Lists, Tuple, Dictionary, Set

- Python provides us with lists, tuples, dictionaries and set, all of which have become synonym with ease of programming and can be used in diverse applications.
- In Python,
  - A **list** can contain different types of elements and is mutable.
  - A **tuple** can contain different types of elements and is immutable.
  - A **dictionary**, the items can be accessed using strings as indices.
  - A **set** is a collection which is *unordered*, *immutable*, and *unindexed*. Sets are used to store multiple items in a single variable.

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[] or list()`	`[5.7, 4, 'Yes', 5.7]`
Tuple	Yes	N0	`() or tuple()`	`(5.7, 4, 'Yes', 5.7)`
Dictionary	No	Yes	`{} or dict()`	`{'June':30, 'July':31}`
Set	No	Yes	`{} or set()`	`{5.7, 4, 'Yes'}`

Lists

- Python provides a type called a list that stores a sequential collection of elements.
- Lists are good for keeping track of things by their order, especially when the order and contents might change.
- Unlike strings, lists are mutable. You can change a list in place, add new elements, and delete or replace existing elements.
- The same value can occur more than once in a list.

List Creation

The list class defines lists. To create a list, you can use list's constructor, as follows:

```
list1 = list()                # Create an empty list
list2 = list([2, 3, 4])       # Create a list with elements 2, 3, 4
list3 = list(["red", "green", "blue"])     # Create a list with strings
list4 = list(range(3, 6))     # Create a list with elements 3, 4, 5
list5 = list('abcd')      # Create a list with characters 'a','b','c','d'
```

You can also create a list by using the following syntax, which is a little simpler:

```
list1 = []                    # Same as list()
list2 = [2, 3, 4]             # Same as list([2, 3, 4])
list3 = ["red", "green"]      # Same as list(["red", "green"])
list4 = [2, "three", 4]       # Create a list of heterogeneous values
list5 = ['a', 'b', 'c', 'd']  # Same as list('abcd')
```

The elements in a list are separated by commas and are enclosed by a pair of brackets (`[ ]`).

Accessing Values in Lists

- Similar to strings, lists can also be sliced and concatenated.
- To access values in lists, square brackets are used to slice along with the index or indices to get value stored at that index.

Example	Output

```
num_list = [12,43,23,65,34,87,34,65,42,45]
print("num_list: \n", num_list)
print("First element:\n", num_list[0])
print("Elements 2 to 5:\n", num_list[2:5])
print("Even indexed elements: \n", num_list[::2])
print("Odd indexed elements: \n", num_list[1::2])
print("Reversed List:\n", num_list[::-1])
```

```
num_list:
 [12, 43, 23, 65, 34, 87, 34, 65, 42, 45]
First element:
 12
Elements 2 to 5:
 [23, 65, 34]
Even indexed elements:
 [12, 23, 34, 34, 42]
Odd indexed elements:
 [43, 65, 87, 65, 45]
Reversed List:
 [45, 42, 65, 34, 87, 34, 65, 23, 43, 12]
```

Deleting Values in Lists

- Items in a list can also be deleted by assigning an empty list to a slice of elements as shown below.

Example	Output

```
list1 = list('PROGRAM')
print('list1=', list1)
list1[2:5] = []
print('After deletion: ')
print('list1=', list1)
```

```
list1= ['P', 'R', 'O', 'G', 'R', 'A', 'M']
After deletion:
list1= ['P', 'R', 'A', 'M']
```

- Elements of the list or the complete list can be deleted by using `del` method.

Example	Output

```
num_list = [1,2,3,4,5,6,7,8]     # a list is defined
print('Original List: \n', num_list)
del num_list[2:4]
print('List with deleted elements: \n',num_list)
                  # deletes numbers at index 2 and 3

del num_list
print('Printing after deleting the list.')
print(num_list)
```

```
Original List:
 [1, 2, 3, 4, 5, 6, 7, 8]
List with deleted elements:
 [1, 2, 5, 6, 7, 8]
Printing after deleting the list.
Traceback (most recent call last):
  File "C:/Users/gyana/OneDrive/CM_FDP/
AIML/MLW Course 2024/Programs/listDemo.
py", line 58, in <module>
    print(num_list)
NameError: name 'num_list' is not defin
ed. Did you mean: 'num_list1'?
```

Updating Values in Lists

- Once created, one or more elements of a list can be easily updated by giving the slice on the left-hand side of the assignment operator.
- You can also append new values in the list using the `append()` method.

Example	Output

```
num_list = [4,3,6,5,7]
print("num_list: \n", num_list)
num_list[3] = 22
print("List after updation:\n", num_list)
num_list.append(33)
print("List after appending:\n", num_list)
del num_list[2]
print("List after deleting:\n", num_list)
```

```
num_list:
 [4, 3, 6, 5, 7]
List after updation:
 [4, 3, 6, 22, 7]
List after appending:
 [4, 3, 6, 22, 7, 33]
List after deleting:
 [4, 3, 22, 7, 33]
```

Nested Lists

- Nested list means a list within another list.
- A list can contain elements of different data types which can include even a list.

Example

```
num_list1 = [4,3,6]
print("num_list: \n", num_list1)
nested_list = ['a', 'd', num_list1, 't', 'k', '5']
print("Nested list: \n", nested_list)
print("Nested list item [0]:", nested_list[0])
print("Nested list item [1]:", nested_list[1])
print("Nested list item [2]:", nested_list[2])
print("Nested list item [3]:", nested_list[3])
print("Nested list item [4]:", nested_list[4])
print()
print("Nested list item [2][0]:", nested_list[2][0])
print("Nested list item [2][1]:", nested_list[2][1])
print("Nested list item [2][2]:", nested_list[2][2])
```

Output

```
num_list:
 [4, 3, 6]
Nested list:
 ['a', 'd', [4, 3, 6], 't', 'k', '5']
Nested list item [0]: a
Nested list item [1]: d
Nested list item [2]: [4, 3, 6]
Nested list item [3]: t
Nested list item [4]: k

Nested list item [2][0]: 4
Nested list item [2][1]: 3
Nested list item [2][2]: 6
```

