Python Code and Notes By Vithusan

Variables

1. **Dynamic Typing**:

- Python uses dynamic typing. This means you don't need to explicitly declare the type of a variable when assigning a value to it.
- The variable type is inferred at runtime based on the value assigned to it.

2. **Variable Naming Conventions**:

- Variable names can contain letters, numbers, and underscores but can't start with a number.
- Python variables are case-sensitive.
- Descriptive variable names improve code readability.
- It's a good practice to use snake_case (lowercase with underscores) for variable names.

3. **Variable Assignment**:

- Assign values to variables using the `=` operator.
- Multiple assignments can be done on a single line.
- Variables can be reassigned to different data types during the execution of a program.

4. **Variable Scope**:

- The scope of a variable defines where it can be accessed or referenced within the code.
- Variables declared inside a function have local scope and are only accessible within that function.
- Variables declared outside functions or at the global level have global scope and can be accessed from anywhere in the code.

5. **Immutable vs. Mutable**:

- Immutable data types (e.g., int, float, str, tuple) cannot be changed after creation. Reassigning a value creates a new object.
- Mutable data types (e.g., list, dict, set) can be modified after creation without changing their identity.

6. **Variable Types**:

- Python has various built-in data types for variables, including integers (`int`), floating-point numbers (`float`), strings (`str`), lists (`list`), tuples (`tuple`), dictionaries (`dict`), sets (`set`), and more.

- Type casting can be done using functions like `int()`, `float()`, `str()`, etc., to convert variables from one type to another.

7. Deleting Variables:

- Use the `del` keyword to delete a variable and free up the memory it occupies.

Example:

Variable assignment

x = 10 y = "Hello" z = [1, 2, 3]

Variable types and type casting

x = float(x) y = list(y)

Variable scope

def my_function(): local_var = "Local variable" print(local_var) # Accessible within the function

my_function() print(x, y, z) # Accessible here

Deleting variables

del z

In [22]:
```
a = 10
b = 10
print(a+b)
```
```
20
```

In [23]:
```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```
```
Python is awesome
```

In [24]:
```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```
```
Python is fantastic
```

In [16]:
```
# Variable assignment
x = 10
y = "Hello"
z = [1, 2, 3]
```

```python
# Printing variable values
print("x:", x)
print("y:", y)
print("z:", z)

# Variable types and type casting
x = float(x)
y = list(y)

print("Type of x:", type(x))
print("Type of y:", type(y))

# Variable scope
global_var = "Global variable"

def my_function():
    local_var = "Local variable"
    print("Inside function - local_var:", local_var)  # Accessible within the funct
    print("Inside function - global_var:", global_var)  # Accessible within the fur

my_function()
print("Outside function - global_var:", global_var)  # Accessible here

# Deleting variables
del z

# Error: This will raise an error because z is deleted
# print("z:", z)
```

```
x: 10
y: Hello
z: [1, 2, 3]
Type of x: <class 'float'>
Type of y: <class 'list'>
Inside function - local_var: Local variable
Inside function - global_var: Global variable
Outside function - global_var: Global variable
```

Datatypes

1. Built-in Data Types:

- Python provides several built-in data types:
  - **Numeric Types**: Integers (`int`), Floating-point numbers (`float`), Complex numbers (`complex`)
  - **Sequence Types**: Strings (`str`), Lists (`list`), Tuples (`tuple`)
  - **Mapping Type**: Dictionaries (`dict`)
  - **Set Types**: Sets (`set`), Frozen sets (`frozenset`)
  - **Boolean Type**: Boolean (`bool`)
  - **None Type**: `None`

2. Dynamic Typing:

- Python is dynamically typed, meaning you don't need to declare the data type explicitly.
- The interpreter infers the data type based on the value assigned to the variable.

3. Mutable vs. Immutable:

- **Mutable Types**: Can be modified after creation (e.g., lists, dictionaries, sets).
- **Immutable Types**: Cannot be modified after creation (e.g., integers, floats, strings, tuples).

4. Type Casting:

- You can convert between different data types using built-in functions like `int()`, `float()`, `str()`, `list()`, `tuple()`, `dict()`, `set()`, etc.

5. Sequences:

- **Strings**: Immutable sequences of characters enclosed in single or double quotes.
- **Lists**: Mutable sequences of elements enclosed in square brackets `[]`.
- **Tuples**: Immutable sequences of elements enclosed in parentheses `()`.

6. Mapping:

- **Dictionaries**: Collection of key-value pairs enclosed in curly braces `{}`. Keys are unique and immutable; values can be of any data type.

7. Sets:

- **Sets**: Unordered collections of unique elements enclosed in curly braces `{}`. Sets do not allow duplicate elements.

8. Boolean Type:

- Represents truth values, `True` or `False`, used for logical operations and conditions.

9. None Type:

- Represents absence of a value or null.

10. Type Checking and Conversion:

- Use `type()` function to check the type of a variable.
- Type conversion is done explicitly using type-casting functions or implicitly in certain operations.

In [7]:

```python
a = 10
b = "vithu"

print(type(b))
print(type(a))
```

```
<class 'str'>
<class 'int'>
```

In [17]:

```python
# Numeric types
integer_num = 10
```

```python
float_num = 3.14
complex_num = 2 + 3j

print("Integer Number:", integer_num)
print("Float Number:", float_num)
print("Complex Number:", complex_num)

# Strings
string_var = "Hello, Python!"
print("String:", string_var)

# Lists
list_var = [1, 2, 3, 4, 5]
print("List:", list_var)

# Tuples
tuple_var = (10, 20, 30)
print("Tuple:", tuple_var)

# Dictionaries
dict_var = {'one': 1, 'two': 2, 'three': 3}
print("Dictionary:", dict_var)

# Sets
set_var = {1, 2, 3, 4, 5}
print("Set:", set_var)

# Boolean
bool_var = True
print("Boolean:", bool_var)

# None Type
none_var = None
print("None Type:", none_var)
```

```
Integer Number: 10
Float Number: 3.14
Complex Number: (2+3j)
String: Hello, Python!
List: [1, 2, 3, 4, 5]
Tuple: (10, 20, 30)
Dictionary: {'one': 1, 'two': 2, 'three': 3}
Set: {1, 2, 3, 4, 5}
Boolean: True
None Type: None
```

In [9]:

```python
#Operators(+ - * / %)

a = 5
b = 2
c = a + b

print(c)
```

```
7
```

Type Casting

1. **Implicit Type Conversion**:

- Python performs implicit type conversion in certain situations automatically.

- For example, adding an integer and a float results in a float, as Python automatically converts the integer to a float.

Example: x = 10 # Integer y = 3.14 # Float

result = x + y # Implicit conversion of integer to float print(result) # Output will be a float: 13.14

2. **Explicit Type Conversion**:

- Python provides built-in functions to explicitly convert variables from one type to another.
- Common type-casting functions include `int()`, `float()`, `str()`, `list()`, `tuple()`, `dict()`, `set()`, etc.

Example:

Explicit type conversion

num_str = "25" # String containing a number

Converting string to integer

num_int = int(num_str) print(num_int) # Output will be an integer: 25

3. **Conversion Between Compatible Types**:

- Type casting works between compatible data types where conversion is meaningful.
- For instance, converting a string representing a numeric value to an integer or float.

Example:

Converting string to float

num_float = float(num_str) print(num_float) # Output will be a float: 25.0

4. **Potential Data Loss**:

- Be cautious when converting between types, as there might be data loss in certain conversions.
- For instance, converting a floating-point number to an integer results in the loss of decimal values.

Example:

Floating-point to integer conversion (data loss)

float_num = 3.75 int_num = int(float_num) print(int_num) # Output will be an integer: 3 (decimal part is truncated)

5. Handling Errors:

- Type casting can sometimes raise errors if the conversion is not possible.
- For instance, trying to convert a string that doesn't represent a valid number to an integer will raise a `ValueError`.

