Advanced Computer Programming

Er. Mukesh Kumar Pokhrel

August 3, 2024

Unit-4:Advanced Data Types and Operation in Python (8 Hrs.)

- 4.1. Mutable and immutable data types
- 4.2. List and tuple data types
- 4.3. Dictionary data types
- 4.4. Sequence data types
- 4.5 Two-dimensional list
- 4.6 Set data types
- 4.7 Lambda
- 4.8 Operation of mutable and immutable data types

4.1 Mutable and Immutable data types

Mutable datatypes are those whose values can be changed in place after they are created. Examples of mutable datatypes in Python include:

- List
  - Lists are ordered collections of items.
  - Items can be added, removed, or changed
  - my_list = [1, 2, 3]
    my_list.append(4) # Adds 4 to the list
    my_list[0] = 0 # Changes the first item to 0
- Dictionaries:
  - Dictionaries are collections of key-value pairs.
  - Entries can be added, removed, or changed
  - my_dict = 'a': 1, 'b': 2
    my_dict['c'] = 3 # Adds a new key-value pair
    my_dict['a'] = 10 # Changes the value for key 'a'

- Sets are unordered collections of unique items.
- Items can be added or removed.
- my_set = 1, 2, 3
  my_set.add(4) # Adds 4 to the set
  my_set.remove(2) # Removes 2 from the set

Immutable Datatypes: Immutable datatypes are those whose values cannot be changed after they are created. Any operation that appears to modify the value actually creates a new object. Examples of immutable datatypes in Python include:

- Number
- String
- Tuples:Tuples are ordered collections of items, similar to lists, but they are immutable.

4.2.1 List

- Creation of list
- Creation of list using range() function
- Updating the elements of a list
- Concatenation of list
- Repetition of Lists
- Membership in Lists
- Aliasing and Cloning lists
- Methods to process Lists
- Finding Biggest and Smallest Elements in a list
- sorting the list elements
- Finding Common Elements in Two Lists

Introduction

A list is a collection of items that are ordered, mutable (changeable), and allows duplicate elements. Lists are one of the most versatile data types in Python, and they are used to store multiple items in a single variable. Lists can contain elements of different data types, including other lists.

Creation of list:

- Empty List:

```
empty_list = []
print(empty_list)
# Output: []
```

- List with elements:

```
# List of integers
int_list = [1, 2, 3, 4, 5]
print(int_list)
# Output: [1, 2, 3, 4, 5]

# List of strings
str_list = ["apple", "banana", "cherry"]
print(str_list)
# Output: ["apple", "banana", "cherry"]

# List with mixed data types
mixed_list = [1, "Hello", 3.14, True]
print(mixed_list)
# Output: [1, "Hello", 3.14, True]
```

```
#Nested list
nested_list = [1, [2, 3], [4, 5]]
print(nested_list)
# Output: [1, [2, 3], [4, 5]]
```

- Creation of list using range() function: We can use range() function to generate a sequence of integers which can be stored in a list.The format of range() function is:

```
range(start, stop, stepsize)
```

- if we do not mention start ,it is assumed to be 0 and the 'stepsize' is taken as 1.

```
#Example 1
list1=list(range(11))
print(list1)
#Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#Example 2
list2=list(range(1,11,2))
print(list2)
#Output:
[1, 3, 5, 7, 9]
```

Accessing the element of List

- Accessing by Index:

```
#Creating a list
my_list = [10, 20, 30, 40, 50]

# Accessing elements
print(my_list[0])
print(my_list[2])
print(my_list[4])
#Output:
10
30
50
```

- Negative Indexing:

```
# Accessing elements with negative indexing
print(my_list[-1])
#Output: 50
#Note -1 indexing means last element of list
print(my_list[-2])
#Output: 40
print(my_list[-5])
#Output: 10
```

- **Slicing:**Slicing allows you to access a range of elements by specifying a start index, an end index, and an optional step.
  syntax:list-name[start:stop:step_size]

```
# Creating a list
my_list = [10, 20, 30, 40, 50]
# Slicing the list

print(my_list[1:4])  # Output: [20, 30, 40]
print(my_list[:3])   # Output: [10, 20, 30]
print(my_list[2:])   # Output: [30, 40, 50]
print(my_list[::2])  # Output: [10, 30, 50]
print(my_list[1:5:2])# Output: [20, 40]
print(my_list[::-1] #Output:[50,40,30,20,10](reverse of
list using slicing method)
```

- Iterating Through a List:

```
#Iterating through a list
for element in my_list:
    print(element,end=' ')

#Output: 10 20 30 40 50
```

- Using the enumerate Function:

```
my_list=[10,20,30,40,50]
# Using enumerate to get index and value
for index, value in enumerate(my_list):
    print(f"Index: {index}, Value: {value}")

#Output:
Index: 0, Value: 10
Index: 1, Value: 20
Index: 2, Value: 30
Index: 3, Value: 40
Index: 4, Value: 50
```

Updating the element of list

List are mutable.It means we can modify the contents of a list.We can append,update or delete the elements of a list depending upon our requirements.

- append(): add new items to the list in the last position.

```
lst=[1,2,3,4]
lst.append(5)
print(lst)
#output:[1,2,3,4,5]
```

- Update the first element of list:

```
lst[1]=0
print(lst)
# Output:[1,0,3,4,5]
```

- Update the first and second elements of list:

```
lst[1:3]=10,11
print[lst]
#Output:[0,10,11,4,5]
```

- delete 11 from the list:

```
lst.remove(11)
print(lst)
#Output:[0,10,4,5]
```

- delete the element using del statement

```
del lst[0]
print(lst)
#Output: [10,4,5]
```

- retrieve the element of list in reverse order.

```
lst.reverse()
print(lst)
#Output:
[5,4,10]
```

Concatenation of two list

- Concatenate using + operator

```
#Example 1
list1=[1,2,3,4,5]
list2=[5,4,3,2,1]
list3=list1+list2
list3
#Output: [1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
#Example 2
list1=[1,2,3,4,5]
list2=[5,4,3,2,1]
list3=list2+list1
list3
#Output:
[5, 4, 3, 2, 1, 1, 2, 3, 4, 5]
```

- Concatenate using extend() method: insert one list into another.

```
list1=[1,2,3,4,5]
list2=[5,4,3,2,1]
list1.extend(list2)
print(list1)
#Output:
[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
This program use extend() function to insert list2 into
list1 which shows the example of concatenate of two list.
```

- list comprehension method to concatenate two list:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3=[j for i in [list1,list2] for j in i]
list3
#Output:[1, 2, 3, 4, 5, 6]
```

Repetition of Lists

We can repeat the elements of a list 'n' number of times using '*' operator.

```
list=[1,2,3]
print(list*2)
#Output:[1,2,3,1,2,3]
```

Membership in Lists

We can check if an element is a member of list or not by using 'in' and 'not in' operator.If the element is a member of the list,then 'in' operator returns True else False.

- 'in' operator

```
x=[10,20,30,40,50]
i=20
print(i in x)

#Output:
True
```

- 'not in' operator

```
print(i not in x)

# Output: False
```

Aliasing and Cloning

- Aliasing: Giving a new name to an existing list is called 'aliasing'.The new name is called 'alias name'.For example:

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

To provide the new name to this list, we can use assignment operator as:

```
y=x
```

In this case, modifications on x will also modify y and vice versa.

```
x=[10,20,30,40,50]
y=x # x is aliased as y
x[1]=99
print(x)
print(y)
#Output:
[10,99,30,40,50]
[10,99,30,40,50]
```

This program shows that any modication i.e replacing 20 with 99 on x list i.e original list will also modify y list.

- Cloning: Obtaining exact copy of an existing object(or list) is called cloning.To clone a list,we can take help of the slicing operations as:

```
y=x[:]
```

- When we clone a list like this, a separate copy of all elements is stored into 'y'.The list 'x' and 'y' are independent lists.

```
x=[10,20,30,40,50]
    y=x[:] # x is aliased as y
    x[1]=99
    print(x)
    print(y)
    #Output:
    [10,99,30,40,50]
    [10,20,30,40,50]
```

also able to use :

```
y=x.copy()
```

Method of list

- if list1 is the name of list then,
- len()- len(list1)- to find the number of elements in a list.
- sum()-list1.sum()- returns the sum of all elements in the list.
- index()-list1.index(x)-returns the position of element in the list.
- append()- list1.append(x)-add new element in the last of list.
- insert()- list1.insert(i,x)-insert x into the list1 in the position specified by i
- copy()- list1.copy()- copies all the list elements into a new list and returns it.
- extend()-list1.extend(list2)- insert list2 into list1
- count()-list1.count(x)-count the occurances of element.
- remove()-list1.remove(x)-remove element(x) from the list.
- pop()- list1.pop()-remove the last element from the list.
- sort()-list1.sort()-sorts the elements of the list into ascending order.
- reverse()-list1.reverse()-Reverse the sequence of elements in the list
- clear()-list1.clear()-delete all the elements from the list.

Finding Biggest and smallest Elemenets in a List

- max()- to find the biggest element of the list.
- min()- to find the smallest element of the list.

```
list1=[10,20,30,40,50]
n1=min(list1)
n2=max(list1)
print(n1)
print(n2)
#Output:
10
50
```

Sorting the list element

- to sort in ascending order:

```
list1=[20,10,60,80,50]
list1.sort()
print(list1)

#Output:
[10, 20, 50, 60, 80]
```

- to sort in descending order:

```
list1=[20,10,60,80,50]
list1.sort(reverse=True)
print(list1)
#Output:
[80, 60, 50, 20, 10]
```

Exercise

QN. Write a program to find the common element of two list.

4.2.2 Tuple

- Creating tuple
- Accessing the tuple element
- Basic operation on Tuples
- Function to process tuple
- Nested Tuples
- Sorting Nested tuples
- Inserting Elements in tuple
- Modifying elements of a tuple
- Deleting Elements of tuple
- Tuple Unpacking

Tuple Introduction

- Tuples are immutable, meaning their elements cannot be changed once assigned.
- They are defined by parentheses () or without them, separated by commas.
- A tuple is an ordered collection of items that are immutable in Python.

Creation of Tuple

- We can create a tuple by writing elements separated by commas inside parentheses() or without small parentheses separating element only by commas.
- **Empty tuple**
  To create an empty tuple, we can simply write empty parenthesis,as:
  tup1=()
- **Tuple with one element**
  - tup2=(10,) # note commas
- **Tuple with multiple element**
  - tup3=(10,20,30,'NEPAL','KHWOPA','MUKESH')
- **tuple with no small parenthesis()** If we do not write any brackets and write elements by separating commas then they are taken as by default as tuple.
  - tup4=1,2,3,4 # no braces

Accessing the tuple elements

```
#Accessing the element by index
my_tuple = (10, 20, 30, 40, 50)
# Access the first element
first_element = my_tuple[0]  # 10
# Access the second element
second_element = my_tuple[1]  # 20
# Access the last element
last_element = my_tuple[-1]  # 50
# Access the second-to-last element
second_last_element = my_tuple[-2]  # 40
```

```
#Accessing element by slicing
# Define a tuple
my_tuple = (10, 20, 30, 40, 50)
# Slice from the second element to the fourth
element (index 1 to 3)
slice_tuple = my_tuple[1:4]
#Tuple Unpacking: we can unpack the elements of a
tuple into individual variables.
# Define a tuple
person_info = ("Alice", 25, "Data Scientist")
# Unpack the tuple
name, age, profession = person_info
```

```
print(name)
print(age)
print(profession)

#Output:
Alice
25
Data Scientist
```

Basic operation on Tuples

- **To concatenate the two tuple using + operator**

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

- **Repetition**:We can repeat the elements in a tuple using the * operator.