Sequence Operations in Lists & Strings

Operation	Description	Example	Output
`x in s`	`True` if element `x` is in sequence `s`.	`'a' in ['a','e','i','o','u']`	`True`
`x not in s`	`True` if element `x` is not in sequence `s`.	`'x' not in ['a','e','i','o','u']`	`True`
`s1 + s2`	Concatenates two sequences `s1` and `s2`.	`[1,2,3,4] + [5,6,7,8]`	`[1,2,3,4,5,6,7,8]`
`s * n`, or `n * s`	`n` copies of sequence `s` concatenated.	`2*[1,2,3,4]`	`[1, 2, 3, 4, 1, 2, 3, 4]`
`s[i]`	i^{th} element in sequence `s`.	`s = [1,2,3,4]` `s[1]`	`2`
`s[i : j]`	Slice of sequence `s` from index `i` to `j-1`	`s = [1,2,3,4]` `s[1:4]`	`[2,3,4]`
`len(s)`	Length of sequence `s`, i.e., the number of elements in `s`.	`s = [1,2,3,4]` `len(s)`	`4`
`min(s)`	Smallest element in sequence `s`. (does not work for heterogenous lists)	`s = [1,2,3,4]` `min(s)`	`1`
		`y = ['a','b','c','d']` `min(y)`	`'a'`
`max(s)`	Largest element in sequence `s`. (does not work for heterogenous lists)	`s = [1,2,3,4]` `max(s)`	`4`
		`y = ['a','b','c','d']` `max(y)`	`'d'`
`sum(s)`		`s = [1,2,3,4]` `sum(s)`	`10`

	Sum of all numbers in sequence `s`. (only works for numeric lists)	`y = ['a','b','c','d']` `sum(y)`	`TypeError`
`for loop`	Traverses elements from left to right in a `for` loop.		
`<, <=, >, >=, =, !=`	Compares two sequences.		
`all()`	Returns `True` if all elements of the list are true (non-zero) (or if the list is empty)	`num_list1 = [1,2,3,4]` `print(all(num_list1))`	`True`
		`num_list2 = [0,2,3,4]` `print(all(num_list2))`	`False`
`any()`	Returns `True` if any element of the list is true. If the list is empty, returns `False`	`num list = [1,2,3,4]` `print(any(num_list))`	`True`
`list()`	Converts an iterable (tuple. string, set, dictionary) to a list	`list1 = 'HELLO'` `print(list1)`	`['H','E','L','L','O']`

`sorted()`	Returns a new sorted list. The original list is not sorted.	`list1 = [3,4,2,7,5,8]`
		`list2 = sorted(listl)`
		`print(list2)`
			`[2, 3, 4, 5, 7, 8]`

Python List Operations

Python lists are versatile and provide a range of methods to perform operations on list items. Here's a quick overview of some common list methods with examples

- **Adding Items**

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

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

  `extend()`: Extends the list by adding all elements from an iterable.

```
fruits = ['apple', 'banana']
fruits.extend(['cherry', 'date'])
print(fruits) # Output: ['apple', 'banana', 'cherry', 'date']
```

  `insert()`: Inserts an element at a specified position.

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

- **Removing Items**

  `remove()`: Removes the first occurrence of an element with the specified value.

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

  `pop()`: Removes the element at the specified position and returns it.

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

  `clear()`: Removes all elements from the list.

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

- **Other Operations**

  `index()`: Returns the index of the first element with the specified value.

```
fruits = ['apple', 'banana', 'cherry']
index_of_banana = fruits.index('banana')
print(index_of_banana) # Output: 1
```

  `count()`: Returns the number of elements with the specified value.

```
fruits = ['apple', 'banana', 'cherry', 'banana']
count_of_banana = fruits.count('banana')
print(count_of_banana) # Output: 2
```

  `sort()`: Sorts the list in ascending order by default, uses ASCII values to sort the list.

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

  `reverse()`: Reverses the order of the list.

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

  `copy()`: Returns a shallow copy of the list.

```
fruits = ['apple', 'banana', 'cherry']
fruits_copy = fruits.copy()
print(fruits_copy) # Output: ['apple', 'banana', 'cherry']
```

Key points to remember

- `insert()`, `remove()`, and `sort()` methods only modify the list and do not return any value.
- If you print the return values of these methods, you will get `None`.
- This is a design principle that is applicable to all mutable data structures in Python. The code given below illustrates this point.

Example	Output

```
num_list = [100, 200, 300, 400]
print(num_list.insert(2, 250))
```

```
None
```

- When one list is assigned to another list using the assignment operator then a new copy of the list is not made. Instead, assignment makes the two variables point to the one list in memory. This is also known as *aliasing*.
- Any change of made to one of the lists makes corresponding changes to the other list.

Example 1

```
num_list1 = [1,2,3,4,5]
num_list2 = num_list1
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
```

Output

```
list1: [1, 2, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1488365621952
ID of list2: 1488365621952


After modification of list1:
list1: [1, 2, 10, 3, 4, 5]
list2: [1, 2, 10, 3, 4, 5]
ID of list1: 1488365621952
ID of list2: 1488365621952
```

- To avoid the above one may use the `copy()` method or the list cloning using slicing.

Example 1

```
num_list1 = [1,2,3,4,5]
num_list2 = num_list1.copy()
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
```

Output

```
list1: [1, 2, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1784061850688
ID of list2: 1784064120576


After modification of list1:
list1: [1, 2, 10, 3, 4, 5]
list2: [1, 2, 3, 4, 5]
ID of list1: 1784061850688
ID of list2: 1784064120576
```

Example 2

```
num_list1 = [4,5,6,7,8]
num_list2 = num_list1[:]
print('list1:',num_list1)
print('list2:',num_list2)

print('ID of list1:',id(num_list1))
print('ID of list2:',id(num_list2))
print()


print('\nAfter modification of list1:')
num_list1.insert(2, 10)

print('list1:',num_list1)
print('list2:',num_list2)
```

Output

```
list1: [4, 5, 6, 7, 8]
list2: [4, 5, 6, 7, 8]
ID of list1: 2731107623040
ID of list2: 2731107489280


After modification of list1:
list1: [4, 5, 10, 6, 7, 8]
list2: [4, 5, 6, 7, 8]
ID of list1: 2731107623040
ID of list2: 2731107489280
```

Shallow Copy and Deep Copy

- In Python, we use = operator to create a copy of an object.
- You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.
- Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

- Shallow Copy
- Deep Copy

- To make these copy work, we use the copy module.

Shallow Copy

- A shallow copy creates a new object which stores the reference of the original elements.
- So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects.
- This means, a copy process does not recurse or create copies of nested objects itself.

Example

```
import copy
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)

print("Old list:\n", old_list)
print("New list (shallow copy):\n", new_list)

old_list.append([4, 4, 4])
print("Old list (changed):\n", old_list)
print("New list:\n", new_list)

old_list[1][1] = 'AA'
print("Old list:", old_list)
print("New list:", new_list)
```

Output

```
Old list:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list (shallow copy):
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Old list (changed):
 [[1, 2, 3], [4, 5, 6], [7, 8, 9], [4, 4, 4]]
New list:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Old list: [[1, 2, 3], [4, 'AA', 6], [7, 8, 9], [4, 4, 4]]
New list: [[1, 2, 3], [4, 'AA', 6], [7, 8, 9]]
```

- In the above program, we made changes to `old_list` i.e., `old_list[1][1] = 'AA'`. Both sublists of `old_list` and `new_list` at `index [1][1]` were modified. This is because, both lists share the reference of same nested objects.