Example: invalid_str = "Hello"

Error: This will raise a ValueError because "Hello" cannot be converted to an integer

invalid_int = int(invalid_str)

In [18]:
```python
# Implicit type conversion
x = 10  # Integer
y = 3.14  # Float

result = x + y  # Implicit conversion of integer to float
print("Implicit Conversion Result:", result)  # Output will be a float: 13.14

# Explicit type conversion
num_str = "25"  # String containing a number

# Converting string to integer
num_int = int(num_str)
print("Integer Conversion Result:", num_int)  # Output will be an integer: 25

# Converting string to float
num_float = float(num_str)
print("Float Conversion Result:", num_float)  # Output will be a float: 25.0

# Floating-point to integer conversion (data loss)
float_num = 3.75
int_num = int(float_num)
print("Floating to Integer Conversion Result:", int_num)  # Output will be an integ

# Handling errors - converting invalid string to integer
invalid_str = "Hello"
# Error: This will raise a ValueError because "Hello" cannot be converted to an int
# invalid_int = int(invalid_str)
```

```
Implicit Conversion Result: 13.14
Integer Conversion Result: 25
Float Conversion Result: 25.0
Floating to Integer Conversion Result: 3
```

In [15]:
```python
#Converting one data type to another Data type

a = int("10") #Convert str to int
b = int("20")

c = a + b

print(c)
```

```
30
```

Get User input

1. `input()` Function:

- The `input()` function is used to take user input from the keyboard.
- It waits for the user to enter some text and press Enter.
- By default, `input()` treats the user input as a string.

Example: user_input = input("Enter something: ") print("You entered:", user_input)

2. Prompting User for Input:

- You can provide a prompt inside the `input()` function to instruct the user on what to input.
- The prompt is displayed on the console before waiting for input.

Example: name = input("Enter your name: ") print("Hello,", name)

3. Type Conversion of Input:

- `input()` returns a string even if the user enters a number or other data types.
- Use type casting functions (`int()` , `float()` , etc.) to convert input to the desired data type.

Example: age = int(input("Enter your age: ")) print("Your age is:", age)

4. Handling User Input:

- User input should be validated and handled carefully, especially when expecting specific types or formats.
- Consider using conditional statements or try-except blocks to handle unexpected input.

Example: while True: try: num = float(input("Enter a number: ")) print("You entered:", num) break except ValueError: print("Invalid input. Please enter a number.")

5. Whitespace Trimming:

- `input()` function returns the user's input with leading and trailing whitespace removed.
- To preserve leading or trailing whitespace, use `raw_input()` in Python 2 or manipulate the input string directly.

Example: user_input = input("Enter something: ") print("Length of input:", len(user_input)) # Counts characters without leading/trailing whitespace

In [1]:
```
a = input() #Default input method will be in string mode
b = input()
c = a + b

print("Without conversion : ", c)

x = int(input())
y = int(input())
z = x + y
```

```
print("With Conversion : ", z)
```

```
12
22
Without conversion :  1222
22
22
With Conversion :  44
```

In [19]:

```
# Getting user input
name = input("Enter your name: ")
print("Hello,", name)

# Converting input to integer
while True:
    try:
        age = int(input("Enter your age: "))
        print("Your age is:", age)
        break
    except ValueError:
        print("Invalid input. Please enter a valid age.")

# Handling whitespace and invalid input
while True:
    user_input = input("Enter a number: ")
    user_input = user_input.strip()  # Removing leading and trailing whitespace

    if user_input.isdigit():  # Checking if the input consists of digits
        num = int(user_input)
        print("You entered:", num)
        break
    else:
        print("Invalid input. Please enter a valid number.")

# Handling floats and invalid input
while True:
    try:
        float_input = float(input("Enter a floating-point number: "))
        print("You entered:", float_input)
        break
    except ValueError:
        print("Invalid input. Please enter a valid floating-point number.")
```

```
Enter your name: Vithu
Hello, Vithu
Enter your age: 22
Your age is: 22
Enter a number: 5
You entered: 5
Enter a floating-point number: 7.2
You entered: 7.2
```

sample input Excercise

In [ ]:

```
name = input("Enter your name : ")
age = int(input("Enter your age : "))

print("My name is " + name)
print("My age is ", age)
```

In [6]:
```
#Question 2
a = int(input())
b = int(input())
c = int(input())

mul = a * b * c
add = a + b + c

div = mul / add

print(div)
```

```
2
34
4
6.8
```

if elase in python

1. Conditional Statements:

- `if` statements are used for conditional execution in Python.
- They allow you to execute a block of code only if a certain condition is true.

2. Syntax:

- `if` statements have the following basic syntax: if condition:
  ```
  # Code to execute if the condition is true
  ```

3. Indentation:

- Python relies on indentation to define blocks of code.
- The code to be executed under the `if` statement must be indented.

4. `else` Statement:

- The `else` statement is used in conjunction with `if` to execute a block of code when the `if` condition is false.

5. `elif` Statement:

- Short for "else if", the `elif` statement allows checking multiple conditions sequentially after an initial `if`.
- It is used when there are multiple conditions to be evaluated.

6. Nested `if` Statements:

- You can have `if` statements inside other `if` or `else` blocks, creating nested conditional structures.

7. Boolean Expressions:

- Conditions in `if` statements evaluate to boolean values (`True` or `False`).
- Common operators used in conditions include `==` (equality), `!=` (not equal), `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `and` , `or` , `not` , etc.

8. Ternary Conditional Operator:

- Python supports a ternary conditional operator, which allows writing a concise `if-else` statement in a single line.

Example: result = "Pass" if marks >= 50 else "Fail"

9. Indentation Errors:

- Improper indentation can lead to syntax errors or unexpected behavior in your code.

10. Flow Control:

- `if` and `else` statements control the flow of execution based on certain conditions.

In [20]:

```
# Getting user input for age
age = int(input("Enter your age: "))

# Simple if-else statement
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")

# Using if-elif-else for multiple conditions
num = int(input("Enter a number: "))

if num > 0:
    print("Number is positive.")
elif num < 0:
    print("Number is negative.")
else:
    print("Number is zero.")

# Nested if statements
x = 10
y = 5

if x > 5:
    if y > 3:
        print("Both conditions are satisfied.")
    else:
        print("Only first condition is satisfied.")
else:
    print("First condition is not satisfied.")
```

```
Enter your age: 22
You are an adult.
Enter a number: 2
Number is positive.
Both conditions are satisfied.
```

In [19]:
```
if(True):
        print("Yes")
else:
        print("No")
```

Yes

In [20]:
```
print("win" == "win")
```

True

In [16]:
```
print("win" == "Win")
```

False

In [25]:
```
mark = int(input("Enter the mark : "))

if(mark > 35):
        print("Pass")
else:
        print("Fails")
```

Enter the mark : 55
Pass

In [27]:
```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

a and b are equal

In [28]:
```
#Short Hand If ... Else
```

In [30]:
```
a = 2
b = 330
print("A") if a > b else print("B")
```

B

In [33]:
```
a = 10
print(a % 3)
```

1

For loop in python

1. Iterating Over Sequences:

- The `for` loop in Python is primarily used for iterating over sequences like lists, tuples, strings, and dictionaries.

2. Syntax:

- The basic syntax of a `for` loop in Python: for item in sequence:
    `# Code block to execute for each item in the sequence`

3. Iterating Through Sequences:

- The loop iterates through each item in the sequence and executes the code block for each item.

4. `range()` Function:

- The `range()` function generates a sequence of numbers that can be used in `for` loops.
- It's commonly used to iterate a specific number of times.

5. `enumerate()` Function:

- The `enumerate()` function is used to loop over a sequence while keeping track of the index.

6. **Loop Control Statements**:

- `break`: Terminates the loop prematurely if a certain condition is met.
- `continue`: Skips the current iteration and moves to the next iteration of the loop.

7. **Nested** `for` **Loops**:

- You can have one or more `for` loops inside another `for` loop, creating nested iterations.

8. **Iteration Over Dictionary**:

- `for` loops can iterate over keys, values, or key-value pairs in a dictionary using methods like `keys()`, `values()`, or `items()`.

9. **Iterable Objects**:

- Any object that can be iterated over is considered iterable and can be used with a `for` loop.

10. `else` **Clause with** `for` **Loop**:

- Python allows an `else` block after a `for` loop. This block executes after the loop completes without encountering a `break` statement.

Example:

Iterating over a list

fruits = ["apple", "banana", "cherry"] for fruit in fruits: print(fruit)

Using range() in for loop

for i in range(5): print(i)

Using enumerate() in for loop

for index, value in enumerate(fruits): print(f"Index: {index}, Value: {value}")

Nested for loops

for i in range(3): for j in range(2): print(f"({i}, {j})")

In [35]:
```
for i in "apple":
    print(i)
```