```
tuple1 = (1, 2, 3)
repeated_tuple = tuple1 * 3
print(repeated_tuple)
# Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

- **Membership Testing**:We can check if an element exists in a tuple using the in and not in operators.

```
tuple1 = (1, 2, 3, 4, 5)
print(3 in tuple1)     # Output: True
print(6 not in tuple1) # Output: True
```

Functions in tuple

- **len()**
- **min()**
- **max()**
- **index()**
- **max()**
- **sorted()**

- **len()** Returns the number of elements in a tuple.

```
tuple1 = (1, 2, 3, 4, 5)
print(len(tuple1))
# output: 5
```

- **min()**-Returns the smallest element in a tuple. The elements must be comparable (i.e., all elements must be of the same type).

```
tuple1 = (1, 2, 3, 4, 5)
print(min(tuple1))
# output 1
```

- **max()**-Returns the largest element in a tuple. The elements must be comparable.

```
tuple1 = (1, 2, 3, 4, 5)
print(max(tuple1))
# output: 5
```

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

```
tuple1 = (1, 2, 2, 3, 2, 4, 5)
print(tuple1.count(2))
# output 3
```

- **index()**-Returns the index of the first occurrence of a specified value. Raises a ValueError if the value is not found.

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1.index(3))
#output 2
```

- **sorted()**-Returns a sorted list of the elements in the tuple. The original tuple remains unchanged.

```
tuple1 = (5, 3, 1, 4, 2)
sorted_tuple = sorted(tuple1)
print(sorted_tuple)
# output [1, 2, 3, 4, 5]
type(sorted_tuple)
# output:list
```

Nested Tuples

Nested tuples in Python refer to tuples that contain other tuples as their elements.

Example 1: Nested tuples

```
nested_tuple = ((1, 2), (3, 4), (5, 6))
print(nested_tuple)
```

Accessing Elements in Nested Tuples Accessing elements in nested tuples involves using indexing multiple times to reach the desired element.

```
# Accessing elements in nested tuples
nested_tuple = ((1, 2), (3, 4), (5, 6))
# Accessing the first tuple within nested_tuple
first_inner_tuple = nested_tuple[0]
print(first_inner_tuple) # Output: (1, 2)
# Accessing elements within the first tuple
first_element = nested_tuple[0][0]
second_element = nested_tuple[0][1]
print(first_element) # Output: 1
print(second_element) # Output: 2
```

Sorting Nested Tuples

```
emp=((10,"Vijay",9000.90),
(20,"Mukesh",5500.75),(30,"Khwopa",8900.00))
print(sorted(emp))
#output:[(10, 'Vijay', 9000.9),
(20, 'Mukesh', 5500.75), (30, 'Khwopa', 8900.0)]
```

The result shows that sorting nested tuple in ascending order based on zero position of the nested tuples

```
print(sorted(emp,key=lambda x:x[1]))
#output: [(30, 'Khwopa', 8900.0),
(20, 'Mukesh', 5500.75), (10, 'Vijay', 9000.9)]
```

The result shows that sorted based on the first position of nested element in ascending order.

Lambda Function Example

A lambda function in Python is a small anonymous function defined with the ‘lambda‘ keyword. Here’s an example that squares a number:

```
# Example of a lambda function
square = lambda x: x ** 2

# Using the lambda function
result = square(5)
print("Square of 5 is:", result)
# Output: Square of 5 is: 25
```

Lambda functions are often used when a simple function is needed for a short period of time, typically as an argument to higher-order functions or in places where a named function would be cumbersome.

Inserting Element in tuple

Tuples in Python are immutable, meaning once they are created, their elements cannot be changed, added, or removed directly. However, we can create a new tuple that includes the elements of the original tuple along with the new element.

```
# Original tuple
 my_tuple=(1,2,3,4,5)
 # First copy the elements of original tuple into new one
 new_tuple=my_tuple
 #new_tuple=my_tuple[0:]#alernate approach of copy
 into new_tuple
 #Then add new element using + operator
 new_tuple=new_tuple+(6,)#note commas for one elements
 print(new_tuple)
 #output: (1,2,3,4,5,6)
```

Modifying Element of Tuple

It is not possible to modify or update an element of a tuple since tuples are immutable. If we want to modify the element of a tuple, we have to create a new tuple with the new value in place of the modified element.

```
#modifying an existing element of a tuple
num=(10,20,30,40,50)
#accept the new number and position number
a=eval(input('enter the element:'))
pos=int(input('enter the position number:'))
new_num=num[:pos]+(a,)+num[pos:]
print(new_num)
#output
enter the element:45
enter the position number:4
(10, 20, 30, 40, 45, 50)
```

if we want to replace the new value in place of existing element inside tuple:

```
# Original tuple
num = (10, 20, 30, 40, 50)
# Input the new number and position number
a = eval(input('Enter the element: '))
pos = int(input('Enter the position number: '))
# Create a new tuple with the modified element
new_num = num[:pos] + (a,) + num[pos+1:]
print(new_num)
Enter the element: 45
Enter the position number: 4
(10, 20, 30, 40, 45)
```

Note: (a,) is used to make single value input into tuple

Deleting the elements of tuple

In Python, tuples are immutable, which means you cannot delete elements from a tuple directly. However, we can create a new tuple that excludes the elements we want to "delete."

```
# Original tuple
num = (10, 20, 30, 40, 50)

# Index of the element to delete
# Deleting the element at index 2 (value 30)
index = 2

# Create a new tuple excluding the element at index
new_num = num[:index] + num[index + 1:]
print(new_num)
#output: (10, 20, 40, 50)
```

Tuple Unpacking

Tuple unpacking in Python refers to a technique that allows us to assign the elements of a tuple to individual variables in a single statement.Let's see a practical example of tuple unpacking:

```
# Define a tuple
person_info = ("Alice", 30, "New York")
# Unpack the tuple into variables
name, age, city = person_info
# Print the unpacked variables
print("Name:", name)
print("Age:", age)
print("City:", city)
#Output:
Name: Alice
Age: 30
City: New York
```

Questions For Assignment

- Write a program to swap the two tuples in python.
- Given is a nested tuple. Write a program to modify the first item (22) of a list inside a following tuple to 222.

```
Given:
tuple1 = (11, [22, 33], 44, 55)
Expected:
tuple1: (11, [222, 33], 44, 55)
```

- Write a program to Sort a tuple of tuples by 2nd item.
- Write a program Check if all items in the tuple are the same
- Write a Python program to unpack a tuple into several variables.
- Write a Python program to add an item to a tuple.
- Write a Python program to convert a tuple to a string.
- Write a Python program to create the colon of a tuple.
- Write a Python program to find repeated items in a tuple.
- Write a Python program to convert a list to a tuple.
- Write a Python program to convert a tuple to a dictionary.
- . Write a Python program to unzip a list of tuples into individual lists.
- Write a Python program to convert a list of tuples into a dictionary.
- Write a Python program to print a tuple with string formatting. Sample tuple : (100, 200, 300) Output : This is a tuple (100, 200, 300)

- Write a Python program to compute the element-wise sum of given tuples. Original lists: (1, 2, 3, 4) (3, 5, 2, 1) (2, 2, 3, 1) Element-wise sum of the said tuples: (6, 9, 8, 6)
- Write a Python program to compute the sum of all the elements of each tuple stored inside a list of tuples. Original list of tuples: [(1, 2), (2, 3), (3, 4)] Sum of all the elements of each tuple stored inside the said list of tuples: [3, 5, 7] Original list of tuples: [(1, 2, 6), (2, 3, -6), (3, 4), (2, 2, 2, 2)] Sum of all the elements of each tuple stored inside the said list of tuples: [9, -1, 7, 8]
- Write a Python program to compute the sum of all the elements of each tuple stored inside a list of tuples. Original list of tuples: [(1, 2), (2, 3), (3, 4)] Sum of all the elements of each tuple stored inside the said list of tuples: [3, 5, 7] Original list of tuples: [(1, 2, 6), (2, 3, -6), (3, 4), (2, 2, 2, 2)] Sum of all the elements of each tuple stored inside the said list of tuples: [9, -1, 7, 8]
- Write a Python program to convert a given list of tuples to a list of lists. Original list of tuples: [(1, 2), (2, 3), (3, 4)] Convert the said list of tuples to a list of lists: [[1, 2], [2, 3], [3, 4]] Original list of tuples: [(1, 2), (2, 3, 5), (3, 4), (2, 3, 4, 2)] Convert the said list of tuples to a list of lists: [[1, 2], [2, 3, 5], [3, 4], [2, 3, 4, 2]]

4.3 Dictionary Data Types

- Introduction
- Creation of Dictionary
- Accessing the element of Dictionary
- Operation on Dictionaries
- Dictionary Method
- Using for loop with Dictionaries
- Sorting the Elements of Dictionary Using Lambdas
- Converting List into Dictionary
- Concerting String into Dictionary
- Passing Dictionaries to functions
- Ordered Dictionaries

Introduction of Dictionary

A dictionary in Python is a powerful and versatile data structure used to store collections of data in key-value pairs.All the key-value pairs in the dictionary are inserted in the curly braces .

Features of Dictionary:

- **Key-Value Pairs:**Dictionaries consist of key-value pairs enclosed in curly braces . Each key is separated from its value by a colon :.
- **Unordered:**Dictionaries are unordered collections. The order in which elements are stored is not guaranteed. The elements in a dictionary are not accessible by their index and their order is not preserved.
- **Mutable:**Dictionaries can be modified after creation. You can add new key-value pairs, delete existing ones, or modify the values associated with existing keys.

Creation of Dictionary

The most common and simplest way to create a dictionary is by enclosing commaseparated key-value pairs inside curly braces .
Example:

```
#Creating a dictionary of student names and their ages
student_dict = {'Alice': 20, 'Bob': 21, 'Charlie': 19}
print(student_dict)
#output:
{'Alice': 20, 'Bob': 21, 'Charlie': 19}
```

Accessing the element of Dictionary

Accessing elements in a dictionary in Python involves retrieving the value associated with a specific key.To access the elements of a dictionary, we should not use indexing or slicing.

- Using square bracket [] notation: we can access a value in a dictionary by specifying the key inside square brackets[].