Deep Copy

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

Example

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

print("Old list:\n", old_list)
print("New list (deep copy):", new_list)
print()
old_list[1][0] = 'BB'
print("Old list (updated):", old_list)
print("New list:", new_list)
```

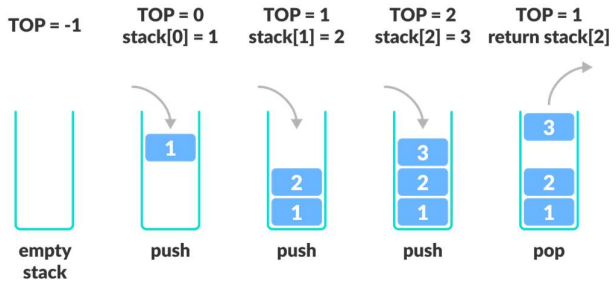
Output

```
Old list:
 [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list (deep copy): [[1, 1, 1], [2, 2, 2], [3, 3, 3]]

Old list (updated): [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to `old_list`, we can see only the `old_list` is modified. This means, both the `old_list` and the `new_list` are independent. This is because the `old_list` was recursively copied, which is true for all its nested objects.

List as Stack

- A stack is a fundamental data structure in computer science that follows the Last In, First Out (LIFO) principle. In simpler terms, the last element added to the stack is the first one to be removed.



- The basic operations associated with a stack include:
  - **Push**: Adding an element to the top of the stack.
  - **Pop**: Removing the element from the top of the stack.
  - **Peek (or Top)**: Viewing the element at the top of the stack without removing it.
  - **IsEmpty**: Checking if the stack is empty.
- **Push Operation Using List in Python**
  - A stack is initialized as an empty list, and elements are successively added to the top of the stack using the `append` method.
- **Pop Operation Using List in Python**
  - The stack is manipulated using the `pop` method, which removes and returns the last element.
- **Top Operation Using List in Python**
  - The top element of the stack is accessed using indexing `stack[-1]` that points to the last element (Top of Stack).

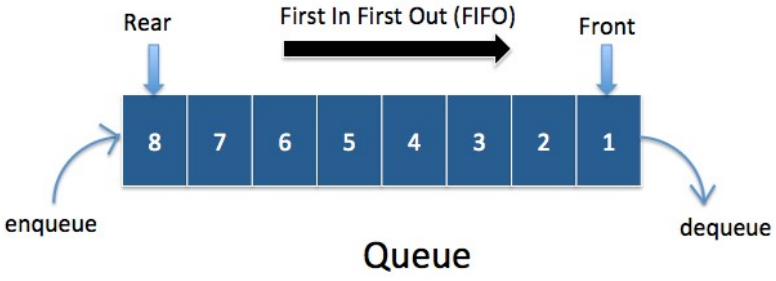
Example	Output

```
# Create an empty stack
stack = []

# Push elements onto the stack
stack.append(21)
stack.append(32)
stack.append(43)
print('Stack:\n', stack)

# Display the top of the stack
print('Top of Stack:\n', stack[-1])

# insert an element
stack.append(54)

# Display the stack
print('Stack (after push):\n', stack)

# element inserted at the back
print('Top of Stack after '+\
      'inserting an element:\n', stack[-1])

popped_element = stack.pop()
print('Popped Element:\n', popped_element)

# Display the updated stack
print('Stack (after pop):\n', stack)
```

```
Stack:
 [21, 32, 43]
Top of Stack:
 43
Stack (after push):
 [21, 32, 43, 54]
Top of Stack after inserting an element:
 54
Popped Element:
 54
Stack (after pop):
 [21, 32, 43]
```

List as Queue

- A queue is another essential data structure that follows the **First In, First Out (FIFO)** principle.
- In a queue, the first element added is the first one to be removed.
- Queues are commonly used in scenarios where tasks or processes are executed in the order they are received.
- Some common operations associated with queues include:
  - **Enqueue**: Adding an element to the rear (end) of the queue.
  - **Dequeue**: Removing the element from the front (head) of the queue.
  - **Front**: Viewing the element at the front without removing it.
  - **Rear**: Viewing the element at the rear without removing it.
  - **IsEmpty**: Checking if the queue is empty.



Example	Output

```
# Create an empty queue
queue = []

# Add elements onto the queue
queue.append('Chandragupta')
queue.append('Bindusar')
queue.append('Ashoka')

# Display the queue
print("Queue after inserting data:")
print(queue, '\n')

# Display the front of queue
print('Front:\n', queue[0])

# Display the rear of queue
print('Rear:\n', queue[-1])

print("\nElements removed from queue:")
print(" First  -> ",queue.pop(0))
print(" Second -> ",queue.pop(0))
print(" Third  -> ",queue.pop(0))

print("\nQueue after removing elements:")
print(queue)
```

```
Queue after inserting data:
['Chandragupta', 'Bindusar', 'Ashoka']

Front:
 Chandragupta
Rear:
 Ashoka

Elements removed from queue:
 First  ->  Chandragupta
 Second ->  Bindusar
 Third  ->  Ashoka

Queue after removing elements:
[]
```

Looping in Lists

- In python looping through lists can be done in following ways.

Example	Output

```
# Looping in lists using for loop
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Looping in lists Using a for Loop with range()
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
    print(fruits[i])

# Looping in lists Using a while Loop
fruits = ["apple", "banana", "cherry"]
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1

# Looping in list comprehension
fruits = ["apple", "banana", "cherry"]
[print(fruit) for fruit in fruits]
```

```
apple
banana
cherry

apple
banana
cherry

apple
banana
cherry

apple
banana
cherry
```

The Enumerate Function in Lists

- In Python, the `enumerate()` function is a built-in function that adds a counter to an iterable and returns it as an enumerate object.
- This can be particularly useful when you need both the item and its index while looping over an iterable.

Example	Output

```
colors = ["red", "green", "blue"]
for index, color in enumerate(colors):
    print(index, '--->',color)

# Using a custom start index
for index, color in enumerate(colors, start=1):
    print(index, '--->', color)
```

```
0 ---> red
1 ---> green
2 ---> blue

1 ---> red
2 ---> green
3 ---> blue
```

List Comprehensions

- List comprehensions in Python provide a concise way to create lists. They are often more readable and faster than traditional for loops.
- A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.
- **Syntax**

```
newList = [expression(element) for element in oldList if condition]
```

- **Parameter:**
  - `expression`: Represents the operation you want to execute on every item within the `iterable`.
  - `element`: The term "variable" refers to each value taken from the `iterable`.
  - `iterable`: specify the sequence of elements you want to iterate through. (e.g., a list, tuple, or string).
  - `condition`: (Optional) A filter helps decide whether an element should be added to the new list.
  - `Return`: The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.

