limited (Only count() and index())	Many (append, remove, etc.)
Use Case	Fixed collections	Dynamic collections
Performance	Faster due to immutability	Slower due to mutability

Introduction to Sets:

A **set** is a built-in data structure in Python that represents a collection of unique elements. Sets are unordered, meaning that the items do not have a defined order, and they cannot contain duplicate elements.

They are particularly useful for performing mathematical set operations like union, intersection, and difference.

Characteristics of Sets:

- **Unordered**: The elements do not maintain any specific order.
- **Mutable**: Sets can be modified after creation (you can add or remove elements).
- **No Duplicates**: Sets automatically remove duplicate values.
- **Heterogeneous**: A set can contain elements of different data types (integers, strings, etc.).

---

1. Creating Sets

Sets can be created using curly braces {} or the set() function.

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

Using the set() Function:

```
my_set = set([1, 2, 3, 4])  # Creating a set from a list
```

2. Accessing Elements in a Set

Sets are unordered collections, so **you cannot access elements by index**. However, you can iterate through the elements in a set.

```
my_set = {1, 2, 3}
for item in my_set:
    print(item)
```

3. Adding and Removing Elements

You can add or remove elements from a set using the following methods:

- **Adding Elements**:
    - add(): Adds a single element to a set.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

- **Removing Elements**:
    - remove(): Removes a specified element from a set. Raises a KeyError if the element is not found.

```
my_set.remove(2)
print(my_set)  # Output: {1, 3, 4}
```

    - discard(): Removes a specified element but does not raise an error if the element is not found.

```
my_set.discard(5)  # No error even if 5 is not in the set
```

- **Clearing a Set**:
    - clear(): Removes all elements from the set.

```
my_set.clear()
print(my_set)  # Output: set()
```

4. Set Operations

Sets support several mathematical operations:

- **Union**: Combines two sets to form a new set with all unique elements.
    - Using **|** operator:

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

    - Using the **union()** method:

```
union_set = set1.union(set2)
```

- **Intersection**: Creates a new set with elements that are common to both sets.
    - Using **&** operator:

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

    - Using the **intersection()** method:

```
intersection_set = set1.intersection(set2)
```

- **Difference**: Creates a new set with elements in the first set but not in the second.
    - Using - operator: ans = {1, 2, 3} - {3, 4, 5} /// "ans" will be {1, 2}
    - Using the **difference()** method: **set1.difference(set2)**

- **Symmetric Difference**: Creates a new set with elements in either set but not in both.
    - Using **^** operator:
    - Using the symmetric_difference() method:

---

5. Set Methods

Sets come with a variety of built-in methods for manipulation and analysis:

- **copy()**: Returns a shallow copy of the set.
- **pop()**: Removes and returns an arbitrary element from the set. Raises a Key Error if the set is empty.
- **update()**: Adds multiple elements from another set or iterable.
- **issubset()**: Checks if one set is a subset of another.
- **issuperset()**: Checks if one set is a superset of another.
- **isdisjoint()**: Checks if two sets have no elements in common.

---

6. Set Comprehension

Similar to list comprehensions. **L = set(i for in range(10))**

7. Advantages of Sets

- **Uniqueness**: Automatically removes duplicates, ensuring all elements are unique.
- **Efficiency**: Sets are optimized for membership testing, making operations like checking if an item exists much faster than lists.
- **Convenience**: Useful for mathematical operations such as unions, intersections, and differences.

---

8. When to Use Sets vs Lists

- Use a **set** when you need to maintain a collection of unique items or perform mathematical set operations.
- Use a **list** when you need to maintain the order of items, allow duplicates, or when you need to frequently change the collection.

---

9. Sets vs Other Data Structures

Feature	Set	List	Tuple
Order	Unordered	Ordered	Ordered
Mutability	Mutable	Mutable	Immutable
Duplicates	No	Yes	Yes
Syntax	Curly braces {}	Square brackets []	Parentheses ()
Performance	Fast membership testing	Slower membership testing	Faster than lists