```
# Creating a dictionary
student_dict = {'Alice': 20, 'Bob': 21, 'Charlie': 19}

# Accessing elements using keys
print(student_dict['Alice'])  # Output: 20
print(student_dict['Bob'])    # Output: 21
```

- Using get method

```
#Creating a dictionary
student_dict = {'Alice': 20, 'Bob': 21, 'Charlie': 19}
# Accessing elements using get() method
print(student_dict.get('Charlie'))
#Output: 19
```

Tutorial

QN. Write a program to create a dictionary with employee details and retrieve the values upon giving the keys.

Operations on Dictionaries

- To know how many key-value pair are there in a dictionary, we can use **len()** function.

```
# Creating a dictionary
student_dict = {'Alice': 20, 'Bob': 21, 'Charlie': 19}
# Finding the length of the dictionary
length = len(student_dict)
print(f'The length of the dictionary is: {length}')
#Output: The length of the dictionary is: 3
```

- We can also insert key-value pair into an existing dictionary.

```
    student_dict['David']= 22
    print(student_dict)
    #Output: {'Alice': 20, 'Bob': 21, 'Charlie': 19,
'David': 22}
```

Operation on Dictionaries

- We can modify the existing value of key by assigning a new value.

```
student_dict['David']=25
print(student_dict)
#output:{'Alice': 20, 'Bob': 21, 'Charlie': 19,
'David': 25}
```

- We can delete a key-value pair from Dictionary. del keyword delete the key-value pair.
  Example:

```
del student_dict['David']
print(student_dict)
#Output: {'Alice': 20, 'Bob': 21, 'Charlie': 19}
```

- To test whether a 'Key' is available in dictionary or not. We can use 'in' or 'not in' operators.

```
'David' not in student_dict
#Output: True
'Alice' in student_dict
#Output: True
```

- **QN. Write the properties of keys used in Dictionaries.**
  - Key should be unique. It means Dublicates Key are not allowed.
  - If we enter same key, the old key will be overwritten and only the new key will be available.

```
student={'Alice':20,'Bob':21,'Alice':25}
print(student)
#Output:{'Alice': 25, 'Bob': 21}
```

  - Keys should be immutable type.For example, We can use a number, string or tuples as key since they are immutable.List are not allowed as key since it is mutable.

Dictionary method

- clear(): Removes all the key value pair from the dictionary.
- copy() : Copies all the element from original dictionary into new one .
- fromkeys():The fromkeys() method in Python is used to create a new dictionary from a given iterable (like a list or tuple).
  Example:

```
# Example 1: Creating a dictionary from a list of keys
keys = ['a', 'b', 'c', 'd']
# Using dict.fromkeys()
my_dict = dict.fromkeys(keys)
print(my_dict)
#Output: {'a': None, 'b': None, 'c': None, 'd': None}
```

```
Example 2:
my_dict = dict.fromkeys(keys,'Unknown')
print(my_dict)
#Output:
{'a': 'Unknown', 'b': 'Unknown', 'c': 'Unknown',
'd': 'Unknown'}
```

Dictionary method

- get()-Returns the value associated with keys.
- items()-Returns an object that contain key-value pair.the pairs are stored as tuple in the object.

```
my_dict.items()
#Output:
dict_items([('a', 'Unknown'), ('b', 'Unknown'),
('c', 'Unknown'), ('d', 'Unknown')])
```

- keys(): Return a sequence of key from the dictionary.

```
# Creating a dictionary
student_dict = {'Alice': 20, 'Bob': 21, 'Charlie': 19}

student_dict.keys()
#Output:dict_keys(['Alice', 'Bob', 'Charlie'])
```

- values(): return a sequence of values from the dictionary.
- update(): Add all the element from one dictionary to other.

```
# Creating two dictionaries
dict1 = {'Alice': 20, 'Bob': 21}
dict2 = {'Charlie': 19, 'David': 22}
# Using update() to add elements from dict2 to dict1
dict1.update(dict2)
```

Dictionary Method

```
# Printing the updated dict1
print(dict1)
#Output: {'Alice': 20, 'Bob': 21, 'Charlie': 19,
'David': 22}
```

- pop()-The pop() method in Python dictionaries is used to remove a specified key and return the corresponding value. If the key is not found, it raises a **KeyError** unless a default value is provided. This method is useful when you want to remove an element from a dictionary and also retrieve its value.

```
# Creating a dictionary
student_ages = {'Alice': 20, 'Bob': 21, 'Charlie': 19}

# Removing the key 'Bob' and returning its value
age_of_bob = student_ages.pop('Bob')


# Printing the value and the updated dictionary
print("Age of Bob:", age_of_bob)
print("Updated dictionary:", student_ages)
#Output:
Age of Bob: 21
Updated dictionary: {'Alice': 20, 'Charlie': 19}
```

Dictionary Method

- setdefault()-The setdefault() method in Python dictionaries is used to retrieve the value of a specified key. If the key does not exist in the dictionary, the method inserts the key with a specified default value and returns the default value.

```
# Creating a dictionary
student_ages = {'Alice': 20, 'Bob': 21}
# Using setdefault() to get the value of an existing key
age_of_Mukesh = student_ages.setdefault('Mukesh', 18)
age_of_alice= student_ages.setdefault('Alice',18)
# Printing the value and the dictionary
print("Age of Alice:", age_of_alice)
print("Age of Mukesh:", age_of_Mukesh)
print("Updated dictionary:", student_ages)
```

```
#Output:
Age of Alice: 20
Age of Mukesh: 18
Updated dictionary:{'Alice': 20, 'Bob': 21, 'Mukesh': 18}
```

Exercise

- QN1. Write a program to retreive keys, values and key-value pairs from a dictionary.

```
# Define a sample dictionary
sample_dict = {
    'name': 'Alice',
    'age': 25,
    'profession': 'Data Scientist',
    'location': 'New York'
}

# Retrieve keys
keys = sample_dict.keys()
print("Keys:")
for key in keys:
    print(key)

# Retrieve values
values = sample_dict.values()
print("\nValues:")
for value in values:
    print(value)
```

Exercise

```
# Retrieve key-value pairs
items = sample_dict.items()
print("\nKey-Value Pairs:")
for key, value in items:
    print(f"{key}: {value}")


#Output:
Keys:
name
age
profession
location
Values:
Alice
25
Data Scientist
New York
Key-Value Pairs:
name: Alice
age: 25
profession: Data Scientist
location: New York
```

Exercise

- QN2.Write a program to create a dictionary and find the sum of values.

```
#program to find the sum of values in the dictionary
#enter the dictionary entries from keyboard
dict=eval(input('Enter the elements in {}:'))
#find the sum of values
s=sum(dict.values())
print(s)
#Output:
Enter the elements in {}:{'Mukesh':50,'Inogen':60,
'Kabi':90}
200
```

Exercise

- QN3.Write a python program to create a dictionary from keyboard and display the elements.

```
#creating a dictionary from the keyboard
x={} #take a empty dictionary
print('How many elements',end=' ')
n=int(input())# n indicate n number of key-value pair
for i in range(n):
  print('Enter the key:',end=' ')
  key=input()# key is in string form
  print('Enter its value',end=' ')
  value=int(input())
  x.update({key:value})
print(x)


#Output:
How many elements 2
Enter the key: Mukesh
Enter its value 40
Enter the key: Sunil
Enter its value 30
{'Mukesh': 40, 'Sunil': 30}
```

Class work

- Write a program to create a dictionary with cricket players name of Nepal and scores in match.Also retrieve run by entering player name.

Using for loops in Dictionaries

Using a for loop with dictionaries in Python is a common way to iterate over the elements in the dictionary. A dictionary in Python is a collection of key-value pairs, and we can loop through these pairs in several ways.

- Looping through Keys

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key in my_dict:
    print(key,end=' ')
#Output:a b c
```

- Looping through values

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
for value in my_dict.values():
    print(value,end=' ')
#Output:1 2 3
```

Exercise

- Q.Write a program to show the usage of for loop to retrieve elements of dictionaries.

```
#Using for loop with dictionaries
colors={'r': 'Red','g':'Green','b':'Blue'}
#Display only keys
for k in colors:
    print(k)
#pass keys to dictionary and display the values
for k in colors:
    print(colors[k])
#items() methods returns key and value pair into k,v
for k,v in colors.items():
    print('Key={} Value={}'.format(k,v))
#Output:
r
g
b
Red
Green
Blue
Key=r Value=Red
Key=g Value=Green
Key=b Value=Blue
```

dict.get(x,0)+1

```
dict={}
str="Book"
for i in str:
  dict[i]=dict.get(i,0)+1
print(dict)
#output: {'B': 1, 'o': 2, 'k': 1}
```

Q.Write a program to find the number of occurrences of each letter in a string using dictionary.

Sorting the elements of Dictionary using lambdas

A lambda is a function that doesnot have a name. Lambda functions are written without using def keyword.It is written using single statement and look like expressions.
Sorting the elements of a dictionary using a lambda function can be done in several ways, depending on whether you want to sort by keys ,by values or by length of keys etc.

```
my_dict = {'c': 3, 'a': 1, 'b': 2}

# Sort dictionary by keys in ascending order
sorted_dict_keys = sorted(my_dict.items(), key=lambda x: x[0])
print("Sorted dictionary by keys(ascending):",sorted_dict_keys)

# Sort dictionary by keys in descending order
sorted_dict_keys_desc =sorted(my_dict.items(),
key=lambda x: x[0], reverse=True)
print("Sorted dictionary by keys
(descending):",sorted_dict_keys_desc)

#Output:
Sorted dictionary by keys (ascending):
[('a', 1), ('b', 2), ('c', 3)]
Sorted dictionary by keys (descending):
[('c', 3), ('b', 2), ('a', 1)]
```

Converting List Into Dictionary

When we have two list, it is possible to convert them into a dictionary.For example,we have two lists containing name of countries and name of their capital cities.

