Implement the following hierarchy . The Book function has name, n (number of authors), authors (list of authors), publisher, ISBN, and year as its data members and the derived class has course as its data member. The derived class method overrides (extends) the methods of the base class.

In [1]:

```
class Book:
    def __init__(self, name, n, authors, publisher, ISBN, year):
        self.name = name
        self.n = n
        self.authors = authors
        self.publisher = publisher
        self.ISBN = ISBN
        self.year = year

    def display(self):
        print(f"Name: {self.name}")
        print(f"Number of authors: {self.n}")
        print(f"Authors: {', '.join(self.authors)}")
        print(f"Publisher: {self.publisher}")
        print(f"ISBN: {self.ISBN}")
        print(f"Year: {self.year}")


class CourseBook(Book):
    def __init__(self, name, n, authors, publisher, ISBN, year, course):
        super().__init__(name, n, authors, publisher, ISBN, year)
        self.course = course

    def display(self):
        super().display()
        print(f"Course: {self.course}")


book = Book("The Great Gatsby", 1, ["F. Scott Fitzgerald"], "Scribner", "9780743273565", 1925)
book.display()

print()

course_book = CourseBook("Data Science from Scratch", 1, ["Joel Grus"], "O'Reilly", "9781492041139", 2019, "Data Scien
course_book.display()
```

```
Name: The Great Gatsby
Number of authors: 1
Authors: F. Scott Fitzgerald
Publisher: Scribner
ISBN: 9780743273565
Year: 1925

Name: Data Science from Scratch
Number of authors: 1
Authors: Joel Grus
Publisher: O'Reilly
ISBN: 9781492041139
Year: 2019
Course: Data Science
```

Implement the following hierarchy . The Staff function has name and salary as its data members, the derived class Teaching has subject as its data member and the class NonTeaching has department as its data member. The derived class method overrides (extends) the methods of the base class.

In [2]:

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

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

class Teaching(Staff):
    def __init__(self, name, salary, subject):
        super().__init__(name, salary)
        self.subject = subject

    def display(self):
        super().display()
        print(f"Subject: {self.subject}")

class NonTeaching(Staff):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def display(self):
        super().display()
        print(f"Department: {self.department}")
staff_member = Staff("John", 50000)
teaching_member = Teaching("Jane", 60000, "Math")
non_teaching_member = NonTeaching("Joe", 40000, "Finance")

staff_member.display()
print()
teaching_member.display()
print()
non_teaching_member.display()
```

```
Name: John
Salary: 50000

Name: Jane
Salary: 60000
Subject: Math

Name: Joe
Salary: 40000
Department: Finance
```

Create a class called Student, having name and email as its data members andm *init*(self, name, email) and putdata(self) as bound methods. The *init* function should assign the values passed as parameters to the requisite variables. The putdata function should display the data of the student. Create another class called PhDguide having name, email, and students as its data members. Here, the students variable is the list of students under the guide. The PhDguide class should have four bound methods: *init*, putdata, add, and remove. The *init* method should initialize the variables, the putdata should show the data of the guide, include the list of students,

the add method should add a student to the list of students of the guide and the remove function should remove the student (if the student exists in the list of students of that guide) from the list of students.

In [5]:

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

    def putdata(self):
        print("Name:", self.name)
        print("Email:", self.email)

class Student(Person):
    def __init__(self, name, email):
        super().__init__(name, email)

class PhDguide(Person):
    def __init__(self, name, email):
        super().__init__(name, email)
        self.students = []

    def putdata(self):
        super().putdata()
        print("Students:", self.students)

    def add(self, student):
        self.students.append(student)

    def remove(self, student):
        if student in self.students:
            self.students.remove(student)
            print(f"{student.name} has been removed from the list of students.")
        else:
            print(f"{student.name} is not in the list of students.")
# Creating a Student object
student1 = Student("John Doe", "johndoe@example.com")
student1.putdata()  # Output: Name: John Doe, Email: johndoe@example.com

# Creating a PhDguide object with an empty list of students
guide1 = PhDguide("Jane Smith", "janesmith@example.com")
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com, Students: []

# Adding the student1 to guide1's list of students
guide1.add(student1)
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com, Students: [Student: John Doe]

# Removing the student1 from guide1's list of students
guide1.remove(student1)
guide1.putdata()  # Output: Name: Jane Smith, Email: janesmith@example.com, Students: []
```

```
Name: John Doe
Email: johndoe@example.com
Name: Jane Smith
Email: janesmith@example.com
Students: []
Name: Jane Smith
Email: janesmith@example.com
Students: [<__main__.Student object at 0x0000026FCFB731F0>]
John Doe has been removed from the list of students.
Name: Jane Smith
Email: janesmith@example.com
Students: []
```

Write program that has a class point. Define another class location which has two objects (Location and Destination) of class point. Also define function in Location that prints reflection of Destination on the x axis.