```
a
p
p
l
e
```

In [21]:
```
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("Fruit:", fruit)

# Using range() in for loop
for i in range(5):
    print("Number:", i)

# Using enumerate() in for loop
for index, value in enumerate(fruits):
    print("Index:", index, "Value:", value)

# Nested for loops
for i in range(3):
    for j in range(2):
        print("Coordinates:", i, j)

# Iterating over dictionary keys, values, and items
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Iterating over keys
print("Keys:")
for key in person:
    print(key)

# Iterating over values
print("\nValues:")
for value in person.values():
    print(value)

# Iterating over key-value pairs
print("\nKey-Value Pairs:")
for key, value in person.items():
    print(key, ":", value)
```

```
Fruit: apple
Fruit: banana
Fruit: cherry
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Index: 0 Value: apple
Index: 1 Value: banana
Index: 2 Value: cherry
Coordinates: 0 0
Coordinates: 0 1
Coordinates: 1 0
Coordinates: 1 1
Coordinates: 2 0
Coordinates: 2 1
Keys:
name
age
city

Values:
Alice
30
New York

Key-Value Pairs:
name : Alice
age : 30
city : New York
```

In [38]:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
apple
banana
cherry
```

In [ ]:

```
The range() Function
To loop through a set of code a specified number of times, we can use the range() f
The range() function returns a sequence of numbers, starting from 0 by default, and
```

In [40]:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [49]:

```
for i in range(0, 11, 2):
    print(i)
```

```
0
2
4
6
8
10
```

In [55]:

```
#Print multiplication table
```

```python
for i in range(1, 11):
    print(i , "x 2", "=", i * 2)
```

```
1 x 2 = 2
2 x 2 = 4
3 x 2 = 6
4 x 2 = 8
5 x 2 = 10
6 x 2 = 12
7 x 2 = 14
8 x 2 = 16
9 x 2 = 18
10 x 2 = 20
```

In [56]:

```python
#Excercises
```

In [63]:

```python
sum = 0

for i in range(2, 11, 2):
    sum = sum + 1
print(sum)

sum = 0

for i in range(1, 11):
    if i % 2 == 0:
        sum = sum + 1
print(sum)
```

```
5
5
```

In [ ]:

```python
numArray = []
sum = 0

for i in range(5):
    num = int(input())
    numArray.append(num)

    sum = sum + numArray[i]
print("Total:", sum)for i in range(5):
    print(i)
```

Python While Loops

1. **Repetitive Execution**:

- `while` loops are used for executing a block of code repeatedly as long as a specified condition is true.

2. **Syntax**:

- The basic syntax of a `while` loop in Python: while condition:
  ```
  # Code block to execute as long as the condition is true
  ```

3. **Loop Continuation**:

- The loop continues to execute as long as the condition remains true.

- It checks the condition before each iteration, and if the condition becomes false, the loop stops.

4. **Infinite Loops**:

- If the condition in a `while` loop always evaluates to `True`, it results in an infinite loop.

5. **Loop Control Statements**:

- `break`: Terminates the loop prematurely based on a certain condition.
- `continue`: Skips the current iteration and continues to the next iteration of the loop.

6. **Initializing and Updating Variables**:

- Usually, a variable is initialized before the `while` loop and updated within the loop to control the loop's behavior.

7. **Nested `while` Loops**:

- Similar to `for` loops, you can have one or more `while` loops inside another `while` loop, creating nested iterations.

8. **Looping Through User Input**:

- `while` loops are commonly used when expecting user input and validating against certain conditions.

9. **`else` Clause with `while` Loop**:

- Python allows an `else` block after a `while` loop. This block executes when the loop condition becomes false.

10. **Precautions with Infinite Loops**:

- When using `while` loops, ensure there's a mechanism to make the condition eventually become false to avoid infinite loops.

Example:

Simple while loop

counter = 0 while counter < 5: print("Counter:", counter) counter += 1

Using break to exit loop

num = 1 while True: print("Number:", num) if num == 5: break num += 1

Using continue to skip iterations

i = 0 while i < 5: i += 1 if i == 3: continue print("Value of i:", i)

In [78]:
```
i = 0

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

```
0
1
2
3
4
```

In [80]:
```
i = 1
while i < 6:
  print(i)
  i += 1 #Assignment operator
```

```
1
2
3
4
5
```

In [22]:
```
# Simple while loop
counter = 0
while counter < 5:
    print("Counter:", counter)
    counter += 1

# Using break to exit loop
num = 1
while True:
    print("Number:", num)
    if num == 5:
        break
    num += 1

# Using continue to skip iterations
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print("Value of i:", i)
```

```
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Value of i: 1
Value of i: 2
Value of i: 4
Value of i: 5
```

Python Collections

1. List:

- **Ordered**: Lists maintain the order of elements as they are inserted.
- **Mutable**: Elements in a list can be added, removed, or modified after creation.
- **Allows Duplicates**: A list can contain duplicate elements.
- **Creation**: Lists are created using square brackets `[]`.

  my_list = [1, 2, 3, 'apple']

2. Tuple:

- **Ordered**: Tuples, like lists, maintain order.
- **Immutable**: Once created, elements in a tuple cannot be changed.
- **Allows Duplicates**: Similar to lists, tuples can contain duplicate elements.
- **Creation**: Tuples are created using parentheses `()`.

  my_tuple = (1, 2, 'apple')

3. Set:

- **Unordered**: Sets don’t maintain order.
- **Unique Elements**: Sets contain only unique elements; duplicates are automatically removed.
- **Mutable (for the elements)**: Elements can be added or removed after creation, but the set itself (as a whole) is immutable.
- **Creation**: Sets are created using curly braces `{}` or the `set()` function.

  my_set = {1, 2, 3}

4. Dictionary:

- **Key-Value Pairs**: Dictionaries store data in key-value pairs.
- **Unordered**: Prior to Python 3.7, dictionaries didn't maintain the order of elements, but in modern Python versions, they maintain insertion order.
- **Keys are Unique**: Each key in a dictionary must be unique.
- **Mutable**: Elements (key-value pairs) can be added, removed, or modified after creation.
- **Creation**: Dictionaries are created using curly braces `{}` with key-value pairs separated by colons `:`.

  my_dict = {'name': 'Alice', 'age': 30}

1. List Methods:

- **Appending and Extending**:

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

    my_list.append(4)

- `extend()` : Appends elements from another list to the end of the list.

  another_list = [5, 6] my_list.extend(another_list)

- **Inserting and Deleting**:
  - `insert()` : Inserts an element at a specified index.

    my_list.insert(1, 'new_element')

  - `remove()` : Removes the first occurrence of a specified value.

    my_list.remove('new_element')

  - `pop()` : Removes and returns an element at a specified index.

    popped_element = my_list.pop(2)

- **Indexing and Slicing**:
  - Access elements by index: `my_list[index]`
  - Slicing: `my_list[start:end:step]`
- **Other Operations**:
  - `index()` : Returns the index of the first occurrence of a value.
  - `count()` : Returns the number of occurrences of a value.
  - `sort()` : Sorts the list.
  - `reverse()` : Reverses the order of elements in the list.

2. Tuple Operations:

- **Accessing Elements**: Similar to lists, accessed by index.
- **Immutable**: Cannot be modified after creation.

3. Set Methods:

- **Adding and Removing**:
  - `add()` : Adds an element to the set.

    my_set.add(4)

  - `remove()` : Removes a specified element from the set. Raises an error if the element doesn't exist.

    my_set.remove(4)

  - `discard()` : Removes a specified element from the set if it exists; otherwise, does nothing.

    my_set.discard(4)

- **Operations**:
  - `union()` : Returns the union of two sets.

- `intersection()` : Returns the intersection of two sets.
- `difference()` : Returns the difference between two sets.
- `update()` : Updates the set with the union of itself and others.

4. **Dictionary Methods:**

- **Adding, Updating, and Deleting**:

  - `update()` : Adds key-value pairs from another dictionary or iterable.

    my_dict.update({'new_key': 'new_value'})

  - `pop()` : Removes and returns the value for a specified key.

    popped_value = my_dict.pop('new_key')

  - `del` : Deletes an item by key.

    del my_dict['new_key']

- **Accessing Values**:

  - Access values by key: `my_dict[key]`
  - `keys()` : Returns a view of all keys in the dictionary.
  - `values()` : Returns a view of all values in the dictionary.
  - `items()` : Returns a view of all key-value pairs as tuples.

In [ ]:

```
#List
```

In [99]:

```
a = [1, 2, 3, 4, 5]

print("Print list : ", a)
print("Print list element : ", a[2])
print("Remove last elemnt : ", a.pop())
print("Print list : ", a)
print("Remove first elemnt : ", a.pop(0))
print("Print list : ", a)
a.insert(0, 11)
print("Print list : ", a)
```

```
Print list :  [1, 2, 3, 4, 5]
Print list element :  3
Remove last elemnt :  5
Print list :  [1, 2, 3, 4]
Remove first elemnt :  1
Print list :  [2, 3, 4]
Print list :  [11, 2, 3, 4]
```

In [23]:

```
# List creation and basic operations
my_list = [1, 2, 3, 4]

# Appending and extending a list
my_list.append(5)
print("Appended:", my_list)

extension = [6, 7]
my_list.extend(extension)
print("Extended:", my_list)
```

```python
# Inserting and deleting elements
my_list.insert(2, 'inserted')
print("Inserted:", my_list)

my_list.remove('inserted')
print("Removed:", my_list)

popped_value = my_list.pop(3)
print("Popped:", popped_value, "List:", my_list)

# Indexing and slicing
print("First element:", my_list[0])
print("Sliced:", my_list[1:4])

# Other list operations
print("Index of 3:", my_list.index(3))
print("Count of 4:", my_list.count(4))

my_list.sort()
print("Sorted:", my_list)

my_list.reverse()
print("Reversed:", my_list)
```

```
Appended: [1, 2, 3, 4, 5]
Extended: [1, 2, 3, 4, 5, 6, 7]
Inserted: [1, 2, 'inserted', 3, 4, 5, 6, 7]
Removed: [1, 2, 3, 4, 5, 6, 7]
Popped: 4 List: [1, 2, 3, 5, 6, 7]
First element: 1
Sliced: [2, 3, 5]
Index of 3: 2
Count of 4: 0
Sorted: [1, 2, 3, 5, 6, 7]
Reversed: [7, 6, 5, 3, 2, 1]
```

In [ ]:

```python
#Python Tuples
```

In [106…

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


x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
('apple', 'banana', 'cherry')
('apple', 'kiwi', 'cherry')
```

In [107…

```python
#Python Sets
```

In [109…

```python
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

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

In [26]:

```python
# Tuple creation
my_tuple = (1, 2, 3, 'apple')
```

```python
# Accessing elements by index
print("First element:", my_tuple[0])

# Slicing
print("Sliced elements:", my_tuple[1:3])

# Tuple unpacking
a, b, c, d = my_tuple
print("Unpacked values:", a, b, c, d)

# Concatenating tuples
another_tuple = (4, 5)
concatenated_tuple = my_tuple + another_tuple
print("Concatenated tuple:", concatenated_tuple)

# Tuple repetition
repeated_tuple = my_tuple * 2
print("Repeated tuple:", repeated_tuple)

# Length of tuple
print("Length of tuple:", len(my_tuple))
```

```
First element: 1
Sliced elements: (2, 3)
Unpacked values: 1 2 3 apple
Concatenated tuple: (1, 2, 3, 'apple', 4, 5)
Repeated tuple: (1, 2, 3, 'apple', 1, 2, 3, 'apple')
Length of tuple: 4
```

In [24]:

```python
# Set creation and basic operations
my_set = {1, 2, 3, 4}

# Adding and removing elements
my_set.add(5)
print("Added:", my_set)

my_set.discard(3)
print("Discarded:", my_set)

# Set operations
another_set = {3, 4, 5, 6}

union_set = my_set.union(another_set)
print("Union:", union_set)

intersection_set = my_set.intersection(another_set)
print("Intersection:", intersection_set)

difference_set = my_set.difference(another_set)
print("Difference:", difference_set)
```

```
Added: {1, 2, 3, 4, 5}
Discarded: {1, 2, 4, 5}
Union: {1, 2, 3, 4, 5, 6}
Intersection: {4, 5}
Difference: {1, 2}
```

In [110…

```python
#Python Dictionaries
```

In [112…

```python
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
```

```
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

In [25]:
```
# Dictionary creation and basic operations
my_dict = {'name': 'Alice', 'age': 30}

# Adding, updating, and deleting items
my_dict['city'] = 'New York'
print("Updated:", my_dict)

my_dict.update({'age': 31})
print("Age updated:", my_dict)

popped_value = my_dict.pop('age')
print("Popped age:", popped_value, "Dict:", my_dict)

# Accessing values and keys
print("Value for 'name':", my_dict['name'])
print("Keys:", my_dict.keys())
print("Values:", my_dict.values())
```

```
Updated: {'name': 'Alice', 'age': 30, 'city': 'New York'}
Age updated: {'name': 'Alice', 'age': 31, 'city': 'New York'}
Popped age: 31 Dict: {'name': 'Alice', 'city': 'New York'}
Value for 'name': Alice
Keys: dict_keys(['name', 'city'])
Values: dict_values(['Alice', 'New York'])
```

Functions in Python

1. Definition:

- Functions are blocks of reusable code designed to perform a specific task.
- They enhance code readability, reusability, and modularity.

2. Syntax:

- A basic function in Python:

  def function_name(parameters):

  ```
  # Code block
  return value  # Optional, returns a value
  ```

3. Parameters and Arguments:

- **Parameters**: Variables listed in a function's definition.
- **Arguments**: Values passed into a function when it is called.

4. Return Statement:

- `return` : Ends the function's execution and optionally returns a value to the caller.
- If no `return` statement is specified, the function returns `None` by default.

5. Scope:

- Functions create a local scope, meaning variables defined within a function are not accessible outside it (unless specified otherwise).

6. Function Calling:

- Functions are called by using their name followed by parentheses `()` containing arguments (if any).
- Example:

  result = function_name(arg1, arg2)

7. Types of Functions:

- **Built-in Functions**: Predefined functions available in Python (e.g., `print()`, `len()`, `max()`).
- **User-defined Functions**: Functions defined by users to perform specific tasks.

8. Default Arguments:

- Parameters in a function can have default values.
- When an argument isn't passed for a parameter with a default value, the default value is used.

9. Variable-Length Arguments:

- Functions can accept a variable number of arguments using `*args` and `**kwargs`.
- `*args` collects positional arguments into a tuple.
- `**kwargs` collects keyword arguments into a dictionary.

10. Recursion:

- Functions can call themselves, enabling a technique known as recursion.
- Recursion is useful for solving problems that can be broken down into smaller, similar sub-problems.