```
Countries=['USA','India','Germany','France']
Capitals=['Washington','New Delhi','Berlin','Paris']
#First step: to create zip object by
passing two list in zip() function
z=zip(Countries,Capitals)
d=dict(z)
print(d)
#Output:
{'USA': 'Washington', 'India': 'New Delhi',
'Germany': 'Berlin', 'France': 'Paris'}

#single Statement
z=dict(zip(Countries,Capitals))
print(z)
```

Exercise

Qn. Why zip object are exhausted? what does this mean?
When we say that a zip object (or any iterator in Python) is "exhausted," it means that the iterator has been completely iterated over, and there are no more elements left to retrieve. This is a fundamental property of iterators in Python.Let illustrate with one example:

```
# Create two lists
Countries=['USA','India','Germany','France']
Capitals=['Washington','New Delhi','Berlin','Paris']
# Create a zip object
zipped = zip(Countries,Capitals)
for i in zipped:
  print(i)
# Attempt to convert the zip object to a dictionary
after it has been exhausted
my_dict = dict(zipped)
print("Dictionary:", my_dict)

#Output:
('USA', 'Washington')
('India', 'New Delhi')
('Germany', 'Berlin')
('France', 'Paris')
Dictionary: {}
```

After the for loop completes, the zip object zipped is exhausted. This means that all the pairs have been generated and the iterator has no more elements to produce.Then attempt to convert the zip object to a dictionary. However, since the zip object is already exhausted from the previous iteration, there are no elements left to retrieve. As a result, dict(zipped) creates an empty dictionary. Solution of above problem

```
# Create two lists
Countries=['USA','India','Germany','France']
Capitals=['Washington','New Delhi','Berlin','Paris']
# Create a zip object
zipped =list(zip(Countries,Capitals))
for i in zipped:
  print(i)
# Attempt to convert the zip object to a
dictionary after it has been exhausted
my_dict = dict(zipped)
print("Dictionary:", my_dict)
#Output
('USA', 'Washington')
('India', 'New Delhi')
('Germany', 'Berlin')
('France', 'Paris')
Dictionary: {'USA': 'Washington', 'India': 'New Delhi',
'Germany': 'Berlin', 'France': 'Paris'}
```

Converting Strings into Dictionary

When a string is given with key and value pairs separated by some delimiters(or separator) like a comma(,) we can convert the string into dictionary and use it as dictionary.Let's take an example:

```
str="Mukesh=23,Inogen=24,Kabi=20"
list1=[]
for i in str.split(','):
    y=x.split('=')
    list1.append(y)
print(list1)
#Output:
[['Mukesh', '23'], ['Inogen', '24'], ['Kabi', '20']]

d=dict(list1)
print(d)
#Output:
{'Mukesh': '23', 'Inogen': '24', 'Kabi': '20'}
#But this d have both name and age as a string.For
creating new dictionary d1 with name as string and
age as int.
d1={}
for k,v in d.items():
    d1[k]=int(v)
print(d1)
#Output:{'Mukesh': 23, 'Inogen': 24, 'Kabi': 20}
```

Passing Dictionary to function

We can pass a dictionary to a function by passing the name of the dictionary.

```
def fun(dict):
    for i,j in dict.items():
        print(i,' ',j)
d1={'Mukesh': '23', 'Inogen': '24', 'Kabi': '20'}
fun(d1)
#Output:
Mukesh   23
Inogen   24
Kabi   20
```

Ordered Dictionaries

The elements of the dictionary are not ordered.It means the elements are not stored into the same order as they were entered into the dictionary.
An ordered dictionary is a dictionary which will keep the order of elements.The element are stored and maintained in the same order as they were entered into the ordered dictionary.We can create an ordered dictionary using the ordered dictionary using OrderdDict() method of 'collections' module.

```
from collections import OrderedDict
# Creating an OrderedDict
ordered_dict = OrderedDict()
# Adding elements to the OrderedDict
ordered_dict['a'] = 1
ordered_dict['b'] = 2
ordered_dict['c'] = 3
print(ordered_dict)
# Iterating over the OrderedDict
for key, value in ordered_dict.items():
    print(key, value)

#Output:
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
a 1
b 2
c 3
```

4.4 Sequence Data types

Sequence data types in Python refer to collections that hold ordered items, allowing for indexing, slicing, and iteration over their elements. They are fundamental in storing and manipulating collections of objects in a predictable and ordered manner. The primary sequence data types in Python include lists, tuples, strings, and range objects.

Characteristics of Sequence Data Types:

- Ordered: Sequence data types maintain the order of elements as they are inserted or defined.
- Indexing and Slicing: Elements within sequences can be accessed using their position (index) within the sequence. Slicing allows for extracting subsets of elements based on start, stop, and step indices.
- Support for Iteration: Sequence types can be iterated over using loops (e.g., for loop) or iteration functions (enumerate(), zip()).
- Mutable Sequences: Lists are mutable, allowing elements to be modified, added, or removed after creation.
- Immutable Sequences: Tuples and strings are immutable, meaning their elements cannot be changed once the sequence is created.

4.5 Two dimensional list

Two-dimensional list is typically created by nesting lists within a main list. Each sublist represents a row of the matrix or grid, and elements within each sublist represent columns.

```
# Creating a 3x3 matrix as a two-dimensional list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
#Accessing element
# Accessing a specific element in the matrix
print(matrix[0][0])
# Output: 1
print(matrix[1][2])
# Output: 6
```

4.6 Set data types

- Introduction
- Creation of Set data types
- Basic Set operation
- Practical application of Set

Introduction

A set is an unordered collection of unique elements.In Python, sets are defined using curly braces or the set() constructor.

Key Characteristics of Set:

- **Unordered:**The elements in a set are not stored in any particular order. This means that when you print a set, the order of elements might differ from the order in which they were added.

```
my_set = {3, 1, 2}
print(my_set)
# Output could be {1, 2, 3} or {2, 3, 1} etc.
```

- **Unique Elements:**Sets do not allow duplicate elements. If we try to add a duplicate element, it will be ignored.

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

- **Mutable:**Sets are mutable, meaning you can add or remove elements after the set has been created.

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

- **No Indexing or Slicing:**Since sets are unordered, they do not support indexing, slicing, or other sequence-like behavior.

Creation of Set

- Sets can be created using curly braces with elements separated by commas.

```
#Creating a set using literals
my_set = {1, 2, 3, 4, 5}
print(my_set)
# Output: {1, 2, 3, 4, 5}
```

- Creating Sets Using the set() Constructor: The set() constructor is a built-in Python function that creates a set from an iterable (e.g., list, tuple, string).

```
#Creating a set from a list
my_list = [1, 2, 3, 4, 5]
my_set = set(my_list)
print(my_set)
#Output: {1, 2, 3, 4, 5}
```

- An empty set cannot be created using because it is interpreted as an empty dictionary. To create an empty set, you must use the set() constructor.

```
#Creating an empty set
empty_set = set()
print(empty_set)
#Output: set()
```

Operation of Set

- The add() method adds a single element to a set. If the element is already present, the set remains unchanged (no duplicates).

```
#Creating a set
my_set = {1, 2, 3}
print("Initial set:", my_set)  # Output: {1, 2, 3}

# Adding an element
my_set.add(4)
print("After adding 4:", my_set)  # Output: {1, 2, 3, 4}

# Adding a duplicate element
my_set.add(2)
print("After adding 2 again:", my_set)
# Output: {1, 2, 3, 4} (unchanged)
```

- remove(): Removes a specified element from the set. Raises a KeyError if the element is not found.
- discard(): Removes a specified element from the set. Does not raise an error if the element is not found.

Frame Title

- The pop() method removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
# Popping an element
popped_element = my_set.pop()
print("Popped element:", popped_element)
print("Set after popping:", my_set)
# Popping all elements one by one
while my_set:
    print("Popped element:", my_set.pop())
print("Set after popping all elements:", my_set)
# Output: set() (empty set)
```

- The clear() method removes all elements from the set, leaving it empty.

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
print("Initial set:", my_set)  # Output: {1, 2, 3, 4, 5}
# Clearing the set
my_set.clear()
print("Set after clearing:", my_set)
# Output: set()
```

Accessing Set Elements

- Loop through Set Elements Using a for Loop: To access each element in a set, we can use a for loop. This method iterates over the set, processing each element one by one.

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
# Looping through the set elements using a for loop
print("Set elements:")
for element in my_set:
    print(element)
```

- Check if an Element Exists in a Set Using in and not in:To check if a specific element is present in a set, you can use the in operator. To check if an element is not present, use the not in operator.

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
# Checking if an element exists in the set using 'in'
print(3 in my_set)  # Output: True
# Checking if an element does not exist in the
set using 'not in'
print(3 not in my_set)  # Output: False
```

- To acess the specified element of set , first convert to ordered type data then use indexing method to access the specified element.For example, first convert the set into list then use index method to access.

Set Comprehension

Set comprehensions are a concise way to create sets in Python. They are similar to list comprehensions but use curly braces instead of square brackets []. Set comprehensions allow you to generate sets in a readable and efficient manner, especially when applying conditions or transformations to elements from other iterables.

```
#syntax:{expression for item in iterable if condition}
expression: The value to add to the set.
item: The variable representing each element in the iterable.
iterable: The collection of elements to iterate over.
condition: (Optional) A condition to filter elements.
#Example 1
# Using set comprehension to create a set of squares

squares_set = {x**2 for x in range(10)}
print(squares_set)

# Output: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

#Example 2
multiples_of_three = {x for x in range(31) if x % 3 == 0}
print(multiples_of_three)
```

Advanced set operation

- | Operator:

```
# Creating two sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}
# Union of set1 and set2 using the | operator
union_set = set1 | set2
print("Union (|):", union_set)
#Union (|): {1, 2, 3, 4, 5}
```

- union() method:

```
# Union of set1 and set2 using the union() method
union_set_method = set1.union(set2)
print(union_set_method)
# Output: {1, 2, 3, 4, 5}
```

- operator:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
# Intersection of set1 and set2 using the & operator
intersection_set = set1 & set2
print(intersection_set)
# Output: {3}
```

Advanced set Operation

- Using intersection() method

```
# Intersection of set1 and set2 using
the intersection() method
intersection_set_method = set1.intersection(set2)
print("Intersection (intersection_set_method)
# Output: {3}
```

- Using _ operator:

```
# Difference of set1 and set2 using the - operator
difference_set = set1 - set2
print(difference_set)
# Output: {1, 2}
```

- using difference() method:

```
# Difference of set1 and set2 using
the difference() method
difference_set_method = set1.difference(set2)
print("Difference (difference_set_method)
# Output: {1, 2}
```

