Python-Module-4-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

  - 3. What is meant by abstraction mechanism in programming? Give one example abstraction mechanism in Python.
  - 4. Explain the terms accessors and mutators with regard to Python class definition.
  - 5. Explain what the str method does and why it is a useful method to include in a class.
  - 6. Give an example for constructor overloading.
  - 7. Explain method overriding in Python.
  - 8. What is polymorphism? Give an example in the context of OOP in Python.
- Python-Module-4-University-Questions-Part-B
  - 1. Write a Python program to define a class Rectangle with parameters height, width and member functions to find area, and perimeter of it.
  - 2. Create a Student class and initialize it with name and roll number.
  - 3. Write Python program to create a class called as Complex to model complex numbers and implement add( ) and mul() methods to add and multiply two complex numbers. Display the result by overloading the + and operator
  - 4. Write a Python program to create a class called as Rational to model rational numbers and associated operations. Implement the following methods in the class.
  - 5. What is Exception handling? Write a program that opens a file and writes "Hello Good moring" to it. Handle exceptions that can be generated during I/O operations
  - 6. Illustrate the use of abstract classes in Python
  - 7. Define a class Student in Python with attributes to store the roll number, name and marks of three subjects for each student. Define the following methods:
  - 8. Write a Python program to demonstrate the use of try, except and finally blocks.
  - 9. How can a class be instantiated in Python? Write a Python program to express the instances as return values to define a class RECTANGLE with parameters height, width, cornerx, and cornery and member functions to find center, area, and perimeter of an instance

1. Object oriented programming and Procedural Programming

Object oriented Programming	Procedural Programming
Object oriented programming is the problem solving approach and used where computation is done by using objects	Procedural programming uses a list of instructions to do computations step by step

Object oriented Programming	Procedural Programming
It makes development and maintenance easier	Its not easy to maintain the codes when the project becomes lengthy
It simulates the real world entity, So real world problems can be easily solved through oops	It doesnt simulate the real world. It works on step by step instructions divided into small parts called functions
It provides data hiding. So it is more secure than procedural langa	Procedural language doesnt provide any proper way for data binding, so it is less secure.
Example of OOP languages is C++, Java,Python etc	Examples are C, Fortran, Pascal etc

2. Objects and Classes

What are Objects and Classes?

- Object is an instance of a class, A class is a collection of data (variables) and methods (Functions)
- Class is the basic structure of an object and is a set of attributes which can be data members or method members. Some important terms in OOP are as follows
  - **Class** - They are defined by the user. The class provides basic structure for an object.
  - **Data Member** - A variable defined in either a class or an object. It holds the data associated with the class or object.
  - **Instance variable** - A variable that is defined in a method; its scope is only within the object that defines it.
  - **Class Variable** - A variable that is defined in the class and can be used by all the instances of the class.
  - **Instance** - An object is an instance of the class.
  - **Instantiation** - The process of creation of an object of a class.
  - **Method** - Methods are the functions that are defined in the definition of class and are used by various instances of the class.
  - **Function Overloading** - A function defined more than one time with different behaviors is known as function overloading. The operations performed by these functions are different.

- **Inheritance** - a class A that can use the characteristics of another class B is said to be a derived class. ie., a class inherited from B. The process is called inheritance.

Defining a class

- Lets see an example
- We will create a class student

```
class Student:
        def fill_details(self,name,branch,year):
                self.name = name
                self.branch = branch
                self.year = year
                print("A student detail object is created")
        def print_details(self):
                print("Name:",self.name)
                print("Branch:",self.branch)
                print("Year:",self.year)
```

- Here we have defined a class Student
- It has 2 methods inside it
  - fill_details
  - print_details

Creating an Object of a class

- Lets now create Objects based on the student class we created

```
s1 = Student()
s2 = Student()
s1.fill_details('Rahul','CSE','2020')
s1.print_details()
s2.fill_details('Akhil','ECE','2010')
s2.print_details()
```

Output

```
A student detail object is created
Name: Rahul
Branch: CSE
Year: 2020
A student detail object is created
Name: Akhil
Branch: ECE
Year: 2010
```

Constructors

- A constructor is a special type of method (function) which is called when the object is created of a class
- There are 2 types of constructors
    - Parameterized Constructor
    - Non-parameterized Constructor

Creating constructor

- Class functions that begin with double underscore `__` are called special functions as they have special meaning.
- In Python, the method `__init__ ()` simulates the constructor of the class. This method is called when the class is instantiated. It accepts the self-keyword as a first argument which allows accessing the attributes of the class.