In [27]:

```
# Simple function without arguments and return value
def greet():
    print("Hello, there!")

# Function with arguments and return value
def add_numbers(a, b):
    return a + b

# Function with default argument
def greet_person(name="Guest"):
    print(f"Hello, {name}!")

# Function with variable-length arguments
def calculate_total(*args):
    total = sum(args)
    return total
```

```python
# Recursive function to calculate factorial
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Calling functions
greet()  # Output: Hello, there!

result = add_numbers(5, 3)
print("Sum:", result)  # Output: Sum: 8

greet_person("Alice")  # Output: Hello, Alice!
greet_person()  # Output: Hello, Guest!

total = calculate_total(10, 20, 30, 40)
print("Total:", total)  # Output: Total: 100

fact = factorial(5)
print("Factorial of 5:", fact)  # Output: Factorial of 5: 120
```

```
Hello, there!
Sum: 8
Hello, Alice!
Hello, Guest!
Total: 100
Factorial of 5: 120
```

In [117…

```python
def helloWorld():
    print("Hello world")

helloWorld()
```

```
Hello world
```

In [118…

```python
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

In [120…

```python
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

In [129…

```python
#Return Keyword in Python

def add():
    return 1 + 2

result = add()

print(result)

def name():
    return "Vithu"
```

```
name()

getName = name()

print(getName)
```

```
3
Vithu
```

Classes and Objects

1. **Classes**:

- A class is a blueprint for creating objects (instances).
- It defines the properties (attributes) and behaviors (methods) that objects of the class will have.
- Classes encapsulate data and functionality into a single unit.

2. **Objects (Instances)**:

- Objects are instances of classes.
- Each object created from a class has its own unique set of attributes and can perform actions (methods) defined in the class.

3. **Attributes**:

- Attributes are variables that belong to a class or an object.
- They represent the state of the object and can be data variables or class variables.

4. **Methods**:

- Methods are functions defined within a class.
- They define the behavior or actions that objects of the class can perform.
- Methods can interact with attributes of the class.

5. **Constructor (`__init__`)**:

- `__init__` is a special method used to initialize object attributes when the object is created.
- It's called automatically when creating a new instance of the class (`__init__(self, ...)`).

6. **Encapsulation**:

- Classes help in encapsulating data and methods together.
- Access to attributes and methods can be controlled using access specifiers (`public`, `private`, `protected`).

7. **Inheritance**:

- Inheritance allows a new class (child class) to inherit properties and methods from an existing class (parent class).
- The child class can add new attributes or methods or override existing ones.

8. **Polymorphism**:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- It enables a single interface to be used for different data types or classes.

9. **Abstraction**:

- Abstraction hides complex implementation details and only shows essential features of an object.
- It allows the user to focus on what an object does, rather than how it does it.

10. **Class and Instance Variables**:

- Class variables are shared by all instances of the class.
- Instance variables are unique to each instance of the class.

11. **Destructor (`__del__`)**:

- `__del__` is a special method used to perform clean-up operations before an object is destroyed.

12. **Operator Overloading**:

- Allows defining custom behavior for operators such as `+`, `-`, `*`, `/`, etc., for objects of a class.

In [28]:

```
# Creating a class
class Animal:
    # Class attribute
    species = "Mammal"

    # Constructor
    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

    # Instance method
    def make_sound(self):
        pass  # Placeholder method

# Creating a subclass (inheritance)
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Creating another subclass (inheritance)
class Cat(Animal):
    def make_sound(self):
```

```
        return "Meow!"

# Creating objects (instances)
dog = Dog("Buddy", 3)
cat = Cat("Whiskers", 5)

# Accessing attributes and calling methods
print(f"{dog.name} is {dog.age} years old.")
print(f"{cat.name} is {cat.age} years old.")
print(f"{dog.name} says: {dog.make_sound()}")
print(f"{cat.name} says: {cat.make_sound()}")

# Checking class attributes
print(f"{dog.name} is a {dog.species}")
print(f"{cat.name} is a {cat.species}")
```

```
Buddy is 3 years old.
Whiskers is 5 years old.
Buddy says: Woof!
Whiskers says: Meow!
Buddy is a Mammal
Whiskers is a Mammal
```

In [146…

```
class Love:
        fullName = ""

        def vithu(self):
                print("I love you Nila...")
        def nila(self):
                print("I love you Vithu...")

#Create object to acccess class properties
baby1 = Love()
baby2 = Love()

baby1.vithu()
baby2.nila()

baby1.fullName = "Vithu Baby"
print(baby1.fullName)

baby2.fullName = "Nila Baby"
print(baby2.fullName)
```

```
I love you Nila...
I love you Vithu...
Vithu Baby
Nila Baby
```

Constructor and Self Keyword

1. Constructor (`__init__`):

- The constructor method `__init__` is a special method in Python classes.
- It gets called automatically when an object is created from a class.
- It's used to initialize the object's attributes with initial values.
- It allows for setting up the object's state upon creation.

2. **Purpose of the Constructor:**

- Initializing instance variables: `__init__` initializes the object's attributes with the values passed during object creation.
- Setting up the initial state of the object: It helps define what attributes an object will have and their starting values.

3. Syntax:

- The `__init__` method takes at least one parameter, commonly named `self`, which refers to the object itself.
- `self` is passed implicitly when calling methods or accessing attributes within the class.

4. Self Keyword:

- `self` is the conventional name used for the first parameter of instance methods in Python classes.
- It refers to the instance of the class itself and allows access to its attributes and methods within the class.
- It distinguishes between the instance's attributes and local variables within methods.

5. Use of `Self`:

- Inside the `__init__` method and other instance methods, `self` is used to access and modify the object's attributes.
- It is not a reserved keyword in Python but a convention to use `self` as the first parameter.

6. Example:

class Person: def **init**(self, name, age): self.name = name self.age = age

```
def display_info(self):
    print(f"Name: {self.name}, Age: {self.age}")
```

Creating an object of the class Person

person1 = Person("Alice", 30) person1.display_info() # Output: Name: Alice, Age: 30

7. Implicit Invocation:

- When you create an instance of a class (`object = Class()`), Python implicitly passes the object itself (`self`) to the `__init__` method.
- Therefore, you don't need to explicitly pass `self` when creating objects; it's done automatically by Python.

In [29]:
```
class Person:
    def __init__(self, name, age):
        # Initializing instance variables using the constructor
        self.name = name
```

```
        self.age = age

    def display_info(self):
        # Accessing instance variables using self within a method
        print(f"Name: {self.name}, Age: {self.age}")

# Creating an object of the class Person
person1 = Person("Alice", 30)
person1.display_info()  # Output: Name: Alice, Age: 30

# Creating another object of the class Person
person2 = Person("Bob", 25)
person2.display_info()  # Output: Name: Bob, Age: 25
```

```
Name: Alice, Age: 30
Name: Bob, Age: 25
```

In [1]:
```
class Laptop:
    def __init__(self): #wich mean constructure it will call automatically when cre
        self.price = 0
        self.ram = ""
        self.processor = ""

    def display(self):
        print("Ram:", self.ram)
        print("Processor:", self.processor)



hp = Laptop()

hp.price = 160000
hp.ram = "32GB"
hp.processor = "i9"


hp.display()

#Using self we can denote current object
```

```
Ram: 32GB
Processor: i9
```

Types of Class Variable

1. **Instance Variables:**

- Variables that are specific to each instance of a class.
- Defined within methods and initialized using the `self` keyword inside the class's `__init__` method.
- Example:

  class MyClass: def **init**(self, var):

  ```
  self.instance_var = var
  ```

2. **Class Variables:**

- Variables that are shared among all instances of a class.