Example 1

```
# Finding squares of a list of numbers
numbers = [1, 2, 3, 4, 5]
squared_normal = []

# Normal method
for i in numbers:
    squared_normal.append(i**2)

print('Number List: \n', numbers)
print('List of squares (normal loop):')
print('',squared_normal)

squared_LC = [x ** 2 for x in numbers]
print('List of squares (List Comprehension):')
print('',squared_LC)
```

Result

```
Number List:
 [1, 2, 3, 4, 5]
List of squares (normal loop):
 [1, 4, 9, 16, 25]
List of squares (List Comprehension):
 [1, 4, 9, 16, 25]
```

Example 2

```
# filtering even numbers from a list
# Normal method
even_nums = []
for i in range(1, 10):
    if i%2==0:
        even_nums.append(i)

print('List of even no.s (normal loop):')
print('',even_nums)

# List comprehension method
even_numbers = [num for num in range(1, 10) if num % 2 == 0 ]
print('List of even no.s (List Comprehension):')
print('',even_numbers)
```

Result

```
List of even no.s (normal loop):
 [2, 4, 6, 8]
List of even no.s (List Comprehension):
 [2, 4, 6, 8]
```

Example 3	Result

```
# find vowel in the string "Python"
word = "Python"
vowels = "aeiouAEIOU"

# Normal method
res = []
for i in word:
    if i in vowels:
        res.append(i)
print('List of vowels (normal loop):')
print('',res)

# List comprehension method
result = [char for char in word if char in vowels]
print('List of vowels (List Comprehension):')
print('',result)
```

```
List of vowels (normal loop):
 ['o']
List of vowels (List Comprehension):
 ['o']
```

Functional Programming in Lists

- Python provides several built-in functions based on expressions, which work faster than loop-based user defined code. They are
  - `map()`
  - `reduce()`
  - `filter()`
- **Map Function**
  - The function `map` is used for transforming every value in each sequence by applying a function to it.
  - It takes two input arguments:
    - The **iterable object** (i.e. object which can be iterated upon) to be processed and
    - the **function** to be applied
  - It returns the map object obtained by applying the function to the list.
  - **Syntax**

    ```
    result = map(function, iterable object)
    ```
  - The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a function object.

Example	Output

```
# using the map function
print('Using function:')

def cubes(x):
    return x**3

numlist = [4,-5,2,6,3]
cubelist = list(map(cubes, numlist))
print('List of numbers: \n', numlist)
print('List of cubes: \n', cubelist)


# using the map function: lambda function
print('Using lambda function:')

lamdaCubes = lambda x:x**3
lambdacubelist = list(map(lamdaCubes, numlist))

print('List of numbers: \n', numlist)
print('List of cubes: \n', lambdacubelist)
```

```
Using function:
List of numbers:
 [4, -5, 2, 6, 3]
List of cubes:
 [64, -125, 8, 216, 27]

Using lambda function:
List of numbers:
 [4, -5, 2, 6, 3]
List of cubes:
 [64, -125, 8, 216, 27]
```

- We can even pass more than one sequence in the `map()` function. There are two requirements explained as follows.

- The function must have as many arguments as there are sequences.
- Each argument is called with the corresponding item from each sequence (or none if one sequence is shorter than another).

Example

```
numlist1 = [4,-5,2,6,3]
numlist2 = [3,7,2,-8,2]
print('List 1: \n', numlist1)
print('List 2: \n', numlist2)

print('Sum of Cubes Using Function:')
def sum2cubes(x,y):
    return x**3 + y**3

numlist3 = list(map(sum2cubes, numlist1, numlist2))
print(numlist3)

print('Sum of Cubes Using Lambda Function:')
lamdaCubes = lambda x,y:x**3 + y**3
numlist4 = list(map(sum2cubes, numlist1, numlist2))

print(numlist4)
```

Output

```
List 1:
 [4, -5, 2, 6, 3]
List 2:
 [3, 7, 2, -8, 2]
Sum of Cubes Using Function:
[91, 218, 16, -296, 35]
Sum of Cubes Using Lambda Function:
[91, 218, 16, -296, 35]
```

The `filter()` Function

- The `filter()` function constructs a list from those elements of the list for which a function returns `True`.
- **Syntax**

```
filter(function, sequence)
```

- As per the syntax, the `filter()` function returns a sequence that contains items from the sequence for which the function is `True`.
- Essentially, `filter()` can be used to extract elements from an iterable that meets a certain condition.
- If sequence is a string, Unicode, or a tuple, then the result will be of the same type; otherwise, it is always a list.

Example

```
# using the filter() function
# to check whether numbers are divisible
# by 2 and 3
def check(x):
    if x%2==0 and x%3==0:
        return 1

numList = list(range(1,31))
print('List of numbers: \n', numList)

# using for loop
res23 = []
for i in numList:
    if check(i):
        res23.append(i)
print('Using for loop')
print('List of numbers divisible by 2 and 3: \n',res23)

# using filter()
print('Using filter()')
nums23 = list(filter(check, numList))
print('List of numbers divisible by 2 and 3: \n',nums23)

# using lambda function and filter()
print('using lambda function and filter()')
check23 = lambda x: x%2==0 and x%3==0
nums23new = list(filter(check23, numList))
print('List of numbers divisible by 2 and 3: \n',nums23new)
```

Output

```
List of numbers:
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30]

Using for loop
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]

Using filter()
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]

Using lambda function and filter()
List of numbers divisible by 2 and 3:
 [6, 12, 18, 24, 30]
```

The `reduce()` Function

- The `reduce()` function with syntax as given below returns a single value generated by calling the function on the first two items of the sequence, then on the result and the next item and so on
- **Syntax**

```
reduce(function, sequence)
```

- **Key points to remember**
  - If there is only one item in the sequence, then its value is returned.
  - If the sequence is empty, an exception is raised.
  - Creating a list in a very extensive range will generate a `MemoryError` or `OverflowError`.

Example

```
numbers = [6,3,4,7,2,9,1]
sum_result = reduce(lambda x, y: x + y, numbers)
max_result = reduce(lambda a, b: a if a>b else b, numbers)
min_result = reduce(lambda a, b: a if a<b else b, numbers)


print('List of Numbers: \n', numbers)
print('Sum of numbers =',sum_result)
print('Max of numbers =',max_result)
print('Min of numbers =',min_result)
```

Output

```
List of Numbers:
 [6, 3, 4, 7, 2, 9, 1]
Sum of numbers = 32
Max of numbers = 9
Min of numbers = 1
```

- **Optional Initializer Argument**
  - Reduce also accepts an optional third argument, initializer, which is used as the initial value to start the execution.
  - **Syntax**

```
reduce(function, sequence, initializer)
```

  - If the iterable is empty, reduce will return the initializer.
  - Without an initializer, an empty iterable would cause reduce to raise a `TypeError`.

