**SINDHU M D (4VZ22CS027)**

**Python:** python is high-level language (human interpreter) and interpreted language (line-by-line execution.

* C /C++ compiled full program execute at once.

**Application:**

* Data science and machine learning.
* Web development
* s/w development

**Why python:**

* easy to learn: syntax in python similar to English language. Simple
* community support: lot of libraries and resources available in python.
* Versatile: used in web development, data science ect..
* Dynamic typed: need not specify the data type in python.
* Object oriented: concept of class and product (object) but c and c++ are procedure oriented i.e functions.

**Variable:** variable in python like container that is used to store the value and we can change value of variable.

**Rules:**

* variable names contain **alpha-numeric and underscore.**
* variable may start with alphabet or underscore not by number.
* Variable in python is case sensitive.

Two types of assignment: ex a=10, b=20

a,b= 10,20

**Datatypes in python:**

**Integer🡪 int**

**Float🡪 float**

**String🡪str**

**None🡪 None**

**Boolean🡪bool**

**type()** used to check the datatype of variable . a=10

print(type(a)) #<typeclass: int>

**commands :**

single line: #this is single line command

multi-line: starts with triple quote and end with triple quote

What is the output type of input () ?

* Input() always return string , even if the user enters a number.

Is python statically or dynamically typed?

* NO need to declare the data type of variable, python figure it out at runtime.

Difference between / and //?

/ 🡪 floating-point division ex: 5/2=2.5

//🡪 integer or floor division ex : 5//2 = 2

**Sting Manipulation**

**String** is sequence of characters. Python provides useful methods to manipulate the string.

String basic operation : concatenation and replication

* Concatenation is joining of two/more strings using + operator.
* Replication is repeating a string for multiple number of times.