- Defined within the class but outside of any methods.
- Accessed using the class name or an instance.
- Example:

  class MyClass: class_var = "Shared among all instances"

  def **init**(self, var):

  ```
  self.instance_var = var
  ```

- Changes to class variables reflect across all instances:

  ```
  obj1 = MyClass("First")
  obj2 = MyClass("Second")

  print(obj1.class_var)  # Accessing class variable via instance
  print(obj2.class_var)

  MyClass.class_var = "Modified for all"

  print(obj1.class_var)  # All instances reflect the change
  print(obj2.class_var)
  ```

In [ ]:

```
#Instance variables
```

In [30]:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name  # Instance variable
        self.salary = salary  # Instance variable

    def display_info(self):
        print(f"Name: {self.name}, Salary: {self.salary}")

# Creating instances of Employee
emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

# Accessing instance variables
emp1.display_info()  # Output: Name: Alice, Salary: 50000
emp2.display_info()  # Output: Name: Bob, Salary: 60000
```

```
Name: Alice, Salary: 50000
Name: Bob, Salary: 60000
```

In [11]:

```
class phone:
    def __init__(self, brand, price, chargerType):
        self.brand = brand
        self.price = price
        self.chargerType = chargerType

    def display(self):
        print("Brand : ", self.brand)
        print("Price : ", self.price)
        print("Charger Type : ", self.chargerType)


samsung = phone("Samsung", 45000, "Type-C")

samsung.display()
```

```
Brand :  Samsung
Price :  45000
Charger Type :  Type-C
```

In [ ]:
```
#Class variables
```

In [31]:
```
class Employee:
    company = "XYZ Corp"  # Class variable

    def __init__(self, name, salary):
        self.name = name  # Instance variable
        self.salary = salary  # Instance variable

    def display_info(self):
        print(f"Name: {self.name}, Salary: {self.salary}, Company: {Employee.compar

# Creating instances of Employee
emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

# Accessing class variable using class name and instance
print(emp1.company)  # Output: XYZ Corp
print(emp2.company)  # Output: XYZ Corp

# Accessing class variable through a method
emp1.display_info()  # Output: Name: Alice, Salary: 50000, Company: XYZ Corp
emp2.display_info()  # Output: Name: Bob, Salary: 60000, Company: XYZ Corp
```

```
XYZ Corp
XYZ Corp
Name: Alice, Salary: 50000, Company: XYZ Corp
Name: Bob, Salary: 60000, Company: XYZ Corp
```

In [15]:
```
class phone:
    chargerType = "Type-C"

    def __init__(self, brand, price):
        self.brand = brand
        self.price = price

    def display(self):
        print("Brand : ", self.brand)
        print("Price : ", self.price)
        print("Charger Type : ", self.chargerType)


samsung = phone("Samsung", 45000)
samsung.display()

nokia = phone("Nokia", 45000)
nokia.display()

google = phone("Google", 85000)
google.display()

phone.chargerType = "Type-B" #Modify the class variables

apple = phone("Apple", 185000)
apple.display()
```

```
Brand :  Samsung
Price :  45000
Charger Type :  Type-C
Brand :  Nokia
Price :  45000
Charger Type :  Type-C
Brand :  Google
Price :  85000
Charger Type :  Type-C
Brand :  Apple
Price :  185000
Charger Type :  Type-B
```

Types of Class Methods

1. **Instance Methods**:

- **Definition**: Instance methods are regular methods defined inside a class and operate on instances of that class.
- **Access**: They automatically take the instance (self) as the first parameter.
- **Purpose**: They can access and modify instance variables, as well as perform actions related to individual instances.
- **Example**: class MyClass:
  ```
  def instance_method(self):
      # Access instance variables using self
      self.value = 10
  ```

2. **Class Methods**:

- **Definition**: Class methods are methods that operate on the class itself rather than instances.
- **Decorator**: Defined using `@classmethod` decorator and take `cls` (class) as the first parameter.
- **Purpose**: Used for operations that involve the class itself or class variables shared among all instances.
- **Example**: class MyClass:

  ```
  class_variable = "class_variable"

  @classmethod
  def class_method(cls):
      return cls.class_variable
  ```

3. **Static Methods**:

- **Definition**: Static methods are independent of class and instance variables.
- **Decorator**: Defined using `@staticmethod` decorator.
- **Purpose**: Used when a method does not require access to instance or class variables within the class.
- **Example**: class MyClass:
  ```
  @staticmethod
  def static_method():
  ```

```
        return "This is a static method"
```

4. **Special Note:**

- `self`, `cls`, and `@staticmethod`:
  - `self`: Refers to the instance of the class in instance methods.
  - `cls`: Refers to the class itself in class methods.
  - `@staticmethod`: Decorator used to define static methods, making them independent of class and instance variables.

In [34]:
```python
#Instance Method
class MyClass:
    def instance_method(self):
        return "This is an instance method"

# Creating an instance of MyClass
obj = MyClass()

# Calling the instance method
print(obj.instance_method())  # Output: This is an instance method
```

```
This is an instance method
```

In [36]:
```python
#Static method
class MyClass:
    @staticmethod
    def static_method():
        return "This is a static method"

# Calling the static method
print(MyClass.static_method())  # Output: This is a static method
```

```
This is a static method
```

In [38]:
```python
#Class Method
class MyClass:
    class_variable = "class_variable"

    @classmethod
    def class_method(cls):
        return cls.class_variable

# Accessing class method using class name
print(MyClass.class_method())  # Output: class_variable
```

```
class_variable
```

Inheritance and its type

1. **Inheritance:**

- Inheritance is a feature in object-oriented programming that allows a new class (subclass/derived class) to inherit properties and behaviors (methods and attributes) from an existing class (superclass/base/parent class).
- It facilitates code reusability and promotes the creation of a hierarchical structure among classes.

2. **Types of Inheritance**:

a. **Single Inheritance**:

- A subclass inherits from only one superclass.
- It forms a linear hierarchy.
- Example: class Parent:

```
    pass
```

  class Child(Parent):

```
    pass
```

b. **Multiple Inheritance**:

- A subclass inherits from multiple superclasses.
- It allows inheriting attributes and methods from multiple classes.
- Example: class ClassA:

```
    pass
```

  class ClassB:

```
    pass
```

  class Child(ClassA, ClassB):

```
    pass
```

c. **Multilevel Inheritance**:

- A subclass inherits from another subclass.
- It forms a chain of inheritance.
- Example: class Grandparent:

```
    pass
```

  class Parent(Grandparent):

```
    pass
```

  class Child(Parent):

```
    pass
```

d. **Hierarchical Inheritance**:

- Multiple subclasses inherit from a single superclass.
- Each subclass has its own set of attributes and methods.
- Example: class Parent:

```
    pass
```

  class Child1(Parent):

pass

class Child2(Parent):

pass

e. **Hybrid Inheritance:**

- Combination of different types of inheritance.
- Example: class ClassA:

  pass

  class ClassB(ClassA):

  pass

  class ClassC:

  pass

  class ClassD(ClassB, ClassC):

  pass

3. **Inheritance Terminology:**

- **Superclass/Parent/Base Class**: The class whose properties and behaviors are inherited.
- **Subclass/Derived Class/Child Class**: The class that inherits from a superclass.
- **Method Overriding**: Redefining a method in the subclass that already exists in the superclass.

4. **Use of `super()`:**

- `super()` is used to access the methods and properties of a superclass from a subclass.

5. **Advantages:**

- Code Reusability: Avoids redundant code by inheriting attributes and methods.
- Modularity: Divides the code into manageable and organized parts.

In [40]:
```
#Single Inheritance
class Parent:
    def show_parent(self):
        return "Parent class method"

class Child(Parent):
    def show_child(self):
        return "Child class method"

# Creating instances and accessing methods
parent = Parent()
child = Child()

print(parent.show_parent())  # Output: Parent class method
```

```
print(child.show_parent())    # Output: Parent class method
print(child.show_child())     # Output: Child class method
```

```
Parent class method
Parent class method
Child class method
```

In [55]:
```
class dad:
    def phone(self):
        print("Dads phone")

class son(dad):
    def laptop(self):
        print("Sons phone")

vithu = son()
vithu.phone()
```

```
Dads phone
```

In [47]:
```
#Multiple Inheritance
class ClassA:
    def method_a(self):
        return "Method from ClassA"

class ClassB:
    def method_b(self):
        return "Method from ClassB"

class ClassC(ClassA, ClassB):
    def method_c(self):
        return "Method from ClassC"