Advance set operation

- Symmetric Difference(ôperator): The symmetric difference of two sets is a set containing elements that are in either of the sets, but not in both.

```
# Symmetric difference of set1 and set2
using the ^ operator
symmetric_difference_set = set1 ^ set2
print(symmetric_difference_set)
# Output: {1, 2, 4, 5}
```

- Using the symmetric_difference() Method:

```
#Symmetric difference of set1 and set2 using the
symmetric_difference() method
symmetric_differenc1 = set1.symmetric_difference(set2)
print("Symmetric Difference ( symmetric_difference1)
# Output: {1, 2, 4, 5}
```

Subset and Superset

- **Subset:** Set A is a subset of set B if all elements of A are also elements of B. This can be represented as $A \subseteq B$.
- **Superset:** A set B is a superset of set A if all elements of A are also elements of B. This can be represented as $B \supseteq A$.

```
#Creating two sets
set_a = {1, 2, 3}
set_b = {1, 2, 3, 4, 5}

# Checking if set_a is a subset of set_b
is_subset = set_a.issubset(set_b)
print("Is set_a a subset of set_b?:", is_subset)

# Checking if set_b is a subset of set_a
is_subset_reverse = set_b.issubset(set_a)
print("Is set_b a subset of set_a?:", is_subset_reverse)
#Output:
Is set_a a subset of set_b?: True
Is set_b a subset of set_a?: False
```

Subset and Superset

```
# Creating two sets
set_a = {1, 2, 3}
set_b = {1, 2, 3, 4, 5}

# Checking if set_b is a superset of set_a
is_superset = set_b.issuperset(set_a)
print("Is set_b a superset of set_a?:", is_superset)

# Checking if set_a is a superset of set_b
is_superset_reverse = set_a.issuperset(set_b)
print("Is set_a a superset of set_b?:",is_superset_reverse)
#Output:
Is set_b a superset of set_a?: True
Is set_a a superset of set_b?: False
```

copy operation in set

- copy()- used to copy one set to another.

```
original_set = {1, 2, 3, 4}
copy_set = original_set.copy()

print("Original Set:", original_set)
print("Copied Set:", copy_set)
#Output:
Original Set: {1, 2, 3, 4}
Copied Set: {1, 2, 3, 4}
```

4.7 Lambda

A lambda function in Python is a small anonymous function defined with the lambda keyword. It is often used when we need a simple function that is defined inline and not necessarily required elsewhere in your code. Lambda functions can have any number of arguments but only one expression.

- Syntax

```
lambda arguments: expression

lambda: Keyword that indicates the declaration
of a lambda function.

arguments: Comma-separated list of parameters
(similar to regular functions).

expression: Single expression that is
evaluated and returned.
```

- Key Characteristic of Lambda function:
  - **Anonymous:**Lambda functions are anonymous because they are not given a name like normal functions defined with def.
  - Single Expression: Lambda functions can only contain a single expression, and the result of the expression is returned automatically.
  - Use Cases: Lambda functions are commonly used in situations where you need a short function for a specific task, especially as arguments to higher-order functions (functions that take other functions as arguments).

Example of lambda

```
addition = lambda x, y: x + y
# Using the lambda function
result = addition(3, 5)
print("Result of addition:", result)
# Output: 8
```

- Lambda functions are commonly used with higher-order functions like map(), filter(), and reduce() from Python's functools module.
- Lambda with map()
  - Syntax–> map(function, iterable)
  - function: A function to apply to each item in the iterable.
  - iterable: An iterable (e.g., list) containing the items to apply the function to.

```
numbers = [1, 2, 3, 4, 5]
# Using lambda with map to square each number
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
#Output: [1, 4, 9, 16, 25]
```

- Lambda with filter():Applies a function to each item of an iterable (like a list) and returns a new iterable (e.g., filter object) with the items for which the function returns True.

```
# Sample list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Step 1: Filter to get even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print("Even numbers:", even_numbers)
#Output:
Even numbers: [2, 4, 6, 8, 10]
```

- Lambda with reduce: The reduce() function applies a rolling computation to sequential pairs of values in an iterable, reducing them to a single value.

```
from functools import reduce
#Step 2: Reduce to sum the even numbers
sum_even_numbers = reduce(lambda x, y: x + y, even_numbers)
print("Sum of even numbers:", sum_even_numbers)

#Output:
Sum of even numbers: 30
```

Operation of mutable and immutable data types

Operation of mutable and immutable data types like modification, concatenation, slicing already discussed in the each topic of list,dictionary,set(mutable) and immutable(tuple).

Unit:5 Object Oriented Programming(12hrs)

- 5.1 Concepts of object-oriented programming
- 5.2 Classes and objects
  - 5.2.1 Attributes and methods
  - 5.2.2 __init__() and __str__() methods
  - 5.2.3 Delete properties and objects
  - 5.2.4 Iterator in a class
- 5.3 Aggregation and composition
- 5.4 Inheritance
  - 5.4.1 Parent and child classes
  - 5.4.2 __init__() in child class
  - 5.4.3 The super() function
  - 5.4.4 Member overriding
  - 5.4.5 Forms of Inheritance (Single, Hierarchical, Multiple, Multilevel)
- 5.5 Polymorphism and dynamic binding
  - 5.5.1 Abstract class and concrete class
  - 5.5.2 Abstract methods and abstract attributes
- 5.6 Operator overloading in python
  - 5.6.1 Arithmetic operators
  - 5.6.2 Bitwise and shift operators
  - 5.6.3 Comparison operators
  - 5.6.4 Assignment operators
  - 5.6.5 Unary operators

5.1 Concepts of object-oriented programming

- Problems in Procedure Oriented Approach
  - In procedure oriented approach, the programmer concentrates on a specific task, and writes a set of functions to achieve it.When there is another task to be added to the program, he would be writing another set of functions. Whenever he wants to perform a new task, he would be writing a new set of functions so there is no code reusability of an already existing code.
  - In procedure Oriented approach, every task and sub task is represented as a function and one function depend on another function.Hence error in the program needs examination of all functions.Thus debugging or removing error will bwcome difficult.
  - There is another problem with procedure Oriented approach . Programming in this approach is not develop from human being's life.
- Due to the preceding reasons, Computer scientists felt the need of new approach where programming will have severals modules. Each modules represent a class and the classes can be resuable and hence maintenence of code will become easy.Moreover,This approach is built from a single root concept 'object' which represent anything physically exists in the world.All existings things will become object. All animals are object. All human beings are objects.
- In OOPs, all programs involve creation of classes and objects.Python is an object oriented programming language like jave,it doesnot force the programmers to write programs in complete object oriented way.Unlike Java, Python has a blend of both the object oriented and procedure oriented features.

Features of Object Oriented Programming System(OOPs)

There are five imortant features related to Object Oriented Programming System.They are:

- Classes and Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

5.2 Classes and Objects

Class is a model or plan to create object.Class is written with the attributes or action of objects.Attributes are represented by variables and action are performed by methods.Attributes are the variable that belongs to class.Attributes are always public and can be accesses with (.) dot operator.A function written inside the class is called method. A method is called using one of the following two ways:

- classname.methodname()
- instancename.methodname()

Creation of class

Example:"Student class"

If we take 'student' class , we can write code in the class that specifies the attribute and actions performed by any student. For example,a student has attributes like name,age, marks etc.These attributes should written inside the class as variables. Similarly student can perform actions like name,age,marks etc.The actions should be represented by methods in the student class.

```
class Student:
    def__init__(self):
        self.name='Mukesh'
        self.age=20
        self.marks=900
    def talk(self):
        print('Hi,i am',self.name)
        print('My age is',self.age)
        print('My marks are',self.marks)
```

- Generally class name should start with a capital letter.Hence 'S' is a capital letter in 'Student'.We have written the variable inside the special method, i.e. __init__() method.This method is useful to initialize the variable.
- 'self' is the variable that refers to the current class instance.
- The instance contain the variables 'name','age','marks' which are instance variables. To refer to the instance variable , we can use dot operator notation along with self as 'self.name','self.age' and 'self.marks'.
- the method talk() takes the 'self' variable as parameter,This method displays the values of the variables by referring them using 'self'.
- Instance method use 'self' as the first parameter that refers to the location of the instance in the memory.
- To create an instance, the following syntax is used:

```
instancename=classname()
```

- To create the instance(or object) of the Student class, we can write as:

```
s1=Student()
```

- S1 is the object name.When we create an instance, the following things will take place internally:
  - First of all, a block of memory is allocated on the heap.How much memory is to be allocated is decided from the attributes and method available in the Student class.
  - After allocating the memory block, the special method by the name '__init__(self)' is called internally.This method stores the initial data into the variables.Since this method is useful to construct the instance, it is called 'Constructor'.
  - Finally,the allocated memory location address of the instance is returned into 's1' variable. To see the memory location in decimal number format,we can use id() function as id(s1).
  - Hence any variable or method in the instance can re referenced by 's1' using dot operator as:

```
s1.name
s1.age
s1.marks
s1.talk()
```

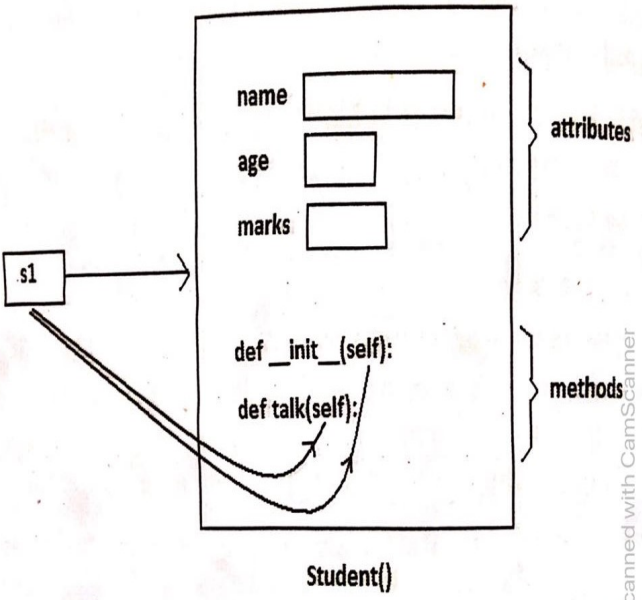

Figure 13.1: Student class instance in memory

The self variable

'self' is a default variable that contains the memory address of the instance of current class.So we can use 'self' to refer to all the variable and instance methods.

- When an instance to the class is created, the instance name contains the memory location of instance.This memory location is internally passed to the 'self'. For example, we create an instance to Student class as:

```
s1=Student()
```

- Here, s1 contain the memory instance address of instance.This memory address is internally and by default passed to 'self' variable. Since 'self' knows the memory address of the instance, it can refer to all the members of the instance. We can use 'self' in two ways:
  - The self variable is used as first parameter in the constructor as:

```
def __init__self()
```

  - 'self' can be used as first parameter in the instance method as:

```
def talk(self):
```

    The memory location of the instance variable is by default available to the talk() method through 'self'.