**String methods**

**1.upper ():** converts strings to upper case.

**2. lower ()** / **case fold() :** converts strings to lower case.

**3. strip ():** remove leading and trailing space from string.

**4.** **lstrip () and rstrip ():** return left trip version and right trip version of string.

**5**. **replace (old, new) :** replace substring from string.

**6.** **center ():** returned centered string.

|  |  |
| --- | --- |
| [count()](https://www.w3schools.com/python/ref_string_count.asp) | Returns the number of times a specified value occurs in a string |
| [encode()](https://www.w3schools.com/python/ref_string_encode.asp) | Returns an encoded version of the string |
| [endswith()](https://www.w3schools.com/python/ref_string_endswith.asp) | Returns true if the string ends with the specified value |
| [expandtabs()](https://www.w3schools.com/python/ref_string_expandtabs.asp) | Sets the tab size of the string |
| [find()](https://www.w3schools.com/python/ref_string_find.asp) | Searches the string for a specified value and returns the position of where it was found |
| [format()](https://www.w3schools.com/python/ref_string_format.asp) | Formats specified values in a string |
| [format\_map()](https://www.w3schools.com/python/ref_string_format_map.asp) | Formats specified values from a dictionary in a string |
| [index()](https://www.w3schools.com/python/ref_string_index.asp) | Searches the string for a specified value and returns the position of where it was found |
| [isalnum()](https://www.w3schools.com/python/ref_string_isalnum.asp) | Returns True if all characters in the string are alphanumeric |
| [isalpha()](https://www.w3schools.com/python/ref_string_isalpha.asp) | Returns True if all characters in the string are in the alphabet |
| [isascii()](https://www.w3schools.com/python/ref_string_isascii.asp) | Returns True if all characters in the string are ascii characters |
| [isdecimal()](https://www.w3schools.com/python/ref_string_isdecimal.asp) | Returns True if all characters in the string are decimals |
| [isdigit()](https://www.w3schools.com/python/ref_string_isdigit.asp) | Returns True if all characters in the string are digits |
| [isidentifier()](https://www.w3schools.com/python/ref_string_isidentifier.asp) | Returns True if the string is an identifier |
| [islower()](https://www.w3schools.com/python/ref_string_islower.asp) | Returns True if all characters in the string are lower case |
| [isnumeric()](https://www.w3schools.com/python/ref_string_isnumeric.asp) | Returns True if all characters in the string are numeric |
| [isprintable()](https://www.w3schools.com/python/ref_string_isprintable.asp) | Returns True if all characters in the string are printable |
| [isspace()](https://www.w3schools.com/python/ref_string_isspace.asp) | Returns True if all characters in the string are whitespaces |
| [istitle()](https://www.w3schools.com/python/ref_string_istitle.asp) | Returns True if the string follows the rules of a title |
| [isupper()](https://www.w3schools.com/python/ref_string_isupper.asp) | Returns True if all characters in the string are upper case |
| [join()](https://www.w3schools.com/python/ref_string_join.asp) | Converts the elements of an iterable into a string |
| [ljust()](https://www.w3schools.com/python/ref_string_ljust.asp) | Returns a left justified version of the string |
| [lower()](https://www.w3schools.com/python/ref_string_lower.asp) | Converts a string into lower case |
| [lstrip()](https://www.w3schools.com/python/ref_string_lstrip.asp) | Returns a left trim version of the string |
| [maketrans()](https://www.w3schools.com/python/ref_string_maketrans.asp) | Returns a translation table to be used in translations |
| [partition()](https://www.w3schools.com/python/ref_string_partition.asp) | Returns a tuple where the string is parted into three parts |
| [replace()](https://www.w3schools.com/python/ref_string_replace.asp) | Returns a string where a specified value is replaced with a specified value |
| [rfind()](https://www.w3schools.com/python/ref_string_rfind.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rindex()](https://www.w3schools.com/python/ref_string_rindex.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rjust()](https://www.w3schools.com/python/ref_string_rjust.asp) | Returns a right justified version of the string |
| [rpartition()](https://www.w3schools.com/python/ref_string_rpartition.asp) | Returns a tuple where the string is parted into three parts |
| [rsplit()](https://www.w3schools.com/python/ref_string_rsplit.asp) | Splits the string at the specified separator, and returns a list |
| [rstrip()](https://www.w3schools.com/python/ref_string_rstrip.asp) | Returns a right trim version of the string |
| [split()](https://www.w3schools.com/python/ref_string_split.asp) | Splits the string at the specified separator, and returns a list |
| [splitlines()](https://www.w3schools.com/python/ref_string_splitlines.asp) | Splits the string at line breaks and returns a list |
| [startswith()](https://www.w3schools.com/python/ref_string_startswith.asp) | Returns true if the string starts with the specified value |
| [strip()](https://www.w3schools.com/python/ref_string_strip.asp) | Returns a trimmed version of the string |
| [swapcase()](https://www.w3schools.com/python/ref_string_swapcase.asp) | Swaps cases, lower case becomes upper case and vice versa |
| [title()](https://www.w3schools.com/python/ref_string_title.asp) | Converts the first character of each word to upper case |
| [translate()](https://www.w3schools.com/python/ref_string_translate.asp) | Returns a translated string |
| [upper()](https://www.w3schools.com/python/ref_string_upper.asp) | Converts a string into upper case |
| [zfill()](https://www.w3schools.com/python/ref_string_zfill.asp) | Fills the string with a specified number of 0 values at the beginning |

**Accessing string characters**

You can access individual characters in a string using **indexing**. Python uses **zero-based indexing**, so the first character has an index of 0.

text = "Python"

print(text[0]) # Output: P

print(text[2]) # Output: t

You can also use **negative indexing** to start counting from the end of the string.

print(text[-1]) # Output: n

print(text[-3]) # Output: h

**Slicing Strings:**

You can extract a portion (substring) of a string using slicing.

text = "Python Programming"

print(text[0:6]) # Output: Python (extracts from index 0 to 5)

print(text[:6]) # Output: Python (same as above)

print(text[7:]) # Output: Programming (from index 7 to the end)

**Special case:**

**string [start : end: skip/step]**

name= sindhu

name[::2] ouput: snh

**Escape sequence:**

Escape sequence spl characters in string that starts with backslash(\)

\n : newline

\t : tab space

\\ : backslash

**Operators:**

**1. Assignment Operators**

Assignment operators are used to assign values to variables. The simplest one is = which assigns the value on the right to the variable on the left. There are also compound assignment operators that combine arithmetic operations with assignment**.**

**Common Assignment Operators:**

=:Assigns value on the right to the variable on the left.

+=: Adds right operand to the left operand and assigns the result to the left operand.

-=: Subtracts the right operand from the left operand and assigns the result to the left operand.

\*=: Multiplies the left operand by the right operand and assigns the result to the left operand.

/=: Divides the left operand by the right operand and assigns the result to the left operand.

%=: Takes modulus of left operand by right operand and assigns the result to the left operand.