Example

- The below is a parameterized Constructor, since init has r and i has parameters

```python
class ComplexNumber:
        def __init__(self,r=0,i=0):
                self.real = r
                self.imag = i
        def getNumber(self):
                print(f'{self.real} + {self.imag}i')

num1 = ComplexNumber(2,3)
num1.getNumber()
```

Output

```
2 + 3i
```

Non-Parameterized Constructor

- The non parameterised constructor has only self as an argument

```
class Student:
        def __init__(self):
                print("This is non parameterized constructor")
        def show(self,name):
                print("Hello",name)

s1 = Student()
s1.show("Roshan")
```

Output

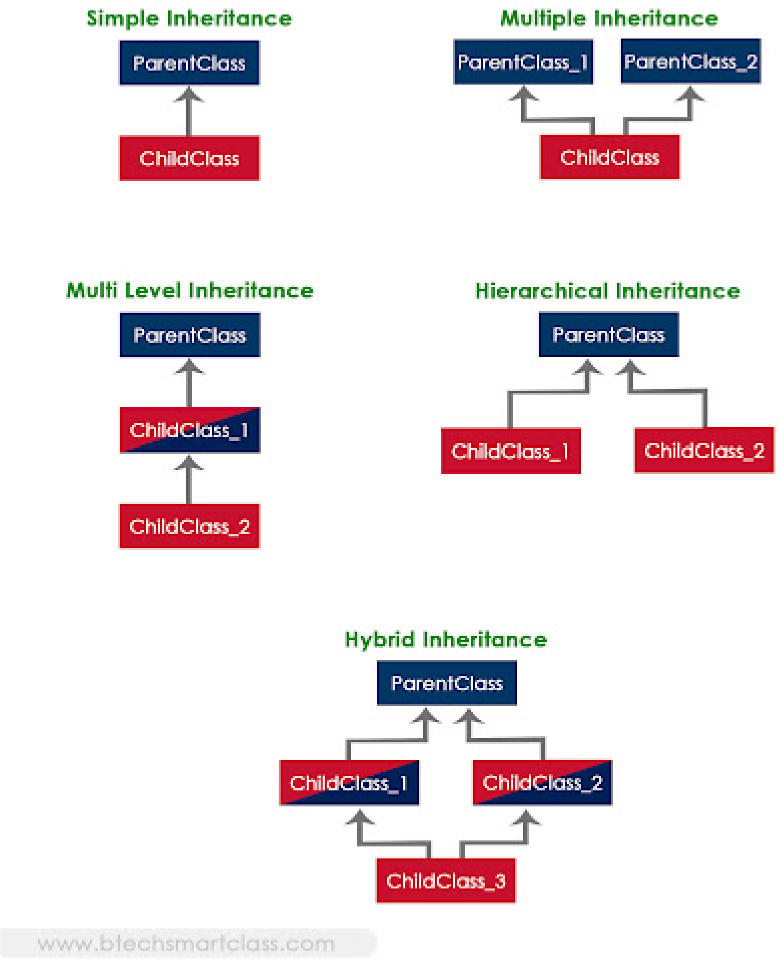
```
This is non parameterized constructor
Hello Roshan
```

3. Inheritance in Python

What is Inheritance?

- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch
- In inheritance, the child class acquires the properties and can access all the data membersa re functions defined in the parent class.

Types of inheritance



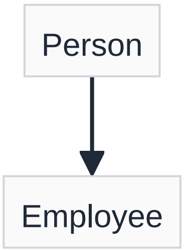
www.btechsmartclass.com

Single inheritance

When a child class inherits from one parent class. its called single inheritance

Example

Let `Person` be the parent class and `Employee` be the child class.