5.2.1 Attributes and method

The variable which are written inside the class are of 2 types:

- Instance variable
- Class variables or Static variables

Instance variables

Instance variables are the variables whose separate copy is created in every instance(or object).If 'x' is an instance variable and if we create 3 instances, there will be 3 copies of 'x' in these 3 instances.When we modify the copy of 'x' in any instance, it will not modify the other two copies.

```
class Sample:
  def __init__(self):
    self.x=10
  def modify(self):
    self.x+=1
#create 2 instances
s1=Sample()
s2=Sample()
print('x in s1=',s1.x)
print('x in s2=',s2.x)
#modify x in s1
s1.modify()
print('x in s1=',s1.x)
print('x in s2=',s2.x)
```

```
#Output:
x in s1=10
x in s2=10
x in s1=11
x in s2=10
```

- **Class variable or Static Variable:**
  Unlike instance variables , class variables are the variables whose single copy is available to all the instance of class. if we modify the copy of class variable in an instance, it will modify all the copies in the other instances.For example, if 'x' is a class variable and if we create 3 instances, the same copy of x is passed to these 3 instances.When we modify the copy of 'x' in any instance using a class method , the modify copy is sent to other two instances.

```
Class Sample:
    ‘#this is class var
      x=10
      #this is class method
      @classmethod
      def modify(cls):
        cls.x+=1
#create instance two
s1=Sample()
s2=Sample()
print(’x in s1=’,s1.x)
print(’x in s2=’,s2.x)
#modify x in s1
s1.modify()
print(’x in s1=’,s1.x)
print(’x in s2=’,s2.x)
#Output:
x in s1=10
x in s2=10
x in s1=11
x in s2=11
```

Class method acting on class variable.To mark this method as class method, we should use built-in decorator statement @classmethod.

Types of method

- Instance method
  - Accessor methods
  - Mutator methods
- Class methods
- Static methods

Instance method

Instance methods are the methods which act upon the instance variables of the class.Instance methods are bound to instance(or objects) and hence called as: instancename.method().Since instance variable are available in the instance, instance methods need to know the memory address of the instance.This is provided through 'self' variable by default as first parameter for the instance method.

```
#instance methods to process data of the objects
class student:
  def __init__(self,n,m):
    self.name=n
    self.marks=m
  #this is instance method
  def display(self):
    print('Hi',self.name)
    print('Your marks',self.marks)
  def calculate(self):
    if(self.marks>=600):
      print('You have got first grades')
    elif(self.marks>=500):
      print('You got the second grade')
    elif(self.marks>=350):
      print('You have got third grade')
    else:
      print('You have failed')
```

```
#create instance with some data from keyboard
n=int(input('how many students?'))
i=0
while(i<n):
  name=input('enter name:')
  marks=int(input('enter marks:'))
  #create the student class instance and store data
  s=student(name,marks)
  s.display()
  s.calculate()
  i+=1
  print('--------------------------------------------------------')
#Output
how many students?3
enter name:Mukesh
enter marks:600
Hi Mukesh
Your marks 600
You have got first grades
--------------------------------------------------------
enter name:Inogen
enter marks:350
Hi Inogen
Your marks 350
You have got third grade
--------------------------------------------------------
```

```
enter name:Kabi
enter marks:400
Hi Kabi
Your marks 400
You have got third grade
--------------------------------------------------
```

Each instance are not stored in separate memory storage in this program. How instance are stored in separate memory so that it can retrieve the instance information after stored data by using unique name of instance?

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    # Instance method to display student information
    def display(self):
        print('Hi', self.name)
        print('Your marks:', self.marks)
    # Instance method to calculate and display the grade
    def calculate(self):
        if self.marks >= 600:
            print('You have got first grade')
        elif self.marks >= 500:
            print('You got the second grade')
        elif self.marks >= 350:
            print('You have got third grade')
        else:
            print('You have failed')
```

```
#Create instances for students
num_students = int(input('How many students? '))
students = []
for _ in range(num_students):
    name = input('Enter name: ')
    marks = int(input('Enter marks: '))
    # Create the Student class instance and store data
    student_instance = Student(name, marks)
    students.append(student_instance)
    student_instance.display()
    student_instance.calculate()
    print('-----------------------------------------------------
# Now you have a list 'students' with separate instances for
each student
How many students? 2
Enter name: Mukesh
Enter marks: 600
Hi Mukesh
Your marks: 600
You have got first grade
-----------------------------------------------------
Enter name: Kabi
Enter marks: 400
Hi Kabi
Your marks: 400
You have got third grade
-----------------------------------------------------
```

```
students[0].display()
students[0].calculate()
#Output
Hi Mukesh
Your marks: 600
You have got first grade
```

Types of instance method

- Accessor methods: Accessor methods simply access or read data of variable.They do not modify the data in the variable.Accessor methods are generally written in the form of getXXX() and hence they are also called getter methods. For example:

```
def getName(self):
    return self.name
```

- Mutator methods are the methods which not only read the data but also modify them.They are written in the form of setXXX() and hence they are called setter method.

```
def setName(self,name):
    self.name=name
```

Exercise

Write a python program to store data into instances using mutator method and to retrieve data from instance using accessor method.

```
#accessor and mutator method
class Student:
  #mutator method
  def setName(self,name):
    self.name=name
  #accessor method
  def getName(self):
    return self.name
  #mutator method
  def setMarks(self,marks):
    self.marks=marks
  #accessor method
  def getMarks(self):
    return self.marks
```

```
#Create instances for students
num_students = int(input('How many students? '))
students = []
for _ in range(num_students):
    # Create the Student class instance and store data
    student_instance = Student()
    name = input('Enter name: ')
    student_instance.setName(name)
    marks = int(input('Enter marks: '))
    student_instance.setMarks(marks)
    students.append(student_instance)
    print('Hi',student_instance.getName())
    print('Your maarks',student_instance.getMarks())
    print('-----------------------------------------------------
#Output:
How many students? 1
Enter name: Mukesh
Enter marks: 400
Hi Mukesh
Your maarks 400
-------------------------------------------------------
print(students[0].getName())
print(students[0].getMarks())
#Output:
Mukesh
400
```

Class method

In Python, a class method is a method that is bound to the class and not the instance of the class. This means that it can be called on the class itself, rather than on an instance of the class. To define a class method, you use the @classmethod decorator. The first parameter of a class method is cls, which refers to the class itself.

```
class MyClass:
    class_variable = "I am a class variable"
    def __init__(self, instance_variable):
        self.instance_variable = instance_variable
    @classmethod
    def class_method(cls):
        print(f"Class method called. class_variable:
        {cls.class_variable}")
    def instance_method(self):
        print(f"Instance method called. instance_variable:
        {self.instance_variable}")
# Creating an instance of MyClass
my_instance = MyClass("I am an instance variable")
# Calling the instance method
my_instance.instance_method()
# Calling the class method
MyClass.class_method()
```

```
#Output:

Instance method called. instance_variable:
I am an instance variable

Class method called. class_variable:
I am a class variable
```

Static method

A static method is a method that belongs to a class but doesn't require an instance of the class to be called. Static methods do not have access to self or cls, which means they cannot modify object or class state. They are defined using the @staticmethod decorator.

```
class MathOperations:
    # Defining a static method
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def subtract(a, b):
        return a - b

# Calling the static method without creating
an instance of the class

result_add = MathOperations.add(5, 3)
result_subtract = MathOperations.subtract(5, 3)

print("Addition:", result_add)         # Output: Addition: 8
print("Subtraction:", result_subtract) # Output: Subtraction: 2
```

5.2.2 __init__() and __str__() methods

In Python, __init__() and __str__() are special methods used in classes.

- __init__() method:
  - This method is called the initializer or constructor method.
  - It is automatically called when a new instance of the class is created.
  - It is typically used to initialize instance variables (attributes) of the class

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = MyClass(3, 4)
```

- __str__() method:
  - This method is used to define a printable string representation of an object.
  - It is called when print() or str() function is used on an object.
  - It should return a string that is readable and human-friendly.

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return (f"MyClass object with x={self.x} and
        y={self.y}")

obj = MyClass(3, 4)
print(obj)

# Output: MyClass object with x=3 and y=4
```

5.2.3 Delete properties and objects

Delete attribute: We can delete an attribute of an object using the del statement.

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"MyClass object with x={self.x} and y={self.y}"

obj = MyClass(3, 4)
print(obj)  # Output: MyClass object with x=3 and y=4
print(id(obj.x))
print(id(obj.y))

# Deleting an attribute
del obj.x

#Trying to access deleted attribute will raise an
AttributeError
#Output
MyClass object with x=3 and y=4
139460385472816
139460385472848
```

```
id(obj.x)
#Output:
AttributeError: 'MyClass' object has no
attribute 'x'

id(obj.y)

#Output
139460385472848
```

- **Delete object:**We can delete an entire object using the del statement. However, this only decrements the reference count for the object, and the object will be garbage collected when there are no more references to it. In Python, garbage collection is the process of automatically freeing memory occupied by objects that are no longer in use or referenced by the program.

```
obj = MyClass(3, 4)
print(obj)
# Output: MyClass object with x=3 and y=4

# Deleting the object
del obj

# Trying to access the deleted object will raise
a NameError
```

5.2.4 Iterator in class

- __iter__():The __iter__ method is expected to return the iterator object itself.It is called once when the iteration starts.
- The __next__ method is called to obtain the next item from the iterator. It should return the next item in the sequence. If there are no more items to return, it should raise the StopIteration exception.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Passing Members of One class to Another Class

It is possible to pass the members(i.e attributes and methods) of a class to another class.Let's take an Emp class with constructor that defines attributes 'id','name', and 'salary'.This class has an instance method display() to display these values. If we create an object(or instance) of Emp class,it contain copy of all attributes and methods.To pass all these members of Emp class to another class, we should pass Emp class instance to the other class. For example, let's create an instance of Emp class as:

```
e=Emp()
```

Then pass this instance 'e' to a method of other class, as:

```
Myclass.mymethod(e)
```