Examples:

x = 5 # Assigns 5 to x

x += 3 # Equivalent to x = x + 3, now x is 8

x -= 2 # Equivalent to x = x - 2, now x is 6

x \*= 4 # Equivalent to x = x \* 4, now x is 24

x /= 6 # Equivalent to x = x / 6, now x is 4.0

**2. Comparison Operators**

Comparison operators are used to compare two values. They return either True or False depending on the condition.

**Common Comparison Operators:**

==: Checks if two values are equal.

!=: Checks if two values are not equal.

>: Checks if the left operand is greater than the right operand.

<: Checks if the left operand is less than the right operand.

>=: Checks if the left operand is greater than or equal to the right operand.

<=: Checks if the left operand is less than or equal to the right operand.

Examples:

a = 10

b = 20

print(a == b) # Output: False

print(a != b) # Output: True

print(a > b) # Output: False

print(a < b) # Output: True

print(a >= 10) # Output: True

print(b <= 25) # Output: True

**3. Logical Operators**

Logical operators are used to combine conditional statements. They evaluate expressions and return either True or False.

**Common Logical Operators:**

**and:** Returns True if both conditions are true.

**or:** Returns True if at least one condition is true.

**not:** Reverses the logical state of its operand (True becomes False, and vice versa).

Examples:

x = 5

y = 10

z = 15

# and operator

print(x > 0 and y > 5) # Output: True (both conditions are True)

# or operator

print(x > 10 or z > 10) # Output: True (one of the conditions is True)

# not operator

print(not(x > 10)) # Output: True (reverses False to True)

**4. Membership Operators**

Membership operators test for membership within a sequence, such as a list, string, or tuple. They return True or False based on whether the value is found in the sequence.

**Membership Operators:**

**in:** Returns True if the specified value is found in the sequence.

**not in:** Returns True if the specified value is not found in the sequence.

Examples:

my\_list = [1, 2, 3, 4, 5]

my\_string = "Python"

print(3 in my\_list) # Output: True (3 is in the list)

print(6 not in my\_list) # Output: True (6 is not in the list)

print("P" in my\_string) # Output: True ("P" is in the string)

print("z" not in my\_string) # Output: True ("z" is not in the string)

**5. Bitwise Operators**

Bitwise operators perform operations on binary representations of integers. These operators are useful for low-level programming tasks like working with bits and bytes.

**Common Bitwise Operators:**

&: Bitwise AND (sets each bit to 1 if both bits are 1).

|: Bitwise OR (sets each bit to 1 if one of the bits is 1).

^: Bitwise XOR (sets each bit to 1 if only one of the bits is 1).

~: Bitwise NOT (inverts all the bits).

<<: Left shift (shifts bits to the left by a specified number of positions).

>>: Right shift (shifts bits to the right by a specified number of positions).

Examples:

a = 5 # In binary: 101

b = 3 # In binary: 011

# Bitwise AND

print (a & b) # Output: 1 (binary: 001)

# Bitwise OR

print (a | b) # Output: 7 (binary: 111)

# Bitwise XOR

print(a ^ b) # Output: 6 (binary: 110)

# Bitwise NOT

print(~a) # Output: -6 (inverts all bits)

# Left shift

print(a << 1) # Output: 10 (binary: 1010)

# Right shift

print(a >> 1) # Output: 2 (binary: 010)

In Python, the **decimal data type** is not built-in but can be imported from the decimal module. The other data types listed (int, float, and str) are built-in data types in Python.

from decimal import Decimal

# Using the Decimal data type to perform precise arithmetic

x = Decimal('10.5')

y = Decimal('3')

result = x / y

print(result) # Output: 3.500000000000000000000000000

**Which of the following statements about Python constants is true?**

They are variables whose values cannot be changed.

They are defined using the const keyword.

Python doesn't have a built-in constant type.

None of the above

**Answer:** Option

**Explanation:**

Python doesn't have a built-in constant type, unlike some other programming languages. Conventionally, constants in Python are defined using all uppercase letters to indicate that their values should not be changed, but this is only a convention and not enforced by the language.

* The **pass statement** in Python is used as a placeholder for code that has not been implemented yet.

It is a keyword that creates an empty code block and does nothing. It is commonly used when defining functions, loops, and conditional statements that will be filled in later.