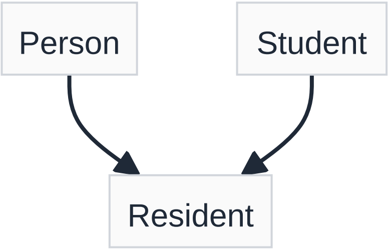
```
class Person:
        def __init__(self,name):
                self.name = name
        def getName(self):
                return self.name
        def isEmployee(self):
                return False
```

```
class Employee(Person):
        def isEmployee(self):
                return True

        p = Person("Anu")
        print(p.getName())
        p.isEmployee()

        e = Employee("Ammu")
        print(e.getName(), e.isEmployee())
```

- You can see Employee is defined like `Employee(Person)` which indicates it is deriving from class Person

Multiple inheritance

When a child class inherits from multiple parent classes, it is called multiple inheritance



Example

```
class Person:
        def __init__(self,name,age):
                self.name = name
                self.age = age
        def showName(self):
                print(self.name)
        def showAge(self):
                print(self.age)

class Student:
        def __init__(self,rollno):
                self.rollno = rollno
        def getRollNo(self):
                print(self.rollno)

class Resident(Person,Student):
        def __init__(self,name,age,rollno):
            Person.__init__(self,name,age)
            Student.__init__(self,rollno)

r = Resident("Roshan",21,101)
r.showName()
r.showAge()
r.getRollNo()
```

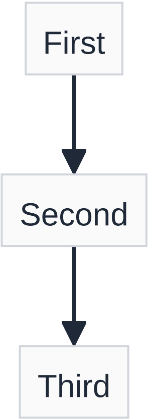
- Here Resident inherist from both Person and Student classes

Output

```
Roshan
21
101
```

Multilevel Inheritance

This is achieved when a derived class inherits another derived class.



Program

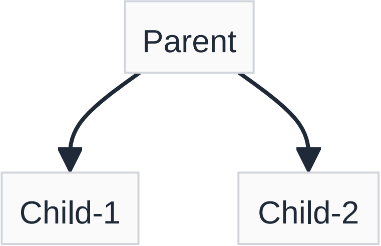
```
class First:
        def first(self):
                print("I am the first class")

class Second(First):
        def second(self):
                print("I am the second class")

class Third(Second):
        def third(self):
                print("I am the third class")

t = Third()
t.first()
t.second()
t.third()
```

Output

```
I am the first class
I am the second class
I am the third class
```

Hierarchical Inheritance

When more than one derived classes are created from a single base – it is called hierarchical inheritance.



Program

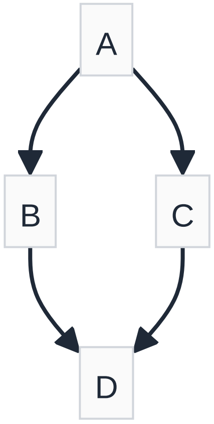
```
class Parent:
        def func1(self):
                print("This Function is in Parent")

class Child1(Parent):
        def func2(self):
                print("This function is in child 1")

class Child2(Parent):
        def func3(self):
                print("This function is in child 2")



c1 = Child1()
c2 = Child2()

c1.func1()
c1.func2()

c2.func1()
c2.func3()
```

Output

```
This Function is in Parent
This function is in child 1
This Function is in Parent
This function is in child 2
```

Hybrid Inheritance

The hybrid inheritance is the combination of more than one type of inheritance, We may use any combination as a single with multiple inheritances, multi-level with multiple inheritances etc.



4. Errors and Exceptions

Two common errors are

- Syntax errors
  - Occurs when you type the code incorrectly
- Exceptions
  - They are different from syntax error, they occur during the execution of a program when something unexpected happens

Common built-in exceptions

- **NameError:** This exception is raised when the program cannot find a local or global name. The name that could not be found is included in the error message.
- **TypeError:** This exception is raised when a function is passed an object of the inappropriate type as its argument. More details about the wrong type are provided in the error message
- **ValueError**: This exception occurs when a function argument has the right type but an inappropriate value.
- **ZeroDivisionError:** This exception is raised when you divide a number by 0
- **FileNotFoundError:** This exception is raised when the file or directory that the program requested does not exist.