Here, Myclass is the other class and mymethod() is a static method that belongs to Myclass. In Myclass , the method mymethod() will be declared as a static method as it acts neither on the class variable nor instance variable of Myclass.

```
#Example 1:
class Emp:
    def __init__(self,id,name,salary):
        self.id=id
        self.name=name
        self.salary=salary
    def display(self):
        print('Id=',self.id)
        print('name=',self.name)
        print('salary=',self.salary)
class Myclass:
    @staticmethod
    def mymethod(e):
        e.salary+=1000
        e.display()

#create Emp class instance e
e=Emp(10,'Mukesh Kumar Pokhrel',20000)
#call static method by Myclass and pass e
Myclass.mymethod(e)
```

```
#Example 2:
class Emp:
    def __init__(self, id, name, salary):
        self.id = id
        self.name = name
        self.salary = salary
    def display(self):
        print('Id =', self.id)
        print('Name =', self.name)
        print('Salary =', self.salary)
class Myclass:
    def mymethod(self, e):
        e.salary += 1000
        e.display()

# Create Emp class instance e
e = Emp(10, 'Mukesh Kumar Pokhrel', 20000)
# Create Myclass instance
myclass_instance = Myclass()
# Call instance method by Myclass instance and pass e
myclass_instance.mymethod(e)

#Output:
Id= 10
name= Mukesh Kumar Pokhrel
salary= 21000
```

Inner Class

Writing a class within another class is called creating inner class or nested class.Inner Classes are useful when we want to sub group the data of class.Let's take an example:

```
class person:
  def __init__(self):
    self.name='Mukesh'
    self.db=self.Dob()
  def display(self):
    print('Name=',self.name)
  class Dob:
    def __init__(self):
      self.dd=25
      self.mm=11
      self.yy=2052
    def display(self):
      print('DOB={}/{}/{}'.format(self.dd,self.mm,self.yy))
p=person()
p.display()
#create inner class object
x=p.db
x.display()
#Output:
Name= Mukesh
DOB=25/11/2052
```

5.3 Aggregation and Composition

In aggregation, the lifetime of the components is independent of the containing object. The components can exist even if the containing object is destroyed.

```
class Student:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
class Teacher:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name
class Classroom:
    def __init__(self, class_name, teacher, student):
        self.class_name = class_name
        self.teacher = teacher
        self.student = student
    def __str__(self):
        return (f"Classroom {self.class_name} with Teacher
        {self.teacher} and Student {self.student}")
```

```
# Example usage
teacher = Teacher("Inogen")
students =Student("Mukesh")
classroom = Classroom("Physics", teacher, students)
print(classroom)
del classroom
# Even if the classroom object is deleted
# The teacher and students objects still exist
print(teacher)
print(students)

#Output:
Classroom Physics with Teacher Inogen and Student Mukesh
Inogen
Mukesh
```

Composition

In composition, the lifetime of the composed objects (components) is tightly bound to the lifetime of the containing object. When the containing object is destroyed, so are the components.

```
class PrintedCircuitBoard:
    def __init__(self, pcb_id):
        self.pcb_id = pcb_id
    def __str__(self):
        return f"PCB {self.pcb_id}"
class Amplifier:
    def __init__(self, amplifier_id):
        self.amplifier_id = amplifier_id
        self.pcb = PrintedCircuitBoard(pcb_id=amplifier_id)
    def __str__(self):
        return f"Amplifier {self.amplifier_id} with {self.pcb}"
# Example usage
amp = Amplifier(1)
print(amp)  # Output: Amplifier 1 with PCB 1
# If we delete the amplifier object
del amp
# The PrintedCircuitBoard object is also deleted
```

5.4 Inheritance

This is a mechanism where a new class (child class) inherits the properties and behaviors (attributes and methods) of an existing class (parent class). Inheritance allows for code reusability and the creation of a hierarchical relationship between classes.

5.4.1 Parent and child classes

- In object-oriented programming (OOP), a parent class (also known as a base class or superclass) is a class that provides attributes and methods that can be inherited by another class.
- A child class (also known as a derived class or subclass) is a class that inherits attributes and methods from a parent class and can also have its own additional attributes and methods.

```
class Employee:
    def __init__(self, id, name, salary):
        self.id = id
        self.name = name
        self.salary = salary
    def display(self):
        print('Id =', self.id)
        print('Name =', self.name)
        print('Salary =', self.salary)
```

```
class Manager(Employee):
    def __init__(self, id, name, salary, department):
        super().__init__(id, name, salary)
        self.department = department
    def display(self):
        super().display()#Call parent class display method
        print('Department =', self.department)
    def give_raise(self, amount):
        self.salary += amount

# Create an instance of the Employee class
e = Employee(1, 'John Doe', 30000)
e.display()
print()

# Create an instance of the Manager class
m = Manager(2, 'Jane Smith', 50000, 'HR')
m.display()
print()

# Give the manager a raise and display details again
m.give_raise(5000)
m.display()
```

```
Id = 1
Name = John Doe
Salary = 30000

Id = 2
Name = Jane Smith
Salary = 50000
Department = HR

Id = 2
Name = Jane Smith
Salary = 55000
Department = HR
```

Member Overriding

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

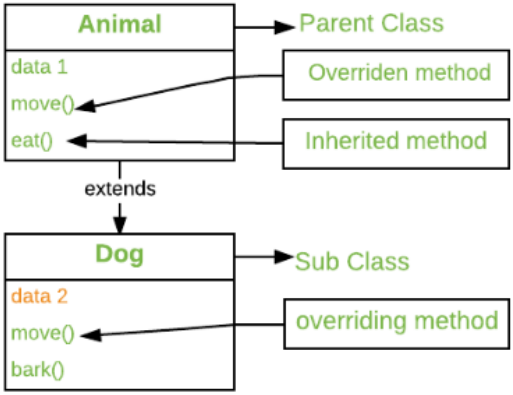

Figure: Method Overiding

```
# Defining parent class
class Parent():
    # Constructor
    def __init__(self):
        self.value = "Inside Parent"
    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):
    # Constructor
    def __init__(self):
        self.value = "Inside Child"
    # Child's show method
    def show(self):
        print(self.value)

# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
# Output
Inside Parent
Inside Child
```

5.4.5 Forms of Inheritance

- Single
- Hierarchical
- Multiple
- Multilevel

Single inheritance

- A class inherits from one and only one superclass.
- Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

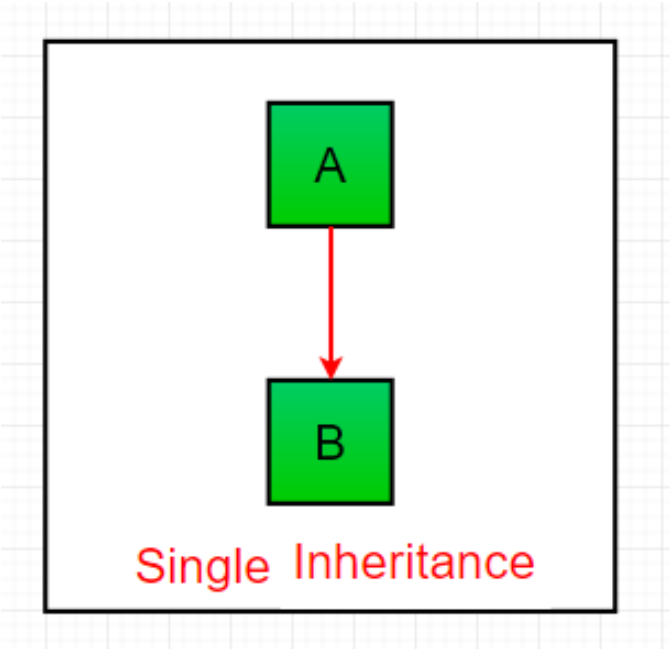

Figure: Single inhertance

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
d = Dog()
d.speak()  # Inherited from Animal
d.bark()   # Defined in Dog

#Output:
Animal speaks
Dog barks
```

Hierarchical Inheritance

- Multiple classes inherit from a single superclass.
- This is useful for defining a common set of features in the superclass that are shared by multiple subclasses.

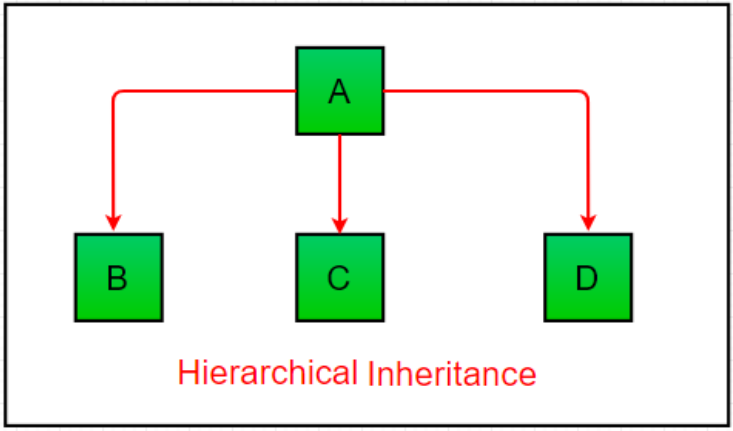

Figure: Hierarchical Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

class Cat(Animal):
    def meow(self):
        print("Cat meows")

d = Dog()
d.speak()  # Inherited from Animal
d.bark()   # Defined in Dog

c = Cat()
c.speak()  # Inherited from Animal
c.meow()   # Defined in Cat

#Output
Animal speaks
Dog barks
Animal speaks
Cat meows
```

Multiple Inheritance

- When a class can be derived from more than one base class this type of inheritance is called multiple inheritances.
- In multiple inheritances, all the features of the base classes are inherited into the derived class.