* Tuples are a **sequence data type** in Python, meaning that their elements are ordered and indexed, and they can be accessed by their position in the sequence. Dictionaries and sets are not sequence data types, because they are unordered.
* The bytearray data type is used to represent a mutable sequence of bytes, rather than characters. While a bytearray can hold character data, it is not specifically designed to represent a sequence of characters like the str data type.
* The bytes data type in Python is used to represent a sequence of bytes. It is similar to the bytearray data type, but it is immutable, meaning it cannot be changed after it is created. While the bytes type can hold character data, it is not specifically designed to represent a sequence of characters like the str data type.
* The complex data type in Python is used to represent complex numbers, which are numbers with a real and imaginary part (e.g., 3 + 4j). This data type is not used to represent a sequence of characters, but rather to work with complex mathematical operations involving real and imaginary numbers.
* The str type is the standard way to represent sequences of characters in Python.
* The **bool()** function can be used to convert a number to a boolean. 0 becomes False, and any non-zero value becomes True.
* The **split()** method in Python splits a string into a list of substrings based on a specified separator. If no separator is provided, it defaults to splitting the string by **"whitespace".**

**Lists in Python**

**1. What is a List?**

A list is a collection of items that **are ordered,** **mutable (changeable**), and allow **duplicate elements**. Lists can hold items of **different data types**, such as integers, strings, or even other lists.

Syntax:

my\_list = [element1, element2, element3, ...]

Example:

fruits = ["apple", "banana", "cherry"]

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

mixed = ["apple", 3, True]

**2. Accessing List Elements**

You can access individual elements in a **list using indexing**. Remember that Python uses **zero-based indexing**, so the first item is at index 0.

**Syntax:**

list\_name[index]

Example:

fruits = ["apple", "banana", "cherry"]

print(fruits[0]) # Output: apple

print(fruits[2]) # Output: cherry

You can also use **negative indexing** to access elements from the end of the list:

print(fruits[-1]) # Output: cherry

print(fruits[-2]) # Output: banana

**3. Modifying Lists**

Lists are **mutable,** which means you can change the value of items in a list.

Changing a specific element:

fruits[1] = "orange"

print(fruits) # Output: ['apple', 'orange', 'cherry']

**Adding elements:**

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

**fruits.append("grape")**

print(fruits) # Output: ['apple', 'orange', 'cherry', 'grape']

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

fruits.insert(1, "kiwi")

print(fruits) # Output: ['apple', 'kiwi', 'orange', 'cherry']

**extend() :** if we want add more than one element, it takes list that we want to add as an argument. i.e extend the list with another list.

**Note:** Rember the difference between append() and extend().

**Removing elements:**

**1.remove():** Removes the first occurrence of an element.

fruits.remove("orange")

print(fruits) # Output: ['apple', 'kiwi', 'cherry']

**2.pop():** Removes the element at a specific index (or the last item if no index is provided).

fruits.pop() # Removes the last item

print(fruits) # Output: ['apple', 'kiwi']

fruits.pop(0) # Removes the first item

print(fruits) # Output: ['kiwi']

**3.clear():** Removes all elements from the list. But it will return empty list.

fruits.clear()

print(fruits) # Output: []

**4.del keyword:** delete the list completely. Does not return any list.

**del list[index] or del list**

**Note :** remember the difference between **del keyword** and **clear()**

**4. Slicing Lists**

You can extract a portion of a list using slicing.

Syntax:

list\_name[start:stop:step]

start: The index to start the slice (inclusive).

stop: The index to stop the slice (exclusive).

step: The number of steps to **skip elements (default is 1**) i.e don’t skip anything.

Examples:

numbers = [0, 1, 2, 3, 4, 5, 6]

print(numbers[1:4]) # Output: [1, 2, 3] (from index 1 to 3)

print(numbers[:4]) # Output: [0, 1, 2, 3] (from start to index 3)

print(numbers[2:]) # Output: [2, 3, 4, 5, 6] (from index 2 to end)

print(numbers[::2]) # Output: [0, 2, 4, 6] (every 2nd element)

**5. List Functions and Methods**

Python provides several **built-in functions** and **methods** for working with lists.

