DAY-3

1. map():

Applies a given function to all items in an input list (or any iterable) and returns a new iterable (map object) with the results.

Syntax: map(function, iterable)

Ex:

```
def square(x):
    return x * 2
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square,
numbers)
print(list(squared_numbers))
```

Output: [2, 4, 6, 8, 10]

2. filter():

Filters elements from an iterable for which a function returns True.

Syntax: filter(function, iterable)

Ex:

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5]
even_numbers = filter(is_even,
numbers)

print(list(even_numbers))
```

Output: [2, 4]

3.reduce():

Applies a function of two arguments cumulatively to the items of an iterable, from left to right, to reduce the iterable to a single value. It is available in the functools module.

Syntax:

reduce(function,iterable[initialize])

Ex:

```
from functools import reduce
def add(x, y):
    return x + y
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(add, numbers)
print(sum_of_numbers)
```

Output: 15

4. Exception handling :

Attempts to execute 10/0

Zerodivisionerror is raised because division by zero is not allowed

- Try except block is used to handle error with code.
- Syntaxerror,cannot be handling.
- The try block letsyou test a block of code for error.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try-and except blocks

Ex:

```
try:
    number = int(input("Enter a
number: "))
    result = 10 / number
    print(f"The result is {result}")
except ValueError:
       print("That's not a valid
number!")
except ZeroDivisionError:
       print("You can't divide by
zero!")
```

output:Enter a number: 5
The result is 2.0

5.classes and objects:

Class: A blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
Object: An instance of a class. It contains data and behavior as defined by its class.

Ex:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
  def bark(self):
    print(f"{self.name} says woof!")
my_dog = Dog("Buddy", 3)
my_dog.bark()
```

Output: Buddy says woof!

6.Inheritance:

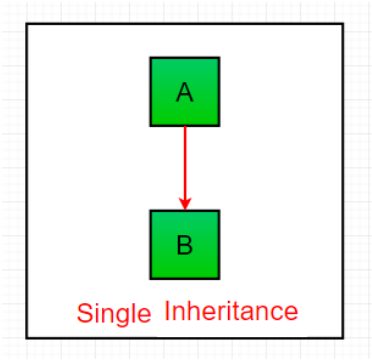
A feature in object-oriented programming where a new class (subclass) inherits attributes and methods from an existing class (superclass). It allows for code reuse and the creation of a hierarchical relationship between classes.

Ex:

```
class Parent:
    def greet(self):
        print("Hello from Parent!")
class Child(Parent): pass
obj = Child()
obj.greet()
```

output:Hello from Parent!

- **Single Inheritance**: A type of inheritance where a class (subclass) inherits from only one parent class (superclass). This allows the subclass to use the attributes and methods of the single parent class.

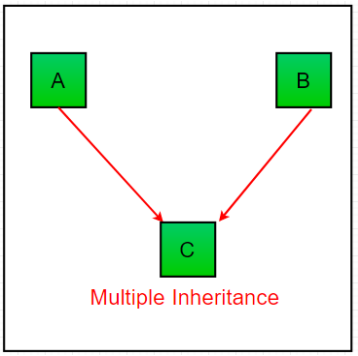


Ex:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")
class Dog(Animal):
    def bark(self):
        print("Dog says woof!")
my_dog = Dog()
my_dog.speak()
my_dog.bark()
```

Output:Animal makes a sound.Dog says woof!

Multiple Inheritance: A type of inheritance where a class (subclass) inherits from more than one parent class (superclasses). This allows the subclass to use attributes and methods from all its parent classes.



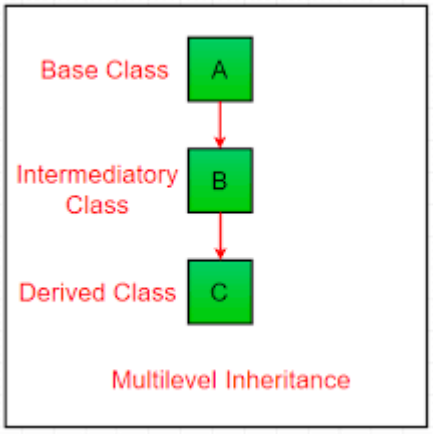
Ex:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")
class Pet:
    def play(self):
        print("Pet plays with a toy.")
class Dog(Animal, Pet):
    def bark(self):
        print("Dog says woof!")
my_dog = Dog()
```

```
my_dog.speak()
my_dog.play()
my_dog.bark()
```

Output: Animal makes a sound.
Pet plays with a toy.
Dog says woof!