Handling Exceptions

```
a = True
while a:
        try:
                x = int(input("Please enter a number: "))
                print("Dividing 50 by", x,"will give you: ", 50/x)
        except ValueError:
                print("The input was not an integer. Please try again...")
```

- Here we use try and except
- When and exception happens, it will jump to except block

Output

```
Please enter a number: a
The input was not an integer. Please try again...
Please enter a number: 2
Dividing 50 by 2
```

Raising exception

- An exception can be raised forcefully by using the raise clause

Example

```python
try:
        age = int(input("Enter the age:"))
        if(age<18):
                raise ValueError
        else:
                print("the age is valid")
except ValueError:
        print("The age is not valid")
```

Output

```
Enter the age:16
The age is not valid
```

Python-Module-4-University-Questions-Part-A

1. How to create a destructor in Python? Give an example.

- Destructors are called when an object gets destroyed.
- The **del()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.
  Program

```python
# Python program to illustrate destructor
class Employee:

        # Initializing
        def __init__(self):
                print('Employee created.')

        # Deleting (Calling destructor)
        def __del__(self):
                print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output

```
Employee created.
Destructor called, Employee deleted.
```

2. Write a Python class which has two methods get_distance and print_distance. get_distance accept a distance in kilometres from the user and print_distance,print the distance in meter.

```python
class DistanceConverter:
  def get_distance(self):
    self.distance_in_km = input("Enter distance in kilometers: ")
  def print_distance(self):
    """Prints the distance in meters."""
    distance_in_meters = self.distance_in_km * 1000
    print(distance_in_meters)

# Create an instance of the class
converter = DistanceConverter()

# Get the distance from the user
converter.get_distance()
```

```
# Print the distance in meters
converter.print_distance()
```

3. What is meant by abstraction mechanism in programming? Give one example abstraction mechanism in Python.

- Abstraction means hiding the complexity and only showing the essential features of the object.
- So in a way, Abstraction means hiding the real implementation and we, as a user, knowing only how to use it

```
class Car:
  def __init__(self, make, model, year):
    self.make = make
    self.model = model
    self.year = year

  def accelerate(self):
    print(f"The {self.make} {self.model} is accelerating!")

# Create a Car object
my_car = Car("Honda", "Civic", 2023)

# Use the object's method (abstraction)
my_car.accelerate()  # Prints "The Honda Civic is accelerating!"
```

- In this example, the `Car` class abstracts the concept of a car.
- You can create multiple `Car` objects with different attributes, and the `accelerate` method hides the underlying mechanics of how a car accelerates, focusing on the action itself. This makes the code more readable and easier to maintain.

4. Explain the terms accessors and mutators with regard to Python class definition.

- The Methods that allow a user to observe but not change the state of an object are called **accessors**.
- Methods that allow a user to modify an object’s state are called **mutators**.

Examples

```
class Fruit:
        def __init__(self, name):
                self.name = name
        def setFruitName(self, name):
                self.name = name
        def getFruitName(self):
                return self.name
f1 = Fruit("Apple")
print("First fruit name: ", f1.getFruitName())
f1.setFruitName("Grape")
print("Second fruit name: ", f1.getFruitName())
Output:
```

- Here the function `getFruitName` is an accessor
- `setFruitName` is a mutator

5. Explain what the str method does and why it is a useful method to include in a class.

- The `__str__` method in Python is a special method that defines how an object should be represented as a human-readable string.
- It's automatically called whenever you use the built-in `str()` function on an object of that class, or when you directly print the object.

Example

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

  # Define the __str__ method
  def __str__(self):
    return f"Name: {self.name}, Age: {self.age}"

# Create a Person object
person = Person("Alice", 30)

# Print the object using str() or directly
print(str(person))  # Output: Name: Alice, Age: 30
print(person)        # Output: Name: Alice, Age: 30 (same as str(person))
```

- In this example, the `__str__` method returns a string that includes the person's name and age.
- This makes the object much more informative when printed compared to the default representation.