**5.1 Common Functions:**

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

print(len(fruits)) # Output: 3

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

numbers = [5, 2, 9, 1]

print(sorted(numbers)) # Output: [1, 2, 5, 9]

print(numbers) # **Original list remains unchanged: [5, 2, 9, 1]**

**sum(list):** Returns the sum of elements in a list (**for numerical lists**).

numbers = [1, 2, 3, 4]

print(sum(numbers)) # Output: 10

**5.2 Common Methods:**

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

print(fruits.index("apple")) # Output: 0

**count(element):** Returns the number of occurrences of an element in the list.

numbers = [1, 2, 3, 1, 1]

print(numbers.count(1)) # Output: 3

**reverse():** Reverses the elements of the list in place.

fruits.reverse()

print(fruits) # Output: ['cherry', 'orange', 'apple']

**sort():** Sorts the list in place **(ascending by default).**

numbers = [5, 2, 9, 1]

numbers.sort()

print(numbers) # Output: [1, 2, 5, 9]

**6. Nested Lists**

Lists can **contain other lists**, allowing you to create nested lists. This can be **useful for storing matrix-like data structures.**

Example:

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

# Accessing elements in a nested list

print(matrix[0]) # Output: [1, 2, 3] (first row)

print(matrix[1][1]) # Output: 5 (element in the second row, second column)

**The extend()** method is used to add elements from one list to the end of another list.

The **copy()** function is used to create a shallow copy of a list in Python.

**What is the correct way to create a new list by multiplying each element of an existing list by 2 in Python?**

new\_list = list \* 2

new\_list = [element \* 2 for element in old\_list]

new\_list = old\_list.multiply(2)

new\_list = old\_list \* 2

**Answer:** Option b

* The **clear method** in Python lists is used to remove all elements from the list, **leaving it empty.**
* **The + operator** is used to concatenate two lists in Python.
* **The remove method** removes an element **by value**, while the **pop method** removes an element **by index.**
* **The list() constructor** in Python can be used to **create an empty list**, **convert a tuple to a list**, and **convert a string to a list**.
* The **rindex()** method is used to find the index of the last occurrence of a specific element in a list.
* How can you remove all occurrences of a specific element from a list?

list.remove\_all(element)

list.delete(element)

list.remove(element, all=True)

list = [x for x in list if x != element]

**Answer:** Option

**Explanation:**

To remove all occurrences of a specific element, you can use a list comprehension to filter out the unwanted element.

* Checking if not list: is a common way to check if a list is empty in Python.
* Using **set() to** remove duplicates and then converting it back to a list creates a new list with unique elements.
* How can you check if a list contains only numeric elements in Python?

all(isinstance(x, int) for x in my\_list)

Using all() with isinstance() checks if all elements in the list are of type int.

* How can you create a list of squared values from another list in Python?

squared\_list = list.square\_values()

squared\_list = [x^2 for x in original\_list]

squared\_list = list.map(lambda x: x\*\*2, original\_list)

squared\_list = [x\*\*2 for x in original\_list]

**Answer:** Option d

**Technical Coding Answers**

**1. Reverse a list without reverse() or slicing**

def reverse\_list(lst):

result = []

for i in lst:

result = [i] + result

return result

**2. Find largest and smallest elements**

lst = [3, 1, 7, 5]

print(max(lst)) # 7

print(min(lst)) # 1

**3. Count frequency of each element**

lst = [1, 2, 2, 3, 1]

freq = {}

for item in lst:

freq[item] = freq.get(item, 0) + 1

print(freq)

**4. Check if list is palindrome**

def is\_palindrome(lst):

return lst == lst[::-1]

**5. Remove duplicates without using set**

def remove\_duplicates(lst):

result = []

for i in lst:

if i not in result:

result.append(i)

return result

**6. Flatten a nested list**

def flatten(lst):

result = []

for i in lst:

if isinstance(i, list):

result += flatten(i)

else:

result.append(i)

return result

**7. Rotate a list by k elements**

def rotate\_list(lst, k):

k = k % len(lst)

return lst[-k:] + lst[:-k]

**8. Find second largest number**

def second\_largest(lst):

unique = list(set(lst))

unique.sort()

return unique[-2]

**9. Split list into two halves**

def split\_list(lst):

mid = len(lst) // 2

return lst[:mid], lst[mid:]

**10. Find all pairs with given sum**

def find\_pairs(lst, target):

pairs = []

seen = set()

for num in lst:

if target - num in seen:

pairs.append((num, target - num))

seen.add(num)

return pairs

**. Tuples in Python**

A tuple is a collection of items that is **ordered and immutable** (unchangeable). Tuples are similar to lists, but once a tuple is created, you cannot modify it. They are often used to group related data together.

**Syntax:**

my\_tuple = (element1, element2, element3, ...)