In [8]:

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

class Location(Point):
    def __init__(self, x1, y1, x2, y2):
        super().__init__(x1, y1)
        self.destination = Destination(x2, y2)

    def reflect_destination_on_x_axis(self):
        self.destination.y = -self.destination.y
        print("Reflected Destination point on X axis: ({}, {})".format(self.destination.x, self.destination.y))
class Destination(Point):
    pass

# Example usage
l = Location(1, 2, 3, 4)
l.reflect_destination_on_x_axis() # prints (3, -4)
```

```
Reflected Destination point on X axis: (3, -4)
```

Write program that has classes such as Student, Course and Department. Enroll astudent in a course of particular department

In [9]:

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

class Course(Department):
    def __init__(self, name, department):
        super().__init__(department)
        self.course_name = name

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

class Enroll(Student, Course):
    def __init__(self, name, roll_no, course_name, department):
        Student.__init__(self, name, roll_no)
        Course.__init__(self, course_name, department)

    def get_enrolled(self):
        print(f"{self.name} with roll no. {self.roll_no} has enrolled for {self.course_name} in {self.name} department
enrollment = Enroll("Alice", 101, "Calculus", "Math")
enrollment.get_enrolled()
```

```
Math with roll no. 101 has enrolled for Calculus in Math department.
```

Create a class student with following member attributes: roll no, name, age and total marks. Create suitable methods for reading and printing member variables. Write a python program to overload '==' operator to print the details of students having same marks.

In [10]:

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

    def display_student_info(self):
        print(f"Roll No: {self.roll_no}")
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Total Marks: {self.total_marks}")

    def __eq__(self, other):
        if isinstance(other, Student):
            return self.total_marks == other.total_marks
        return False
# create two student objects
s1 = Student(1, "John", 20, 90)
s2 = Student(2, "Mary", 21, 80)

# compare the students based on their total marks
if s1 == s2:
    print(f"{s1.name} and {s2.name} have the same marks!")
else:
    print(f"{s1.name} and {s2.name} do not have the same marks.")
```

```
John and Mary do not have the same marks.
```

Write a program to create a class called Data having “value” as its data member. Overload the (>) and the (<) operator for the class. Instantiate the class and compare the objects using *lt* and *gt*.

In [11]:

```
class Data:
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

    def __gt__(self, other):
        return self.value > other.value

# Testing the Data class
d1 = Data(10)
d2 = Data(20)
d3 = Data(15)

print(d1 > d2) # False
print(d2 > d3) # True
print(d3 < d1) # False
print(d3 < d2) # True
```

```
False
True
False
True
```

The following illustration creates a class called data. If no argument is passed while instantiating the class a false is returned, otherwise a true is returned.

In [15]:

```
class Data:
    def __init__(self, value=None):
        if value:
            self.value = value
            self.status = True
        else:
            self.status = False

    def __str__(self):
        return f"status: {self.status}"
data_obj1 = Data()
print(data_obj1)
# Output: status: False

data_obj2 = Data(10)
print(data_obj2)
# Output: status: True
```

```
status: False
status: True
```

In [2]:

```
class Fraction:

  # parameterized constructor
  def __init__(self,x,y):
    self.num = x
    self.den = y

  def __str__(self):
    return '{}/{}'.format(self.num,self.den)

  def __add__(self,other):
    new_num = self.num*other.den + other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __sub__(self,other):
    new_num = self.num*other.den - other.num*self.den
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __mul__(self,other):
    new_num = self.num*other.num
    new_den = self.den*other.den

    return '{}/{}'.format(new_num,new_den)

  def __truediv__(self,other):
    new_num = self.num*other.den
    new_den = self.den*other.num

    return '{}/{}'.format(new_num,new_den)

  def convert_to_decimal(self):
    return self.num/self.den
```

In [3]:

```
fr1 = Fraction(3,4)
fr2 = Fraction(1,2)
print(fr1 + fr2)
print(fr1 - fr2)
print(fr1 * fr2)
print(fr1 / fr2)
```