6. Give an example for constructor overloading.

```
class Person:
        def __init__(self, name, age=None):
                self.name = name
                self.age = age


# Example Usage
person1 = Person("Alice")
person2 = Person("Bob", 25)

# Output
print(person1.name, person1.age)
print(person2.name, person2.age)
```

- In this example, The `Person` class defines a constructor with parameters `name` and optional `age`.
- Two instances, `person1` and `person2`, are created with different sets of parameters.
- The printed outputs display the names and ages for each instance, with `None` for `person1` and 25 for `person2`.

7. Explain method overriding in Python.

- In Python, method overriding refers to the ability of a subclass to redefine the behavior of a method inherited from its parent class.
- This allows you to create specialized versions of methods that are tailored to the specific needs of the subclass.

```
class Animal:
  def make_sound(self):
    print("Generic animal sound")

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

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

# Create objects
dog = Dog()
cat = Cat()

# Call the make_sound method on each object
dog.make_sound()  # Output: Woof!
cat.make_sound()  # Output: Meow!
```

- - The `Animal` class defines a `make_sound` method with a generic message.
- The `Dog` and `Cat` classes inherit from `Animal` and override the `make_sound` method to provide specific sounds for dogs and cats.

- When you call `make_sound` on a `Dog` object, the overridden version in the `Dog` class executes, printing "Woof!".
- Similarly, for a `Cat` object, the overridden version in the `Cat` class executes, printing "Meow!".

8. What is polymorphism? Give an example in the context of OOP in Python.

Polymorphism in object-oriented programming (OOP) refers to the ability of objects of different classes to respond differently to the same method call. This allows you to write generic code that can work with a variety of objects without needing to know their specific types beforehand.

Example

- **Same example in Question 8**

Python-Module-4-University-Questions-Part-B

1. Write a Python program to define a class Rectangle with parameters height, width and member functions to find area, and perimeter of it.

```
class Rectangle:
  def __init__(self, height, width):
    self.height = height
    self.width = width
  def area(self):
    return self.height * self.width
  def perimeter(self):
    return 2 * (self.height + self.width)

rectangle = Rectangle(5, 10)
print("Area of the rectangle:", rectangle.area())
print("Perimeter of the rectangle",rectangle.perimeter())
```

2. Create a Student class and initialize it with name and roll number.

Make methods to :

1. **Display - Display all informations of the student.**
2. **setAge - Assign age to student**
3. **setTestMarks - Assign marks of a test to the student.**

```python
class Student:
  def __init__(self, name, roll_number):
    self.name = name
    self.roll_number = roll_number
    self.age = None  # Initialize age as None initially
    self.test_marks = None  # Initialize test marks as None initially

  def display(self):
    print(f"Name: {self.name}")
    print(f"Roll Number: {self.roll_number}")
    if self.age is not None:
      print(f"Age: {self.age}")
    if self.test_marks is not None:
      print(f"Test Marks: {self.test_marks}")

  def set_age(self, age):
    self.age = age

  def set_test_marks(self, marks):
    self.test_marks = marks

# Example usage
student1 = Student("Alice", 123)
student1.display()  # Output: Name: Alice, Roll Number: 123

student1.set_age(18)
student1.set_test_marks(85.5)
student1.display()  # Output: Name: Alice, Roll Number: 123, Age: 18, Test Marks: 85.5
```

3. *Write Python program to create a class called as Complex to model complex numbers and implement `__add__( )` and `__mul__()` methods to add and multiply two complex numbers. Display the result by overloading the + and* operator*

```python
class Complex:
  def __init__(self, real, imag):
    self.real = real
    self.imag = imag

  def __add__(self, other):
    return Complex(self.real + other.real, self.imag + other.imag)

  def __mul__(self, other):
    return Complex(self.real * other.real - self.imag * other.imag,
                   self.real * other.imag + self.imag * other.real)

# Create complex numbers
c1 = Complex(3, 2)
c2 = Complex(1, -4)

# Add and multiply complex numbers using overloaded operators
sum_complex = c1 + c2
product_complex = c1 * c2

# Print the results (assuming desired output format)
print(f"Sum: {sum_complex.real} + {sum_complex.imag}j")
print(f"Product: {product_complex.real} + {product_complex.imag}j")
```

4. Write a Python program to create a class called as Rational to model rational numbers and associated operations. Implement the following methods in the class.