Example	Output

```
from functools import reduce

# Define a function to sum the squares of two numbers
def sum_squares(x, y):
    return x + y**2

# Define a function to sum the cubes of two numbers
sum_cubes = lambda x,y:x+y**3

# List of numbers
numbers = [6,-3,4,-7,2,9,1]

# Use reduce to apply the sum_squares function to the list
result_square = reduce(sum_squares, numbers, 0)
result_cubes = reduce(sum_cubes, numbers, 0)

print('List of Numbers: \n', numbers)
print('Sum of square of numbers = ',result_square)
print('Sum of cube of numbers = ',result_cubes)
```

```
List of Numbers:
 [6, -3, 4, -7, 2, 9, 1]
Sum of square of numbers =  196
Sum of cube of numbers =  648
```

Tuple

- Tuple is a data structure supported by Python used to store collections of data.
- Tuples can store homogeneous as well as heterogeneous data.
- It is very similar to lists but differs in two things.
  - First, a tuple is a sequence of immutable objects. This means that while you can change the value of one or more items in a list, you cannot change the values in a tuple.
  - Second, tuples use parentheses to define its elements whereas lists use square brackets.

Creating and Using Tuples

To create a tuple, you can simply enclose the items within parentheses. For example:

```
mytuple = ("apple", "banana", "cherry")
print(mytuple)

mytuple = tuple(["apple", "banana", "cherry"])
print(mytuple)
```

If you need to create a tuple with only one item, you must include a comma after the item, otherwise, it will not be recognized as a tuple:

```
# Correct way to create a single item tuple
single_item_tuple = ("apple",)

# Incorrect, this is not a tuple but a string
not_a_tuple = ("apple")
```

Tuples can contain items of any data type and can even contain a mix of different types:

```
tuple1 = ("abc", 34, True, 40, "male")
```

Tuple Characteristics

- **Ordered**: The items in a tuple have a specific order that will not change.
- **Unchangeable**: Once a tuple is created, you cannot change, add, or remove items.
- **Allow Duplicates**: Tuples can have items with the same value, which means they can contain duplicates.
- **Indexing**: Each item in a tuple has an index, starting from 0.
- **Lightweight**: They consume relatively small amounts of memory compared to other sequences like lists.
- **Heterogeneous**: They can store objects of different data types and domains, including mutable objects.
- **Nestable**: They can contain other tuples, so you can have tuples of tuples.
- **Iterable**: They support iteration, so you can traverse them using a loop or comprehension while you perform operations with each of their elements.
- **Sliceable**: They support slicing operations, meaning that you can extract a series of elements from a tuple.
- **Combinable**: They support concatenation operations, so you can combine two or more tuples using the concatenation operators, which creates a new tuple.
- **Hashable**: They can work as keys in dictionaries when all the tuple items are immutable.

Accessing Tuple Items

- You can access tuple items by referring to the index number:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1]) # Outputs "banana"
```

- Negative indexing means starting from the end of the tuple:

```
print(thistuple[-1]) # Outputs "cherry"
```

- Like other Python sequences, tuples allow you to extract a portion or slice of their content with a slicing operation, which uses the following syntax:

```
tuple_object[start:stop:step]
```

Example:

```
>>> days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
            "Saturday", "Sunday",)
>>> print(days[:5])
   ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
>>> days[5:]
   ('Saturday', 'Sunday')
```

Immutability of Tuples

Tuples are immutable, which means that you cannot change tuple items after the tuple has been created. Attempting to do so will result in a TypeError:

```
thistuple = ("apple", "banana", "cherry")
thistuple[1] = "strawberry"      # Raises a TypeError
del thistuple[1]                 # Raises a TypeError
```

Tuple Length

To determine the number of items in a tuple, use the `len()` function:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple)) # Outputs 3
```

Tuple Operations

Tuples support various operations like concatenation, nesting, repetition, and slicing. For instance, you can concatenate tuples using the + operator:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(tuple1 + tuple2) # Outputs (1, 2, 3, 4, 5, 6)
```

Converting Between Tuples and Other Types

You can convert a tuple to a list to modify it, and then convert it back to a tuple:

```
mylist = list(mytuple)
mylist.append("orange")
mytuple = tuple(mylist)
```

Tuples can store any type of object, including mutable ones. This means that you can store lists, sets, dictionaries, and other mutable objects in a tuple. You can change the content of mutable objects even if they're nested in a tuple.

```
student_info = ("Linda", 18, ["Math", "Physics", "History"])
print(student_info[2][2])          # prints "History"

student_info[2][2] = "Computer Science"
print(student_info[2][2])          # prints "Computer Science"
```

Packing and Unpacking of Tuples

- Python has the notion of packing and unpacking tuples.
- For example, when you write an assignment statement like `point = x, y, z`, you're ***packing*** the values of `x`, `y`, and `z` in point. That's how you create new tuple objects.
- You can also do the inverse operation and ***unpack*** the values of a tuple into an appropriate number of variables.

```
>>> point = (7, 14, 21)          # Packing
>>> x, y, z = point              # Unpacking
>>> x
    7
>>> y
    14
>>> z
    21
```

- One can use the * operator in tuples pack and unpack the elements in tuples.

Example	Output

`Tup1 = (1,3,4,2,6,7,5)`
`*val1, val2 = Tup1`
`print('val1:\n',val1)`
`print('val2:\n',val2)`
 	
`val3, *val4 = Tup1`
`print('val3:\n',val3)`
`print('val4:\n',val4)`
 	
`val5, *val6, val7 = Tup1`
`print('val5:\n',val5)`
`print('val6:\n',val6)`
`print('val7:\n',val7)`
	`val1:`
	` [1, 3, 4, 2, 6, 7]`
	`val2:`
	` 5`
	`val3:`
	` 1`
	`val4:`
	` [3, 4, 2, 6, 7, 5]`
	`val5:`
	` 1`
	`val6:`
	` [3, 4, 2, 6, 7]`
	`val7:`
	` 5`

Example	Output

`Tup1 = (1,3,4,2)`
`Tup2 = (4,3,5,3)`
`print((Tup1, Tup2))`
`print((Tup1 + Tup2))`
`print((*Tup1, *Tup2))`
	`((1, 3, 4, 2), (4, 3, 5, 3))`
	`(1, 3, 4, 2, 4, 3, 5, 3)`
	`(1, 3, 4, 2, 4, 3, 5, 3)`

Basic Tuple Operations

- Like strings and lists, you can also perform operations like concatenation. repetition, etc on tuples.
- The only difference is that a new tuple should be created when a change is required in an existing tuple.

Operation	Expression	Output
Length	`len((4,5,3,5,2,3,7,9))`	`8`
Concatenation	`(1,2,3) + (4,5,6)`	`(1,2,3,4,5,6)`
Repetition	`("good.",)*3`	`('good.','good.','good.')`
Membership	`5 in (4,3,5,2,7,9)`	`True`
Iteration	`for i in (4,3,5,2,7,9):` `  print(i, end='')`	`4,3,5,2,7,9`

Comparison
(use <,>,==)
	`T1 = (1,2,3,4,5,6)`
	`T1 = (1,2,3,4,5,6)`
	`print(T1>T2)`
		`False`

Maximum	`max((4,3,5,2,7,9))`	`9`
Minimum	`min((4,3,5,2,7,9))`	`2`
Convert into a tuple (converts a sequence into tuple)	`tuple("hello")`	`('h', 'e', 'l', 'l', 'o')`

	`tuple([4,3,5,2,7,9])`	`(4,3,5,2,7,9)`

Index: returns the index of the first occurrence of the element `Tup.index(element)`	`T1 = (1,2,3,4,5,6)`
	`print(T1.index(3))`
	`print(T1.index(8))`
		
		`2`
		`ValueError`

Index: Find within a range defined between start and stop `Tup.index(element,start,stop)`	`T1 = (4,3,5,2,7,9)` `print(T1.index(7,2,5))`	 `4`

Count: returns the number of elements with a specific value	`T1=`
	`(1,2,3,4,5,6,5,3,5,6)`
	`print(T1.count(3))`
	`print(T1.count(25))`
		
		`2`
		`0`

Sum (not defined for strings)	`T1 = (4,3,5,2,7,9)` `print(T1.sum())`	`30`

The `zip` function

The function `zip` is used to produces a zip object (iterable object), whose i^{th} element is a tuple containing i^{th} element from each iterable object passed as argument to the zip function.

Example