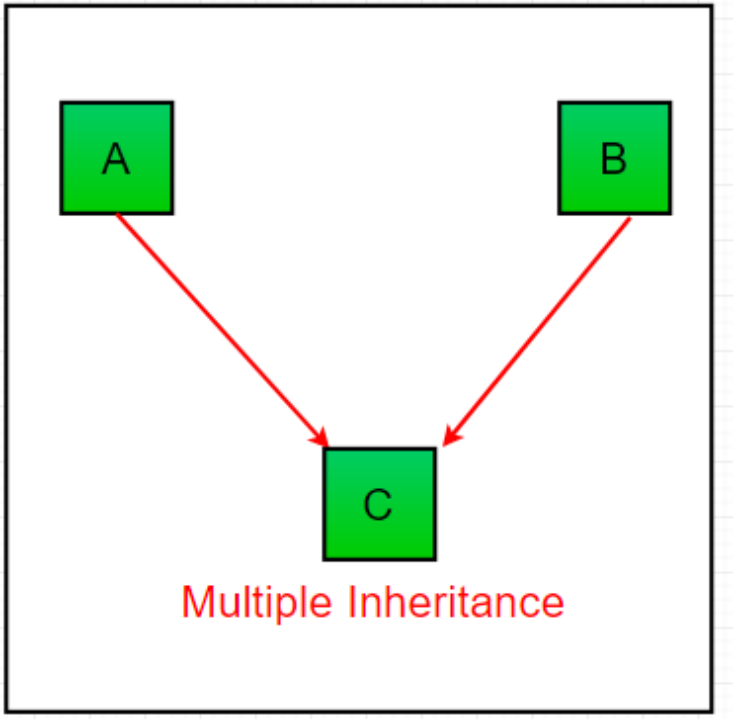

Figure: Multiple Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Mammal:
    def feed(self):
        print("Mammal feeds")

class Dog(Animal, Mammal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # Inherited from Animal
d.feed()   # Inherited from Mammal
d.bark()   # Defined in Dog

#Output
Animal speaks
Mammal feeds
Dog barks
```

Multilevel Inheritance

- In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class.
- This is similar to a relationship representing a child and a grandfather.

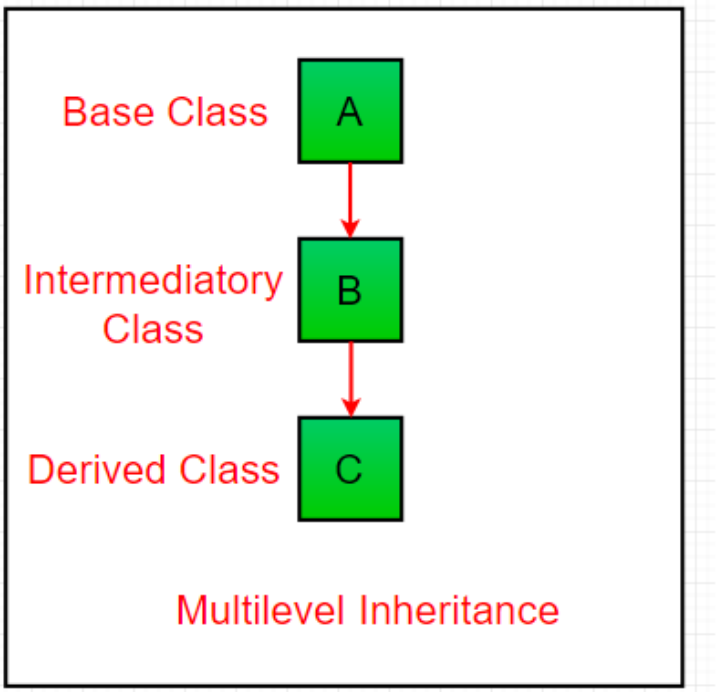

Figure: Multilevel Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Mammal(Animal):
    def feed(self):
        print("Mammal feeds")

class Dog(Mammal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # Inherited from Animal
d.feed()   # Inherited from Mammal
d.bark()   # Defined in Dog

#Output
Animal speaks
Mammal feeds
Dog barks
```

5.5 Polymorphism and dynamic binding

- 5.5.1 Abstract class and concrete class
- 5.5.2 Abstract nethods and abstract attributes

- Polymorphism is a word that came from Greek words, poly means many and 'morphos' means forms.If somethings exhibits various forms,it is called polymorphism.
- Example of polymorphism:
  - Operator overloading
  - Method Overloading
  - Method Overriding
  - Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables the same operation to behave differently on different classes. There are two main types of polymorphism in programming:
    - Compile-Time Polymorphism (Static Polymorphism): Achieved through method overloading or operator overloading.
    - Run-Time Polymorphism (Dynamic Polymorphism): Achieved through method overriding, typically using inheritance and interfaces.

Abstract Class and Concrete class

- A class that contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation.

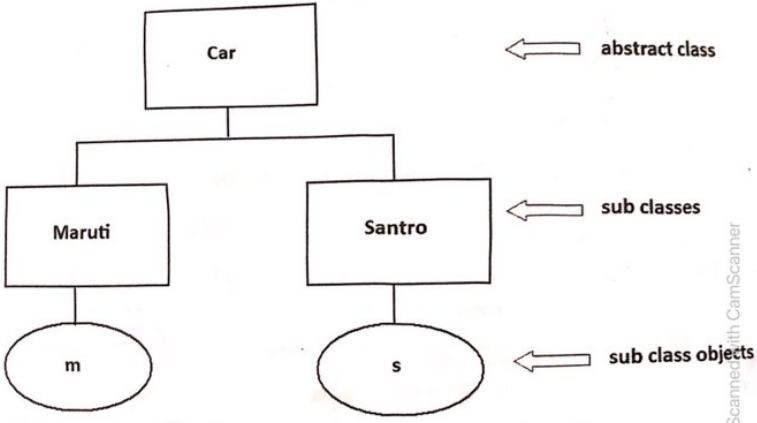

Figure 15.2: Abstract Car Class and its Sub Classes

```
from abc import ABC, abstractmethod
# Abstract class
class Car(ABC):

    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

# Concrete class Maruti
class Maruti(Car):

    def start_engine(self):
        return "Maruti engine started."

    def stop_engine(self):
        return "Maruti engine stopped."
```

```
# Concrete class Santro
class Santro(Car):

    def start_engine(self):
        return "Santro engine started."

    def stop_engine(self):
        return "Santro engine stopped."

# Instantiating subclass objects
m = Maruti()
s = Santro()

print(m.start_engine())  # Output: Maruti engine started.
print(m.stop_engine())   # Output: Maruti engine stopped.

print(s.start_engine())  # Output: Santro engine started.
print(s.stop_engine()) # Output: Santro engine stopped.

#Output
Maruti engine started.
Maruti engine stopped.
Santro engine started.
Santro engine stopped.
```

Abstract method and abstract property

- In Python, abstract classes can have abstract methods as well as abstract attributes. Abstract methods are methods that must be implemented by any subclass, and abstract attributes are attributes that must be defined by any subclass.
- To create abstract attributes, you typically use the @abstractmethod decorator for methods, and for attributes, you can define abstract properties using the @property decorator along with @abstractmethod.

```
from abc import ABC, abstractmethod
# Abstract class
class Car(ABC):

    @property
    @abstractmethod
    def brand(self):
        pass

    @property
    @abstractmethod
    def model(self):
        pass

    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass
```

```
# Concrete class Maruti
class Maruti(Car):

    @property
    def brand(self):
        return "Maruti"

    @property
    def model(self):
        return "Swift"

    def start_engine(self):
        return "Maruti engine started."

    def stop_engine(self):
        return "Maruti engine stopped."
```

```
# Concrete class Santro
class Santro(Car):
    @property
    def brand(self):
        return "Hyundai"
    @property
    def model(self):
        return "Santro"
    def start_engine(self):
        return "Santro engine started."
    def stop_engine(self):
        return "Santro engine stopped."
# Instantiating subclass objects
m = Maruti()
s = Santro()
print(f"Brand: {m.brand}, Model: {m.model}")
print(m.start_engine())  # Output: Maruti engine started.
print(m.stop_engine())   # Output: Maruti engine stopped.

print(f"Brand: {s.brand}, Model: {s.model}")
print(s.start_engine())  # Output: Santro engine started.
print(s.stop_engine())   # Output: Santro engine stopped.
```

5.6 Operator Overloading

The operator overloading in Python means provide extended meaning beyond their predefined operational meaning. Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists.
Suppose the user has two objects which are the physical representation of a user-defined data type class. The user has to add two objects using the "+" operator, and it gives an error. This is because the compiler does not know how to add two objects. So, the user has to define the function for using the operator, and that process is known as "operator overloading".
The user can overload all the existing operators by they cannot create any new operator.

5.6.1 Arithmetic Operator

Arithmetic operators like +, -, *, and / can be overloaded to perform custom operations.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)
    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
    def __truediv__(self, scalar):
        return Vector(self.x / scalar, self.y / scalar)
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
print(v1 + v2)  # Vector(6, 8)
print(v1 - v2)  # Vector(-2, -2)
print(v1 * 2)   # Vector(4, 6)
print(v1 / 2)   # Vector(1.0, 1.5)
```

5.6.2 Bitwise and Shift Operators

Bitwise operators (&, |, ^, <<, >>) can be overloaded to perform operations at the bit level.

```
class BinaryNumber:
    def __init__(self, value):
        self.value = value
    def __and__(self, other):
        return BinaryNumber(self.value & other.value)
    def __or__(self, other):
        return BinaryNumber(self.value | other.value)
    def __xor__(self, other):
        return BinaryNumber(self.value ^ other.value)
    def __lshift__(self, shift):
        return BinaryNumber(self.value << shift)
    def __rshift__(self, shift):
        return BinaryNumber(self.value >> shift)
    def __repr__(self):
        return bin(self.value)

b1 = BinaryNumber(0b1010)
b2 = BinaryNumber(0b1100)
print(b1 & b2)  # 0b1000
print(b1 | b2)  # 0b1110
print(b1 ^ b2)  # 0b0110
print(b1 << 2)  # 0b101000
print(b1 >> 2)  # 0b10
```

5.6.3 Comparison Operator

Comparison operators like ==, !=, <, >, <=, >= can be overloaded to compare objects.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def __eq__(self, other):
        return self.area() == other.area()
    def __lt__(self, other):
        return self.area() < other.area()
    def __le__(self, other):
        return self.area() <= other.area()
    def __repr__(self):
        return f"Rectangle({self.width}, {self.height})"

r1 = Rectangle(2, 3)
r2 = Rectangle(3, 3)
r3 = Rectangle(2, 3)
print(r1 == r2)  # False
print(r1 == r3)  # True
print(r1 < r2)   # True
print(r1 <= r2)  # True
print(r1 <= r3)  # True
```

5.6.4 Assignment Operator

Assignment operators like +=, -=, *=, /= can be overloaded to perform in-place operations.

```
class Counter:
    def __init__(self, count=0):
        self.count = count
    def __iadd__(self, other):
        self.count += other
        return self
    def __isub__(self, other):
        self.count -= other
        return self
    def __imul__(self, other):
        self.count *= other
        return self
    def __itruediv__(self, other):
        self.count /= other
        return self
    def __repr__(self):
        return str(self.count)
```

```
c = Counter(10)
c += 5
print(c)  # 15
c -= 3
print(c)  # 12
c *= 2
print(c)  # 24
c /= 4
print(c)  # 6.0
```

5.6.5 Unary operator

Unary operators like - and can be overloaded to perform custom operations.

```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius
    def __neg__(self):
        return Temperature(-self.celsius)
    def __invert__(self):
        return Temperature(~int(self.celsius))
    def __repr__(self):
        return f"{self.celsius}°C"

t = Temperature(25)
print(-t)   # -25°C
print(~t)   # -26°C (bitwise NOT of 25 is -26 in two's
complement)
```