Use operator overloading.

1. **Reduce() – to return the simplified fraction form**
2. **add() - to add two ratioanal numbers**
3. **lt() - to compare two rational numbers (less than operation)****

```python
class Rational:
  def __init__(self, numerator, denominator):
    self.numerator = numerator
    self.denominator = denominator
    self.reduce()  # Reduce to simplest form upon initialization

  def gcd(self, a, b):  # Helper function for greatest common divisor
    while b != 0:
      a, b = b, a % b
    return a

  def reduce(self):
    gcd_value = self.gcd(self.numerator, self.denominator)
    self.numerator //= gcd_value
    self.denominator //= gcd_value

  def __str__(self):
    return f"{self.numerator}/{self.denominator}"

  def __add__(self, other):
    if not isinstance(other, Rational):
      raise TypeError("Can only add rational numbers.")
    new_numerator = self.numerator * other.denominator + other.numerator * self.denominator
    new_denominator = self.denominator * other.denominator
    return Rational(new_numerator, new_denominator)

  def __lt__(self, other):
    return self.numerator * other.denominator < other.numerator * self.denominator

# Example usage
r1 = Rational(3, 4)
r2 = Rational(2, 3)
```

```
# Reduce (already done in __init__)
# print(r1.reduce())  # Output: None (already simplified)

print(f"r1: {r1}")  # Output: r1: 3/4
print(f"r2: {r2}")  # Output: r2: 2/3

# Addition
sum_rational = r1 + r2
print(f"Sum: {sum_rational}")  # Output: Sum: 17/12

# Less than comparison
is_less = r1 < r2
print(f"Is r1 less than r2? {is_less}")  # Output: Is r1 less than r2? False
```

5. What is Exception handling? Write a program that opens a file and writes "Hello Good moring" to it. Handle exceptions that can be generated during I/O operations

- Exception handling is a mechanism to handle unexpected errors or exceptions that may arise during program execution.
- These exceptions can interrupt the normal flow of the program, but exception handling allows you to gracefully manage them and potentially continue execution or provide informative error messages to the user.

```
try:
  # Open the file in write mode ("w")
  with open("example.txt", "w") as file:
    file.write("Hello Good morning")
  print("File write successful.")
except IOError as e:
  # Handle IOError exception if there's an error during file operations
  print(f"An error occurred while writing to the file: {e}")
```

6. Illustrate the use of abstract classes in Python

- Abstract classes in Python provide a way to define a blueprint for other classes to inherit from.
- They enforce a certain behavior or structure on subclasses without providing a complete implementation themselves.

Example

```python
from abc import ABC, abstractmethod

class Shape(ABC):
  """
  Abstract class representing a geometric shape.
  """

  @abstractmethod
  def calculate_area(self):
    """
    Abstract method that subclasses must implement to calculate area.
    """
    pass

  def get_info(self):
    """
    Example non-abstract method that can be used by subclasses.
    """
    print(f"I am a {type(self).__name__} shape.")  # Uses __name__ to get class name dynamically

# You cannot create an instance of an abstract class directly
try:
  shape = Shape()
except TypeError as e:
  print(f"Error: {e}. You cannot create an instance of an abstract class.")
```

- For defining abstract methods in an abstract class, method has to be decorated with `@abstractmethod` decorator.
- From abc module @abstractmethod decorator has to be imported to use that annotation.

- Abstract class can have both concrete methods as well as abstract methods.
- Abstract class works as a template for other classes.