Example:

my\_tuple = ("apple", "banana", "cherry")

numbers\_tuple = (1, 2, 3, 4)

**Creating a Tuple with One Element:**

To create a tuple with **only one element, include a trailing comma:**

**single\_element\_tuple = ("apple",)**

**2. Accessing Tuple Elements**

You can access elements in a tuple using **indexing,** just like with lists. Tuples also support **negative indexing.**

Example:

fruits = ("apple", "banana", "cherry")

print(fruits[0]) # Output: apple

print(fruits[-1]) # Output: cherry

**Slicing Tuples:**

You can also slice tuples to access a subset of the elements.

print(fruits[1:3]) # Output: ('banana', 'cherry')

**3. Tuple Operations**

Although tuples are immutable, you can perform various operations with them.

**Tuple Concatenation:**

You can combine two or more tuples using **the + operator.**

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

combined\_tuple = tuple1 + tuple2

print(combined\_tuple) # Output: (1, 2, 3, 4, 5, 6)

**Tuple Repetition:**

You can repeat a tuple multiple times using **the \* operator.**

repeated\_tuple = (1, 2) \* 3

print(repeated\_tuple) # Output: (1, 2, 1, 2, 1, 2)

**Checking Membership:**

You can check if an item exists in a tuple using the in operator.

print("apple" in fruits) # Output: True

**4. Tuple Methods**

Though tuples are immutable, Python provides some built-in methods for working with tuples.

**count():** Returns the number of times an element appears in the tuple.

my\_tuple = (1, 2, 3, 1, 1)

print(my\_tuple.count(1)) # Output: 3

**index():** Returns the index of the first occurrence of an element.

my\_tuple = ("apple", "banana", "cherry")

print(my\_tuple.index("banana")) # Output: 1

**5. Advantages of Using Tuples**

**Immutable:** This property ensures that tuple data cannot be modified after creation, making them useful for fixed data.

**Faster than Lists:** Due to immutability, tuples are generally faster than lists.

Can Be Used as Keys in Dictionaries: Since tuples are hashable, they can be used as keys in dictionaries, unlike lists.

**6. Sets in Python**

A set is a collection of unique items that is unordered and unindexed. Sets do not allow duplicate values. Sets are useful for performing operations like union, intersection, and difference.

**Syntax:**

my\_set = {element1, element2, element3, ...}

Example:

fruits\_set = {"apple", "banana", "cherry"}

numbers\_set = {1, 2, 3, 4, 5}

**Empty Set:**

To create an empty set, use the set() function (not {}, which creates an empty dictionary).

empty\_set = set()

**7. Set Operations**

Sets support mathematical operations like union, intersection, and difference.

**Union:**

The union of two sets combines all elements from both sets, removing duplicates.

set1 = {1, 2, 3}

set2 = {3, 4, 5}

union\_set = set1 | set2 # Output: {1, 2, 3, 4, 5}

**Intersection:**

The intersection of two sets returns elements that are common to both sets.

intersection\_set = set1 & set2 # Output: {3}

**Difference:**

The difference between two sets returns elements that are in the first set but not in the second.

difference\_set = set1 - set2 # Output: {1, 2}

**Symmetric Difference:**

The symmetric difference returns elements that are in either of the sets but not in both.

sym\_diff\_set = set1 ^ set2 # Output: {1, 2, 4, 5}

8. Set Methods

Sets come with several useful methods for performing common tasks.

**add():** Adds an element to the set.

fruits\_set.add("orange")

print(fruits\_set) # Output: {'apple', 'banana', 'cherry', 'orange'}

**remove():** Removes a specified element from the set. Raises an error if the element does not exist.

fruits\_set.remove("banana")

print(fruits\_set) # Output: {'apple', 'cherry'}

**discard():** **Removes a specified element without raising an error if it does not exist.**

fruits\_set.discard("banana") # No error if "banana" is not in the set

**pop():** Removes a random element from the set.

fruits\_set.pop()

**clear():** Removes all elements from the set.

fruits\_set.clear()

**9. Differences Between Lists, Tuples, and Sets**

**Feature List Tuple Set**

**Ordering** Ordered Ordered Unordered

**Mutability** Mutable Immutable Mutable

**Duplicates** Allows duplicates Allows duplicates No duplicates

**Indexing** Supports indexing Supports indexing No indexing

**Common Uses** General collection Fixed data Unique items