```
# working of the zip function
colors = ('red', 'yellow', 'orange')
fruits = ['cherry', 'banana','orange']
quantity = ('1 kg', 12, '2 kg')
fruitColor = list(zip(colors, fruits))
print(fruitColor)
print()
fruitColorQuantity1 = list(zip(fruits, colors, quantity))
print(fruitColorQuantity1)
print()
fruitColorQuantity2 = list(zip(fruitColor, quantity))
print(fruitColorQuantity2)
print()
```

Output

```
[('red', 'cherry'), ('yellow', '
banana'), ('orange', 'orange')]

[('cherry', 'red', '1 kg'), ('ba
nana', 'yellow', 12), ('orange',
'orange', '2 kg')]

[(('red', 'cherry'), '1 kg'), ((
'yellow', 'banana'), 12), (('ora
nge', 'orange'), '2 kg')]
```

List Comprehension and Tuples

- The list comprehension concept can be extended to tuples to manipulate the values of one tuple to create a new tuple.

Example

```
# list comprehension in tuples
def double(T):
    return ([i*2 for i in T])
Tup = (1,2,3,4,5)
print('Normal method:')
print("Original values: ", Tup)
print("Double values  : ", double(Tup))
print()

print('Using comprehension:')
doubles1 = tuple(x*2 for x in Tup)
doubles2 = list(x*2 for x in Tup)
print('Creating a tuple:',doubles1)
print('Creating a list:',doubles2)
```

Output

```
Normal method:
Original values:  (1, 2, 3, 4, 5)
Double values  :  [2, 4, 6, 8, 10]

Using comprehension:
Creating a tuple: (2, 4, 6, 8, 10)
Creating a list: [2, 4, 6, 8, 10]
```

Dictionary

- Dictionary is a data structure in which we store values as a pair of key and value. Each key is separated from its value by a colon (:), and consecutive items are separated by commas. The entire items in a dictionary are enclosed in curly brackets({}).
- The syntax for defining a dictionary is

```
dictionary_name = {key_1:value_1, key_2:value_2, key_3:value_3}
```

or

```
dictionary_name = {key_1:value_1,
                   key_2:value_2,
                   key_3:value_3}
```

- The keys in the dictionary must be **unique** and be of any immutable data type (like strings, numbers, or tuples).
- There is no stringent requirement for uniqueness and type of values. That is, value of a key can be of any type.
- Dictionaries are not sequences, rather they are mappings. Mappings are collections of objects that store objects by key instead of by relative position.
- Dictionary keys are case-sensitive. Two keys with the same name but in different case are not the same in Python.

Creating Dictionaries

Example

```
print('Creating an empty dictionary:')
Dict1 = {}
print(Dict1)

print('Creating a dictionary with values:')
Dict2 = {'Name' : 'Arav',
         'Course' : 'BTech',
         'Branch' : 'CSE',
         'Specialization' : 'AIML',
         'Roll_No' : '16/001'}
print(Dict2)

print('Creating a dictionary with comprehension:')
Dict3 = {x: 2**x for x in range(1, 11)}
print(Dict3)
```

Output

```
Creating an empty dictionary:
{}
Creating a dictionary with values:
{'Name': 'Arav', 'Course': 'BTech', 'Branch': 'CS
E', 'Specialization': 'AIML', 'Roll_No': '16/001'
}
Creating a dictionary with comprehension:
{1: 2, 2: 4, 3: 8, 4: 16, 5: 32, 6: 64, 7: 128, 8
: 256, 9: 512, 10: 1024}
```

Accessing, Adding and Modifying Values in Dictionaries

Example

```
Dict = {'Name' : 'Arav',
        'Course' : 'BTech',
        'Branch' : 'CSE',
        'Spec' : 'AIML',
        'Roll_No' : '16/001'}

# Accessing the values of dictionaries
print('Name          :', Dict['Name'])
print('Course        :', Dict['Course'])
print('Branch        :', Dict['Branch'])
print('Specialization:', Dict['Spec'])
print('Roll No       :', Dict['Roll_No'])

# Adding a new key:value pair
Dict['CGPA'] = 9.67
print('CGPA          :', Dict['CGPA'])

# Changing a key:value pair
Dict['Name'] = 'Arav Gupta'
print('Name          :', Dict['Name'])
```

Output

```
Name          : Arav
Course        : BTech
Branch        : CSE
Specialization: AIML
Roll No       : 16/001

CGPA          : 9.67

Name          : Arav Gupta
```

Deletion of Dictionary Items

- One or more items can be deleted by using the `del` command.
- To remove all items `clear()` function can be used.
- To entirely delete the dictionary from the memory `del` command can be used.

Example

```
Dict = {'Name' : 'Arav',
         'Course' : 'BTech',
         'Branch' : 'CSE',
         'Spec' : 'AIML',
         'Roll_No' : '16/001'}
print('Original:')
print(Dict)

del Dict['Course']
print('\nAfter deletion of one item:')
print(Dict)

Dict.clear()
print('\nAfter clearing:')
print(Dict)

del Dict
print('\nAfter deletion of the whole:')
print(Dict)
```

Output

```
Original:
{'Name': 'Arav', 'Course': 'BTech', 'Branch': 'CS
E', 'Spec': 'AIML', 'Roll_No': '16/001'}