7. Define a class Student in Python with attributes to store the roll number, name and marks of three subjects for each student. Define the following methods:

readData()- to assign values to the attributes

computeTotal() – to find the total marks

print_details() - to display the attribute values and the total marks

Create an object of the class and invoke the methods.

```python
class Student:
  """
  Represents a student with roll number, name, and marks in three subjects.
  """

  def __init__(self, roll_number, name, subject1_marks, subject2_marks, subject3_marks):
    """
    Initializes the student object with attributes.

    Args:
        roll_number (int): The student's roll number.
        name (str): The student's name.
        subject1_marks (int): Marks in the first subject.
        subject2_marks (int): Marks in the second subject.
        subject3_marks (int): Marks in the third subject.
    """
    self.roll_number = roll_number
    self.name = name
    self.subject1_marks = subject1_marks
    self.subject2_marks = subject2_marks
    self.subject3_marks = subject3_marks

  def readData(self):
    """
    Prompts the user to enter data for the student's attributes.
    """
    self.roll_number = int(input("Enter roll number: "))
    self.name = input("Enter name: ")
    self.subject1_marks = int(input("Enter marks for subject 1: "))
    self.subject2_marks = int(input("Enter marks for subject 2: "))
    self.subject3_marks = int(input("Enter marks for subject 3: "))

  def computeTotal(self):
    """
    Calculates the total marks of the student.

    Returns:
        int: The total marks of the student.
    """
    return self.subject1_marks + self.subject2_marks + self.subject3_marks

  def print_details(self):
    """
    Prints the student's details and total marks.
    """
    total_marks = self.computeTotal()
    print(f"Roll Number: {self.roll_number}")
    print(f"Name: {self.name}")
    print(f"Subject 1 Marks: {self.subject1_marks}")
    print(f"Subject 2 Marks: {self.subject2_marks}")
    print(f"Subject 3 Marks: {self.subject3_marks}")
    print(f"Total Marks: {total_marks}")

# Create a student object
student1 = Student(None, None, None, None, None)

student1.readData()

# Call the methods to display details
student1.print_details()
```

8. Write a Python program to demonstrate the use of try, except and finally blocks.

```python
def divide(numerator, denominator):
  try:
    # Attempt the division operation
    result = numerator / denominator
  except ZeroDivisionError as e:
    # Handle division by zero error
    print(f"Error: Cannot divide by zero. {e}")
    return None
  finally:
    # Code in finally block always executes, regardless of exceptions
    print("Division operation attempted.")
  return result

# Example usage
num1 = 10
num2 = 0

# Try division with potential error
result = divide(num1, num2)

if result is not None:
  print(f"Division result: {result}")
```

9. How can a class be instantiated in Python? Write a Python program to express the instances as return values to define a class RECTANGLE with parameters height, width, corner_x, and corner_y and member functions to find center, area, and perimeter of an instance

```python
class Rectangle:
  def __init__(self, height, width, corner_x, corner_y):
    self.height = height
    self.width = width
    self.corner_x = corner_x
    self.corner_y = corner_y
  def get_center(self):
    center_x = self.corner_x + self.width / 2
    center_y = self.corner_y + self.height / 2
    return center_x, center_y

  def get_area(self):
    return self.height * self.width

  def get_perimeter(self):
    return 2 * (self.height + self.width)

# Create rectangle instances (example values)
rectangle1 = Rectangle(5, 3, 1, 2)
rectangle2 = Rectangle(8, 4, -2, 0)

# Get and print details for each rectangle
print("Rectangle 1:")
print(f"  Center: ({rectangle1.get_center()[0]}, {rectangle1.get_center()[1]})")
print(f"  Area: {rectangle1.get_area()}")
print(f"  Perimeter: {rectangle1.get_perimeter()}")

print("\nRectangle 2:")
print(f"  Center: ({rectangle2.get_center()[0]}, {rectangle2.get_center()[1]})")
print(f"  Area: {rectangle2.get_area()}")
print(f"  Perimeter: {rectangle2.get_perimeter()}")
```