- **Multilevel Inheritance:** A type of inheritance where a class (grandchild class) inherits from another class (child class), which in turn inherits from a third class (grandparent class). This creates a chain of inheritance.



Ex:

```
# Define the grandparent class
class Animal:
    def speak(self):
        print("Animal makes a sound.")
# Define the parent class that inherits
from Animal
class Mammal(Animal):
    def breathe(self):
        print("Mammal breathes air.")
# Define the child class that inherits
from Mammal
class Dog(Mammal):
    def bark(self):
        print("Dog says woof!")

# Create an object of the Dog class
my_dog = Dog()

# Call methods from all classes
my_dog.speak()   # From Animal
```

```
my_dog.breathe() # From Mammal
my_dog.bark()    # Subclass method

Output: Animal makes a sound.
Mammal breathes air.
Dog says woof!
```

7.Encapsulation:

A core concept in object-oriented programming where the internal state of an object is hidden from the outside world. It is achieved by making attributes private and providing public methods to access or modify these attributes. This helps in protecting the data and ensuring that it is accessed or modified only in controlled ways.

Ex:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # Private
attribute
        self.__age = age    # Private
attribute

    # Public method to get the name
    def get_name(self):
        return self.__name

    # Public method to set the name
    def set_name(self, name):
        self.__name = name

    # Public method to get the age
    def get_age(self):
        return self.__age

    # Public method to set the age
    def set_age(self, age):
```

```
        if age > 0:
            self.__age = age

# Create an object of the Person
class
person = Person("Alice", 30)

# Access and modify the private
attributes using public methods
print(person.get_name())
person.set_name("Bob")
print(person.get_name())
print(person.get_age())
person.set_age(35)
print(person.get_age())
# Direct access to private attributes
is not allowed
# print(person.__name)  # This would
raise an AttributeError
```

Output:
Alice
Bob
30
35

8.Polymorphism: A concept in object-oriented programming where a single function or operator can work in different ways depending on the context or the objects it is operating on. It allows methods to do different things based on the object it is acting upon, typically through method overriding or method overloading.

Ex:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("Dog says woof!")

class Cat(Animal):
    def speak(self):
        print("Cat says meow!")

# Create instances of Dog and Cat
dog = Dog()
cat = Cat()

# Call the speak method on both objects
dog.speak()  # Output: Dog says woof!
cat.speak()  # Output: Cat says meow!
```

Output: Dog says woof!
Cat says meow!

9.Constructors: Special methods in a class that are automatically called when an instance of the class is created. In Python, the constructor method is called `__init__`. It initializes the object's attributes and sets up any necessary state.

Ex:

```
class Person:
    def __init__(self, name, age):
        self.name = name  # Initialize the name attribute
        self.age = age    # Initialize the age attribute

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

# Create an instance of Person
person = Person("Alice", 30)

# Call a method to display information
person.display_info()  # Output: Name: Alice, Age: 30
```

Output: Name: Alice, Age: 30

10.Destructors: Special methods in a class that are automatically called when an instance of the class is destroyed. In Python, the destructor method is called `__del__`. It is used for cleanup operations before an object is destroyed, such as releasing resources or closing files.

Ex:

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

print(f"Resource'{self.name}'created.")
def __del__(self):
print(f"Resource’{self.name}'is being
destroyed.")
# Create an instance of Resource
resource = Resource("SampleResource")

# Delete the instance explicitly
del resource
```

output:Resource 'SampleResource' created.
Resource 'SampleResource' is being destroyed.