After deletion of one item:
{'Name': 'Arav', 'Branch': 'CSE', 'Spec': 'AIML',
'Roll_No': '16/001'}

After clearing:
{}

After deletion of the whole:
Traceback (most recent call last):
  File "C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW C
ourse 2024\Programs\dictDemo.py", line 62, in <mo
dule>
    print(Dict)
NameError: name 'Dict' is not defined. Did you me
an: 'Dict1'?
```

Beware of duplicate keys

- Keys must have unique values. Not even a single key can be duplicated in a dictionary.
- If you try to add a duplicate key, then the last assignment is retained. This is shown in the example given below.

Example

```
Dict = {'Name' : 'Arav',
         'Course' : 'BTech',
         'Branch' : 'CSE',
         'Spec' : 'AIML',
         'Roll_No' : '16/001',
         'Name': 'Manas'}
print('Effect of having duplicate keys:')
print('Name          :', Dict['Name'])
print('Course        :', Dict['Course'])
print('Branch        :', Dict['Branch'])
print('Specialization:', Dict['Spec'])
print('Roll No       :', Dict['Roll_No'])
```

Output

```
Effect of having duplicate keys:
Name          : Manas
Course        : BTech
Branch        : CSE
Specialization: AIML
Roll No       : 16/001
```

- In a dictionary, keys should be strictly of a type that is immutable.
- This means that a key can be of strings, number, or tuple type but it cannot be a list which is mutable.
- In case you try to make your key of a mutable type, then a `TypeError` will be generated as shown below.

Example

```
79 # Tuples and lists as keys of dictionary
80 d1 = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
81 print('d1:\n',d1)
82
83 d2 = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
84 print('d1:\n',d2)
```

Output

```
d1:
 {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
Traceback (most recent call last):
  File "C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW Course 2024\Pr
ograms\dictDemo.py", line 83, in <module>
    d2 = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions and Methods

Operation	Description	Example	Output
`len(Dict)`	Returns the length of dictionary, the number of items (key-value pairs)	`Dict1 = {'Roll_No':'16/001' 'Name': 'Arav', 'Course': 'BTech'}` `print(len(Dict1))`	`3`
`str(Dict)`	Returns a string representation of the dictionary	`Dict1 = {'Roll_No':'16/001' 'Name': 'Arav', 'Course': 'BTech'}` `print(str(Dict1))`	`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech'}`
`Dict.clear()`	Removes all the elements from the dictionary		`{}`
`Dict.copy()`	Returns a shallow copy of the dictionary		

`dict.fromkeys(seq, val)`	Returns a dictionary with the specified keys and value	`Subjects = [ 'ICP', 'DS', 'AA', 'ToC']`
		`Score = -1`
		`Marks = dict.fromkeys (Subjects, Score)`
		`print(Marks)`
			`{'ICP': -1, 'DS': -1, 'AA': -1, 'ToC': -1}`

`Dict.get(key)`	Returns the value of the specified key	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.get('Name'))`	`Arav`
`Dict.items()`	Returns a list containing a tuple for each key value pair	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.items())`	`dict_items([('Roll_No', '16/001'), ('Name', 'Arav'), ('Course', 'BTech')])`
`Dict.keys()`	Returns a list containing the dictionary's keys	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.keys())`	`dict_keys(['Roll_No', 'Name', 'Course'])`
`Dict.pop()`	Removes the element with the specified key		
`Dict.popitem()`	Removes the last inserted key-value pair		

`Dict.setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`Dict1.setdefault('Marks', 0)`
		`print(Dict1)`
			`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech', 'Marks': 0}`

	the specified value		

`Dict.update()`	Updates the dictionary with the specified key-value pairs	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`Dict2 = {'Marks': 86, 'Grade':'A'}`
		`Dict1.update(Dict2)`
		`print(Dict1)`
			`{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech', 'Marks': 86, 'Grade': 'A'}`

`Dict.values()`	Returns a list of all the values in the dictionary	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}` `print(Dict1.values())`	`dict_values(['16/001', 'Arav', 'BTech'])`

`in / not in`	Checks whether a given key is present in the dictionary or not.	`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
		`print('Name' in Dict1)`
		`print('Marks' in Dict1)`
			`True`
			`False`

Looping in Dictionaries

Example	Output

`Dict1 = {'Roll_No':'16/001', 'Name': 'Arav', 'Course': 'BTech'}`
`for i, j in Dict1.items():`
`  print(i, '=',j)`
	`Roll_No = 16/001`
	`Name = Arav`
	`Course = BTech`

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
  - Set items are unchangeable, but you can remove items and add new items.
- Sets are written with curly brackets.
- Sets are unordered, so you cannot be sure in which order the items will appear.
- Set items are unordered, unchangeable, and do not allow duplicate values.
  - **Unordered**
    - Unordered means that the items in a set do not have a defined order.
    - Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
  - **Unchangeable**
    - Set items are unchangeable, meaning that we cannot change the items after the set has been created.
    - Once a set is created, you cannot change its items, but you can remove items and add new items
  - **Duplicates Not Allowed**
    - Sets cannot have two items with the same value.
    - **Example**

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

Output

```
{'apple', 'cherry', 'banana'}
```

The `set()` Constructor

- It is also possible to use the `set()` constructor to make a set.
- **Examples**

```
thisset1 = set(("apple", "banana", "cherry")) # set using a tuple
thisset2 = set(["apple", "banana", "cherry"]) # set using a list
thisset3 = set({"apple":1, "banana":2, "cherry":3})
                                        # set using a dictionary

print(thisset1)
print(thisset2)
print(thisset3)
```

Output

```
{'banana', 'apple', 'cherry'}
{'banana', 'apple', 'cherry'}
{'banana', 'apple', 'cherry'}
```

Set Items - Data Types

- Set items can be of any data type, both homogeneous and heterogeneous.
- **Examples**

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
set1 = {"abc", 34, True, 40, "male"}
```

Access Set Items

- You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.
- **Example**
  - Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
      print(x)
```

Output

```
banana
apple
cherry
```

- **Example**
  - Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
print("orange" in thisset)
```

Output

```
True
False
```

Set Methods and Functions

Operation	Description
`len(setName)`	Determines how many items a set has
`setName.add(item)`	Adds an element to the set
`setName.clear()`	Removes all the elements from the set
`setName.copy()`	Returns a copy of the set
`set1.difference(set2)`	Returns a set containing the difference between two or more sets
`set1.difference_update(set2)`	Removes the items in this set that are also included in another, specified set
`setName.discard(item)`	Remove the specified item
`set1.intersection(set2)`	Returns a set, that is the intersection of two other sets
`set1.intersection_update(set2)`	Removes the items in this set that are not present in other, specified set(s)
`set1.isdisjoint(set2)`	Returns whether two sets have a intersection or not
`set1.issubset(set2)`	Returns whether another set contains this set or not
`set1.issuperset(set2)`	Returns whether this set contains another set or not
`setName.pop()`	Removes an element from the set
`setName.remove(item)`	Removes the specified element

`set1.symmetric_difference(set2)`	Returns a set with the symmetric differences of two sets
`set1.symmetric_difference_update(set2)`	inserts the symmetric differences from this set and another
`set1.union(set2)`	Return a set containing the union of sets
`set1.update(set2)`	Update the set with the union of this set and others
`set1 == set2 or set1 != set2`	Returns True (or False) if sets are equivalent

Change Items

- Once a set is created, you cannot change its items, but you can add new items.
- **Add Items**
  - Once a set is created, you cannot change its items, but you can add new items.
  - To add one item to a set use the `add()` method.
  - **Example**
    - Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Output