```
10/8
2/8
3/8
6/4
```

Write a program with class Bill. The users have the option to pay the bill either by cheque or by cash. Use the inheritance to model this situation.

In [4]:

```
# Write code here
class Bill:

  def __init__(self,items,price):
    self.total = 0
    self.items = items
    self.price = price

    for i in self.price:
      self.total = self.total + i

  def display(self):
    print('Item \t\t\t Price')
    for i in range(len(self.items)):
      print(self.items[i],'\t',self.price[i])
    print("*"*10)

    print("Total",self.total)

class CashPayment(Bill):

  def __init__(self,items,price,deno,value):
    super().__init__(items,price)

    self.deno = deno
    self.value = value

  def show_cash_payment(self):
    super().display()
    for i in range(len(self.deno)):
      print(self.deno[i],"*",self.value[i],"=",self.deno[i]*self.value[i])

class ChequePayment(Bill):

  def __init__(self,items,price,cno,name):
    super().__init__(items,price)

    self.cno = cno
    self.name = name

  def show_cheque_payment(self):
    super().display()
    print('Cheque no',self.cno)
    print('Bank name',self.name)
```

In [5]:

```
items = ["External Hard Disk", "RAM", "Printer", "Pen Drive"]
price = [5000, 2000, 6000, 800]

deno = [10, 20, 50, 100, 500, 2000]
value = [1, 1, 1, 20, 4, 5]
cash = CashPayment(items, price, deno, value)
cash.show_cash_payment()
```

```
Item                    Price
External Hard Disk      5000
RAM     2000
Printer         6000
Pen Drive       800
**********
Total 13800
10 * 1 = 10
20 * 1 = 20
50 * 1 = 50
100 * 20 = 2000
500 * 4 = 2000
2000 * 5 = 10000
```

In [6]:

```
items = ["External Hard Disk", "RAM", "Printer", "Pen Drive"]
price = [5000, 2000, 6000, 800]
option = int(input("Would you like to pay by cheque or cash (1/2): "))

if option == 1:
    name = input("Enter the name of the bank: ")
    cno = input("Enter the cheque number: ")
    cheque = ChequePayment(items, price, cno, name)
    cheque.show_cheque_payment()

else:
    deno = [10, 20, 50, 100, 500, 2000]
    value = [1, 1, 1, 20, 4, 5]
    cash = CashPayment(items, price, deno, value)
    cash.show_cash_payment()
```

```
Would you like to pay by cheque or cash (1/2): 2
Item                    Price
External Hard Disk      5000
RAM     2000
Printer         6000
Pen Drive       800
**********
Total 13800
10 * 1 = 10
20 * 1 = 20
50 * 1 = 50
100 * 20 = 2000
500 * 4 = 2000
2000 * 5 = 10000
```

Create a class called 'Matrix' containing constructor that initializes the number of rows and number of columns of a new Matrix object. The Matrix class has methods for each of the following: 1 - get the number of rows 2 - get the number of columns 3 - set the elements of the matrix at given position (i,j) 4 - adding two matrices. If the matrices are not addable, "Matrices cannot be added" will be displayed. (Overload the addition operator to perform this) 5 - multiplying the two matrices. If the matrices cannot be multiplied, "Matrices cannot be multiplied" will be displayed. (Overload the multiplication operator to perform this) 1 mark for creating appropriate objects of this class and demonstrating all the methods with correct output

In [2]:
```
class Matrix:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.matrix = []
        for i in range(self.rows):
            self.matrix.append([0] * self.columns)

    def get_rows(self):
        return self.rows

    def get_columns(self):
        return self.columns

    def set_element(self, i, j, value):
        self.matrix[i][j] = value

    def __add__(self, other_matrix):
        if self.rows != other_matrix.rows or self.columns != other_matrix.columns:
            return "Matrices cannot be added"
        result_matrix = Matrix(self.rows, self.columns)
        for i in range(self.rows):
            for j in range(self.columns):
                result_matrix.matrix[i][j] = self.matrix[i][j] + other_matrix.matrix[i][j]
        return result_matrix

    def __mul__(self, other_matrix):
        if self.columns != other_matrix.rows:
            return "Matrices cannot be multiplied"
        result_matrix = Matrix(self.rows, other_matrix.columns)
        for i in range(result_matrix.rows):
            for j in range(result_matrix.columns):
                result = 0
                for k in range(self.columns):
                    result += self.matrix[i][k] * other_matrix.matrix[k][j]
                result_matrix.matrix[i][j] = result
        return result_matrix