# Creating instances and accessing methods
obj = ClassC()

print(obj.method_a())  # Output: Method from ClassA
print(obj.method_b())  # Output: Method from ClassB
print(obj.method_c())  # Output: Method from ClassC
```

```
Method from ClassA
Method from ClassB
Method from ClassC
```

In [48]:
```
class dad:
    def phone(self):
        print("Dads phone")

class mom:
    def sweet(self):
        print("Moms sweet")

class son(dad, mom):
    def laptop(self):
        print("Sons phone")

vithu = son()
vithu.phone()
vithu.sweet()
```

```
Dads phone
Moms sweet
```

In [42]:
```
#Multilevel Inheritance
class Grandparent:
```

```python
    def method_gp(self):
        return "Method from Grandparent"

class Parent(Grandparent):
    def method_p(self):
        return "Method from Parent"

class Child(Parent):
    def method_c(self):
        return "Method from Child"

# Creating instances and accessing methods
obj = Child()

print(obj.method_gp())  # Output: Method from Grandparent
print(obj.method_p())   # Output: Method from Parent
print(obj.method_c())   # Output: Method from Child
```

```
Method from Grandparent
Method from Parent
Method from Child
```

In [56]:

```python
class grandpa:
    def phone(self):
        print("grandpa phone")

class dad(grandpa):
    def money(self):
        print("Dads money")

class son(dad):
    def laptop(self):
        print("sons laptop")

vithu = son()
vithu.laptop()
vithu.money()

d1 = dad()
d1.phone()

vithu.phone() #note it
```

```
sons laptop
Dads money
grandpa phone
grandpa phone
```

In [44]:

```python
#Hierarchical Inheritance
class Parent:
    def method_p(self):
        return "Method from Parent"

class Child1(Parent):
    def method_c1(self):
        return "Method from Child1"

class Child2(Parent):
    def method_c2(self):
        return "Method from Child2"

# Creating instances and accessing methods
obj1 = Child1()
obj2 = Child2()
```

```python
print(obj1.method_p())  # Output: Method from Parent
print(obj1.method_c1())  # Output: Method from Child1
print(obj2.method_p())  # Output: Method from Parent
print(obj2.method_c2())  # Output: Method from Child2
```

```
Method from Parent
Method from Child1
Method from Parent
Method from Child2
```

In [57]:

```python
class dad:
    def money(self):
        print("Dads money")

class son1(dad):
    pass
class son2(dad):
    pass
class son3(dad):
    pass

vithu = son2()
vithu.money()
```

```
Dads money
```

In [58]:

```python
#Hybrid inheritance
class ClassA:
    def method_a(self):
        return "Method from ClassA"

class ClassB(ClassA):
    def method_b(self):
        return "Method from ClassB"

class ClassC:
    def method_c(self):
        return "Method from ClassC"

class ClassD(ClassB, ClassC):
    def method_d(self):
        return "Method from ClassD"

# Creating instances and accessing methods
obj = ClassD()

print(obj.method_a())  # Output: Method from ClassA
print(obj.method_b())  # Output: Method from ClassB
print(obj.method_c())  # Output: Method from ClassC
print(obj.method_d())  # Output: Method from ClassD
```

```
Method from ClassA
Method from ClassB
Method from ClassC
Method from ClassD
```

In [ ]:

```python
class dad:
    def money(self):
        print("Dads money")

class land():
    def important(self):
        print("important land")

class son1(dad, land):
```

```
    pass
class son2(dad):
    pass
class son3(dad):
    pass

vithu = son2()
vithu.money()
```

Super Keyword in Python

1. Purpose:

- **Accessing Superclass Methods**: `super()` allows a subclass to invoke methods from its superclass.
- **Avoiding Hardcoding**: It provides a dynamic way to call methods in the superclass without hardcoding the superclass name.

2. Syntax:

- The general syntax for using `super()` is: `super().method()`.
- In Python 3, you can simply use `super().method()` without passing the class name or self explicitly.

3. Usage:

- **Calling Superclass Methods**: Used to call a method defined in the superclass from a method in the subclass.
- **Initiating the Parent Class**: Often used in the `__init__` method of a subclass to initialize attributes from the parent class.

4. `super()` with Multiple Inheritance:

- In multiple inheritance scenarios, `super()` helps in invoking methods of the superclass in the method resolution order (MRO).
- It ensures that the method called is from the next class in the MRO sequence.

5. Example:

class Parent: def show(self): return "Hello from Parent"

class Child(Parent): def show(self): return super().show() + ", and Child"

Creating an instance of Child

obj = Child()

print(obj.show()) # Output: Hello from Parent, and Child

In [59]:
```python
class Parent:
    def show(self):
        return "Hello from Parent"

class Child(Parent):
    def show(self):
        # Calling superclass method using super()
        return super().show() + ", and Child"

# Creating an instance of Child
obj = Child()

# Accessing the overridden method
print(obj.show())  # Output: Hello from Parent, and Child
```

```
Hello from Parent, and Child
```

In [72]:
```python
class a():
    def __init__(self):
        print("A")

    def display(self):
        print("you are class a")

class b(a):
    def __init__(self):
        super().__init__() #class the parent class constructure
        print("B")

    def display(self):
        print("you are class b")

obj = b()
```

```
A
B
```

Polymorphism in Python

1. Definition:

- **Polymorphism** refers to the ability of different objects to be used in a similar way, even if they belong to different classes.
- It allows methods to be written to handle objects of various classes that have a common interface or superclass.

2. Types of Polymorphism:

a. Compile-Time Polymorphism (Static Binding):

- **Method Overloading**: Defining multiple methods with the same name but different parameters in the same class.
- Python does not support traditional method overloading based on different argument types due to its dynamic nature. However, it does support default arguments and variable-length arguments.

b. Run-Time Polymorphism (Dynamic Binding):

- **Method Overriding**: Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
- It allows a subclass to provide a specialized implementation of a method that is already provided by one of its parent classes.
- Achieved using inheritance and the `super()` keyword to call methods from the superclass.

3. **Example of Polymorphism:**

class Animal: def make_sound(self): pass

class Dog(Animal): def make_sound(self): return "Woof!"

class Cat(Animal): def make_sound(self): return "Meow!"

Function using polymorphism

def animal_sound(animal): return animal.make_sound()

Creating instances

dog = Dog() cat = Cat()

Calling the function with different objects

print(animal_sound(dog)) # Output: Woof! print(animal_sound(cat)) # Output: Meow!

In the example, `animal_sound()` can take any object that has a `make_sound()` method, allowing the function to work with both `Dog` and `Cat` objects without changes, showcasing the polymorphic behavior of Python.

4. **Advantages:**

- **Code Flexibility**: Allows using objects of different classes interchangeably, enhancing code flexibility.
- **Code Reusability**: Methods written to handle a superclass can be reused for its subclasses.

In [73]:

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Function using polymorphism
def animal_sound(animal):
    return animal.make_sound()
```

```
# Creating instances
dog = Dog()
cat = Cat()

# Calling the function with different objects
print(animal_sound(dog))  # Output: Woof!
print(animal_sound(cat))  # Output: Meow!
```

```
Woof!
Meow!
```

In [78]:
```
def add(a,b,c=0):
    print(a+b+c)

add(1,2)
add(1,2,3)
```

```
3
6
```

Encapsulation and Access Modifiers

1. **Encapsulation**:

- **Encapsulation** is a fundamental principle in object-oriented programming.
- It involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit (class).
- It restricts access to some of the object's components, hiding the internal state and requiring interaction through well-defined interfaces.
- Encapsulation helps in achieving data abstraction, data hiding, and modularity in code.

2. **Access Modifiers**:

- Access modifiers define the accessibility or visibility of class members (attributes and methods) from outside the class.
- In Python, there are no explicit keywords like `private`, `public`, or `protected` as in some other languages, but there are conventions and techniques to achieve similar behavior.