```
{'banana', 'apple', 'orange', 'cherry'}
```

- **Add Sets, Tuple, List or Dictionary**
  - To add items from another set, tuple, list or dictionary into the current set, use the `update()` method.
  - **Example**
    - Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
                                # Works with tuple, list or dictionary
thisset.update(tropical)
print(thisset)
```

Output

```
{'pineapple', 'mango', 'banana', 'papaya', 'apple', 'cherry'}
```

Remove Set Items

- To remove an item in a set, use the `remove()`, or the `discard()` method.
- If the item to remove does not exist, `remove()` will raise an error, `discard()` will NOT raise an error.
- **Example**
  - Remove `"banana"` by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Output

```
{'apple', 'cherry'}
```

- **Example**

- Remove `"banana"` by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Output

```
{'apple', 'cherry'}
```

- You can also use the `pop()` method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.
- The `clear()` method empties the set.
- The `del` keyword will delete the set completely.

Join Two Sets

- The `union()` method returns a **new** set containing all items from both sets.
  - **Example**

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

Output

```
{1, 'a', 2, 3, 'b', 'c'}
```

- The `update()` inserts the items in `set2` into `set1`.
  - **Example**

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)       # z = set1|set2 also works
print(set3)
```

Output

```
{1, 'a', 2, 3, 'b', 'c'}
```

- Both `union()` and `update()` will exclude any duplicate items.

Keep ONLY the Duplicates

- The `intersection()` method will return a **new** set, that only contains the items that are present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)   # z = x&y also works
print(z)
```

Output

```
{'apple'}
```

- The `intersection_update()` method will keep only the items that are present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)
```

Output

```
{‘apple’}
```

Keep All, But NOT the Duplicates

- The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)
```

  Output

```
{'google', 'cherry', 'microsoft', 'banana'}
```

- The `symmetric_difference_update()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference_update(y)
print(z)
```

  Output

```
{'google', 'cherry', 'microsoft', 'banana'}
```

Keep Difference Only

- The `difference()` method will return a new set, that contains only the elements that are present in one set but not in other.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z1 = x.difference(y)
print(z1)
z2 = x.difference(y)
print(z2)
```

  Output

```
{'banana', 'cherry'}
{'microsoft', 'google'}
```

- The `difference_update()` method will return a new set, that contains only the elements that are NOT present in both sets.
  - **Example**

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.difference_update(y)
print(x)
```

  Output

```
{'banana', 'cherry'}
```

Checking if Subset

- **Example**

```
x = {1,2,3,4,5,6}
y = {1,2,3,4,5,6,7,8,9,10}
# Checking if subset
print(x.issubset(y))
print(x<=y)
```

Output

```
True
True
```

Checking if Superset

- **Example**

```
x = {1,2,3,4,5,6}
y = {1,2,3,4,5,6,7,8,9,10}
# Checking if superset
print(y.issuperset(x))
print(y>=x)
```

Output

```
True
True
```

Checking if Disjoint

- **Example**

```
x1 = {1,2,3,4,5,6}
y1 = {1,2,3,4,5,6,7,8,9,10}
# Checking if disjoint
print(y1.isdisjoint(x1))

x2 = {1,2,3}
y2 = {4,5,6,7,8,9,10}
# Checking if disjoint
print(y2.isdisjoint(x2))
```

Output

```
False
True
```

Checking if Equivalent or not

- **Example**

```
x1 = {1,2,3,4,5,6}
y1 = {1,2,3,4,5,6}
# Checking if equivalent
print(x1==y1)
print(x1!=y1)

x2 = {1,2,3}
y2 = {4,5,6,7,8,9,10}
# Checking if equivalent
print(y2==x2)
```

```
print(x2!=y2)
```

Output

```
True
False
False
True
```

Difference between tuple, list, set, dictionary

Parameters	List	Tuple	Set	Dictionary
Definition	A list is an ordered, mutable collection of elements.	A tuple is an ordered, immutable collection of elements.	A set is an unordered collection of unique elements.	A dictionary is an unordered collection of key-value pairs.
Syntax	Syntax includes square brackets [,] with ',' separated data.	Syntax includes curved brackets (,) with ',' separated data.	Syntax includes curly brackets {,} with ',' separated data.	Syntax includes curly brackets {,} with ',' separated key-value data.
Creation	A list can be created using the list() function or simple assignment to [].	Tuple can be created using the tuple() function.	A set dictionary can be created using the set() function.	A dictionary can be created using the dict() function.
Empty Data Structure	An empty list can be created by l = [].	An empty tuple can be created by t = ().	An empty set can be created by s = set().	An empty dictionary can be created by {}.
Order	It is an ordered collection of data.	It is also an ordered collection of data.	It is an unordered collection of data.	Ordered collection in Python version 3.7, unordered in Python Version=3.6.
Duplicate Data	Duplicate data entry is allowed in a List.	Duplicate data entry is allowed in a Tuple.	All elements are unique in a Set.	Keys are unique, but two different keys CAN have the same value.
Indexing	Has integer based indexing that starts from '0'.	Also has integer based indexing that starts from '0'.	Does NOT have an index based mechanism.	Has a Key based indexing i.e. keys identify the value.
Addition	New items can be added using the append() method.	Being immutable, new data cannot be added to it.	The add() method adds an element to a set.	update() method updates specific key-value pair.
Deletion	pop() method allows deleting an element.	Being immutable, no data can be popped/deleted.	Elements can be randomly deleted using pop().	pop(key) removes specified key along with its value.
Sorting	sort() method sorts the elements.	Immutable, so sorting method is not applicable.	Unordered, so sorting is not advised.	Keys are sorted by using the sorted() method.
Search	index() returns index of first occurrence.	index() returns index of first occurrence.	Unordered, so searching is not applicable.	get(key) returns value against specified key.
Reversing	reverse() method reverses the list.	Immutable, so reverse method is not applicable.	Unordered, so reverse is not advised.	No integer-based indexing, so no reversal.
Count	count() method returns occurrence count.	count() method returns occurrence count.	count() not defined for sets.	count() not defined for dictionaries.