    def __str__(self):
        return str(self.matrix)
m1 = Matrix(2, 3)
m1.set_element(0, 0, 1)
m1.set_element(0, 1, 2)
m1.set_element(0, 2, 3)
m1.set_element(1, 0, 4)
m1.set_element(1, 1, 5)
m1.set_element(1, 2, 6)
print(m1)

m2 = Matrix(2, 3)
m2.set_element(0, 0, 7)
m2.set_element(0, 1, 8)
m2.set_element(0, 2, 9)
m2.set_element(1, 0, 10)
m2.set_element(1, 1, 11)
m2.set_element(1, 2, 12)
print(m2)

m3 = m1 + m2
print(m3)

m4 = m1 * m2
print(m4)
```

```
[[1, 2, 3], [4, 5, 6]]
[[7, 8, 9], [10, 11, 12]]
[[8, 10, 12], [14, 16, 18]]
Matrices cannot be multiplied
```

In [3]:
```python
import numpy as np

class Matrix:
    def __init__(self, rows, columns):
        self.matrix = np.zeros((rows, columns))
        self.rows = rows
        self.columns = columns

    def get_rows(self):
        return self.rows

    def get_columns(self):
        return self.columns

    def set_element(self, i, j, value):
        self.matrix[i][j] = value

    def __add__(self, other):
        if self.rows != other.rows or self.columns != other.columns:
            print("Matrices cannot be added")
            return
        result = Matrix(self.rows, self.columns)
        result.matrix = np.add(self.matrix, other.matrix)
        return result

    def __mul__(self, other):
        if self.columns != other.rows:
            print("Matrices cannot be multiplied")
            return
        result = Matrix(self.rows, other.columns)
        result.matrix = np.dot(self.matrix, other.matrix)
        return result

    def __str__(self):
        return str(self.matrix)

# creating objects and demonstrating the methods
matrix1 = Matrix(2, 2)
matrix1.set_element(0, 0, 1)
matrix1.set_element(0, 1, 2)
matrix1.set_element(1, 0, 3)
matrix1.set_element(1, 1, 4)

matrix2 = Matrix(2, 2)
matrix2.set_element(0, 0, 5)
matrix2.set_element(0, 1, 6)
matrix2.set_element(1, 0, 7)
matrix2.set_element(1, 1, 8)

print("Matrix 1:")
print(matrix1)
print("Matrix 2:")
print(matrix2)

print("Number of rows in matrix 1:", matrix1.get_rows())
print("Number of columns in matrix 1:", matrix1.get_columns())

result = matrix1 + matrix2
print("Result of addition:")
print(result)

result = matrix1 * matrix2
print("Result of multiplication:")
print(result)
```

```
Matrix 1:
[[1. 2.]
 [3. 4.]]
Matrix 2:
[[5. 6.]
 [7. 8.]]
Number of rows in matrix 1: 2
Number of columns in matrix 1: 2
Result of addition:
[[ 6.  8.]
 [10. 12.]]
Result of multiplication:
[[19. 22.]
 [43. 50.]]
```

In [5]:
```
class Student:
    def __init__(self, roll_no, name, age, total_marks):
        self.roll_no = roll_no
        self.name = name
        self.age = age
        self.total_marks = total_marks

    def get_roll_no(self):
        return self.roll_no

    def get_name(self):
        return self.name

    def get_age(self):
        return self.age

    def get_total_marks(self):
        return self.total_marks

    def __eq__(self, other):
        return self.total_marks == other.total_marks

# create some students
s1 = Student(1, 'John', 20, 90)
s2 = Student(2, 'Jane', 21, 95)
s3 = Student(3, 'Jim', 22, 90)
s4 = Student(4, 'Jill', 23, 80)

# put the students in a list
students = [s1, s2, s3, s4]

# loop through the list and compare the marks
for student in students:
    for other_student in students:
        if student == other_student:
            print('Student with roll no:', student.get_roll_no(), 'and name:', student.get_name(), 'have same marks:', student.ge
```

```
Student with roll no: 1 and name: John have same marks: 90
Student with roll no: 1 and name: John have same marks: 90
Student with roll no: 2 and name: Jane have same marks: 95
Student with roll no: 3 and name: Jim have same marks: 90
Student with roll no: 3 and name: Jim have same marks: 90
Student with roll no: 4 and name: Jill have same marks: 80
```