3. **Access Levels in Python**:

a. **Public**:

- By default, all attributes and methods in a class are public.
- They can be accessed from outside the class.

b. **Private**:

- Python uses a convention to denote private attributes or methods by prefixing them with a single underscore `_`.
- They are intended to be private and should not be accessed directly from outside the class.
- Example: class MyClass:

```
    def __init__(self):
        self._private_var = 10  # Private variable
```

c. **Protected**:

- Python uses a convention to denote protected attributes or methods by prefixing them with a double underscore `__`.
- They are intended to be protected, although it's more about name mangling rather than strict access control.
- Example: class MyClass:
```
    def __init__(self):
        self.__protected_var = 20  # Protected variable
```

4. **Accessing Encapsulated Members**:

- **Getter and Setter Methods**: Used to access and modify private or protected attributes indirectly.
- **Property Decorators**: In Python, `@property` and `@<attribute_name>.setter` decorators are used to define getters and setters, allowing controlled access to attributes.

5. **Purpose**:

- Encapsulation and access modifiers help in maintaining the integrity of the data within a class by controlling its access from outside, promoting data security and preventing accidental modification.

In [79]:
```python
class Car:
    def __init__(self, brand, model, price):
        self.brand = brand  # Public attribute
        self.model = model  # Public attribute
        self.__price = price  # Private attribute

    def get_price(self):
        return self.__price

    def set_price(self, new_price):
        if new_price > 0:
            self.__price = new_price

# Creating an instance of Car
car = Car("Toyota", "Corolla", 25000)

# Accessing public attributes directly
print("Car Brand:", car.brand)  # Output: Toyota
print("Car Model:", car.model)  # Output: Corolla

# Accessing private attribute through getter method
print("Car Price:", car.get_price())  # Output: 25000

# Trying to access private attribute directly
# This will not work as it's private and not accessible directly outside the class
# print("Car Price:", car.__price)  # Throws an AttributeError

# Changing private attribute using setter method
car.set_price(30000)
```

```python
# Accessing updated private attribute through getter method
print("Updated Car Price:", car.get_price())  # Output: 30000
```

```
Car Brand: Toyota
Car Model: Corolla
Car Price: 25000
Updated Car Price: 30000
```

In [94]:
```python
class company():
    def __init__(self):
        self._companyName = "Google" #private

    def getName(self):
        print(self._companyName)

c1 = company()

c1.getName()
```

```
Google
```

Exception Handling in Python

1. Exceptions:

- **Errors** in Python are represented as exceptions. They can occur during program execution due to various reasons like invalid user input, file not found, division by zero, etc.
- Exceptions can be handled to prevent abrupt termination of the program and allow graceful error recovery.

2. Try-Except Block:

- `try` : The code that might raise an exception is placed within the `try` block.
- `except` : If an exception occurs within the `try` block, the execution moves to the corresponding `except` block that handles the exception.
- Example: try:
```
  # Code that might raise an exception
  result = 10 / 0  # Division by zero
```
  except ZeroDivisionError as e:
```
  # Handling the ZeroDivisionError exception
  print("Error:", e)
```

3. Handling Multiple Exceptions:

- Multiple `except` blocks can be used to handle different types of exceptions.
- A single `except` block can handle multiple exceptions by placing them within a tuple.
- Example: try:
```
  # Code that might raise exceptions
  file = open("nonexistent_file.txt", "r")
  result = 10 / 0  # Division by zero
```
  except (FileNotFoundError, ZeroDivisionError) as e:
```
  # Handling multiple exceptions
  print("Error:", e)
```

4. `else` and `finally` Blocks:

- `else` : Used after the `try` block. Executes if no exception occurs.
- `finally` : Used to execute code irrespective of whether an exception occurred or not.
- Example: try:
  ```
  # Code that might raise an exception
  result = 10 / 2  # Division
  ```
  except ZeroDivisionError as e:
  ```
  # Handling the ZeroDivisionError exception
  print("Error:", e)
  ```
  else:
  ```
  # Executed if no exception occurs
  print("Result:", result)
  ```
  finally:
  ```
  # Executed irrespective of an exception
  print("Execution completed.")
  ```

5. Raising Exceptions:

- `raise` statement is used to raise a specific exception manually.
- Example: x = 10 if x > 5:
  ```
  raise ValueError("x should not be greater than 5")
  ```

In [95]:
```python
try:
    # Code that might raise an exception
    result = 10 / 0  # Division by zero
except ZeroDivisionError as e:
    # Handling the ZeroDivisionError exception
    print("Error:", e)
else:
    # Executed if no exception occurs
    print("Result:", result)
finally:
    # Executed irrespective of an exception
    print("Execution completed.")

try:
    # Code that might raise exceptions
    file = open("nonexistent_file.txt", "r")
    result = 10 / 0  # Division by zero
except (FileNotFoundError, ZeroDivisionError) as e:
    # Handling multiple exceptions
    print("Error:", e)

try:
    x = 10
    if x > 5:
        raise ValueError("x should not be greater than 5")
except ValueError as e:
    # Raising an exception manually
    print("Error:", e)
```

```
Error: division by zero
Execution completed.
Error: [Errno 2] No such file or directory: 'nonexistent_file.txt'
Error: x should not be greater than 5
```

In [103…

```python
try:
    print("Hi")
    print("Bye")
    printt("Hey")

except Exception as e:
    print(e)

finally:
    print("Have you understood what what happens haha...")
```

```
Hi
Bye
name 'printt' is not defined
Have you understood what what happens haha...
```

File Handling

1. File Operations:

- **File**: A named location on disk to store related information.
- Python offers built-in functions and methods for creating, reading, updating, and deleting files.

2. File Modes:

- **Read Mode (`'r'`)**: Opens a file for reading. Raises an error if the file does not exist.
- **Write Mode (`'w'`)**: Opens a file for writing. Creates a new file if it does not exist. Truncates the file if it exists.
- **Append Mode (`'a'`)**: Opens a file for appending. Creates a new file if it does not exist. Preserves existing file content.
- **Read and Write Mode (`'r+'`)**: Opens a file for both reading and writing.
- **Binary Mode (`'b'`)**: Adds a binary mode to the existing modes, allowing manipulation of binary files.

3. File Handling Steps:

- **Opening a File**: Using the `open()` function to open a file in a specified mode.
- **Reading from a File**: Using methods like `read()`, `readline()`, or `readlines()` to read content from the file.
- **Writing to a File**: Using methods like `write()` or `writelines()` to write content to the file.
- **Closing a File**: Using the `close()` method to close the file once operations are done. It's crucial for proper memory management and file closure.

4. File Object Attributes and Methods:

- **`read(size)`** : Reads `size` bytes from the file.
- **`readline(size)`** : Reads a line from the file.
- **`write(string)`** : Writes the string to the file.
- **`close()`** : Closes the file.

- `seek(offset, whence)` : Moves the file pointer to a specific position.
- `tell()` : Returns the current position of the file pointer.

5. **Context Managers (`with` statement):**

- The `with` statement in Python ensures that the file is properly closed after its suite finishes, even if an exception is raised.
- It simplifies the process of opening and working with files by automatically handling resource management.

6. **Examples:**

Reading from a file

with open("file.txt", "r") as file: content = file.read() print(content)

Writing to a file

with open("new_file.txt", "w") as file: file.write("Hello, World!")

Appending to a file

with open("existing_file.txt", "a") as file: file.write("\nAppending new content.")

In [ ]:
```
import os

f = open("fruits.txt", "w") #write only

f.write("Hi Vithu\n")
f.write("Hi Nila\n")
f.write("Happy Coding\n")
f.close()

f = open("fruits.txt", "r+") #read and write
f.readline()

#f.read()

#os.remove("fruits.txt")
```

Thank you

Happy Coding

by - Vithusan https://www.linkedin.com/in/vimalathasvithusan

In [ ]:
```

```