Contents

1. Concepts of object-oriented programming

Object-oriented programming revolves around defining and using new types.

In python a class is how python represents a type. A function isinstance() reports whether an object is an instance of a class. i.e whether an object has a particular type.

- ➔ **isinstance('abc', str)**
- ➔ **True**
- ➔ **isinstance(55.2, str)**
- ➔ **False**
- ➔ **isinstance('abc', object)**
- ➔ **True**
- ➔ **isinstance(max, object)**
- ➔ **True**

Python has a class called object. Every other class is based on it. Even classes and functions are instances of object.

OOP Concepts:

Class: A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). A class is a user-defined data type that serves as a template for creating objects. It defines a set of attributes and methods that the created objects can use.

Object: An object is an instance of a class. It is a self-contained component that contains attributes and methods needed to make a particular type of data useful. When a class is defined, no memory is allocated until an object of that class is created. Objects are individual instances of a class that can have different values for the attributes defined in the class.

Attributes: Attributes are the variables that belong to an object. They are used to store the state of an object.

Methods: Methods are functions that belong to a class and define the behaviors of the objects created from the class.

Constructor: A constructor is a special method that is automatically called when an object of a class is created. It initializes the attributes of the object.

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

Encapsulation: Encapsulation is the bundling of data (attributes) and methods that operate on the data into a single unit, or class. It restricts direct access to some of an object's components, which can help prevent the accidental modification of data.

Abstraction: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object. It helps in reducing programming complexity and effort.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common super class. It is typically used to call methods that behave differently depending on the object that invokes the method.

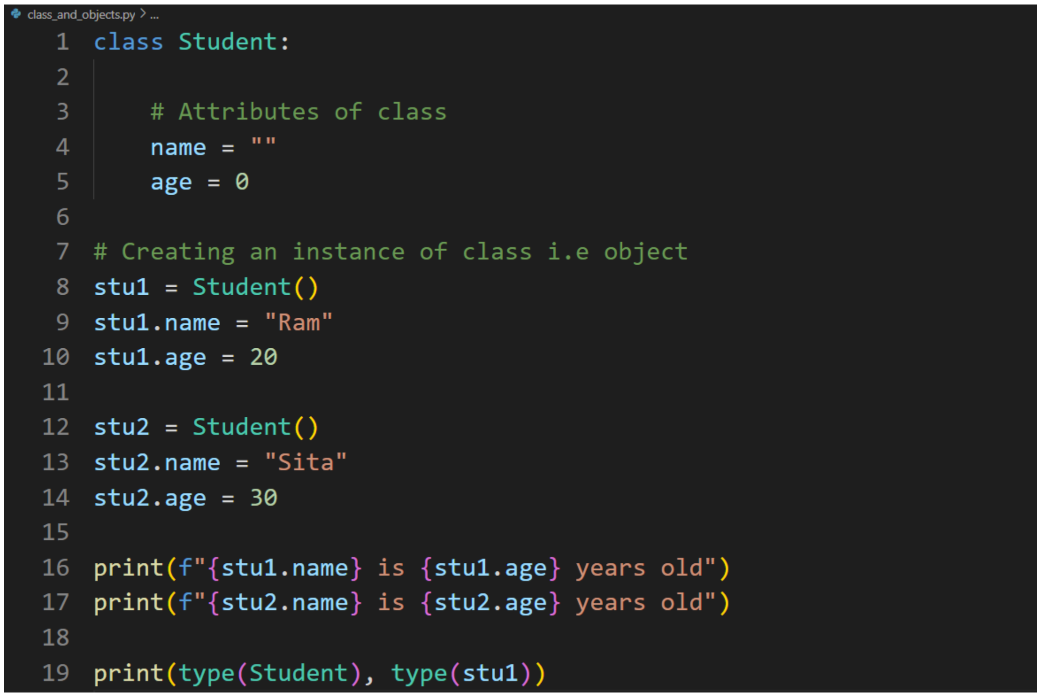
Overriding: Overriding occurs when a child class provides a specific implementation of a method that is already defined in its parent class.

Overloading: Overloading is a feature that allows a class to have more than one method having the same name, if their parameter lists are different.

2. Classes and Objects

Syntax of class:

```
class ClassName:
    # body of class
```

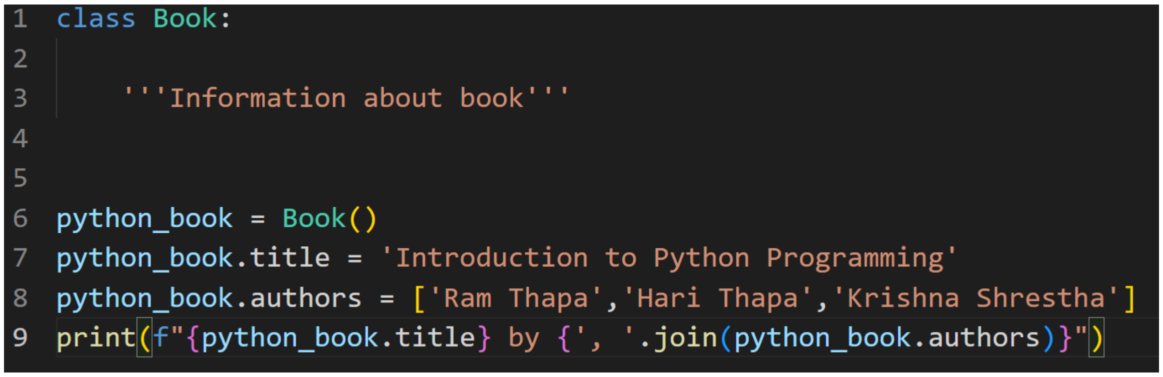

class_and_objects.py > ...

```python
class Student:

    # Attributes of class
    name = ""
    age = 0

# Creating an instance of class i.e object
stu1 = Student()
stu1.name = "Ram"
stu1.age = 20

stu2 = Student()
stu2.name = "Sita"
stu2.age = 30

print(f"{stu1.name} is {stu1.age} years old")
print(f"{stu2.name} is {stu2.age} years old")

print(type(Student), type(stu1))
```

The given code defines a Student class with class-level attributes name and age, and then creates two instances of this class (stu1 and stu2) where the attributes are individually set for each instance. It prints out the name and age for each instance, demonstrating that these attributes can hold different values for different instances. The code also prints the types of the Student class and the stu1 instance, showing <class 'type'> for the class itself and <class '__main__.Student'> for the instance, indicating that Student is a class (created using the type metaclass) and stu1 is an instance of Student.

Try the following:

- ➔ print(isinstance(stu1, Student))
- ➔ print(isinstance(stu1, object))



```python
class Book:

    '''Information about book'''


python_book = Book()
python_book.title = 'Introduction to Python Programming'
python_book.authors = ['Ram Thapa','Hari Thapa','Krishna Shrestha']
print(f"{python_book.title} by {', '.join(python_book.authors)}")
```

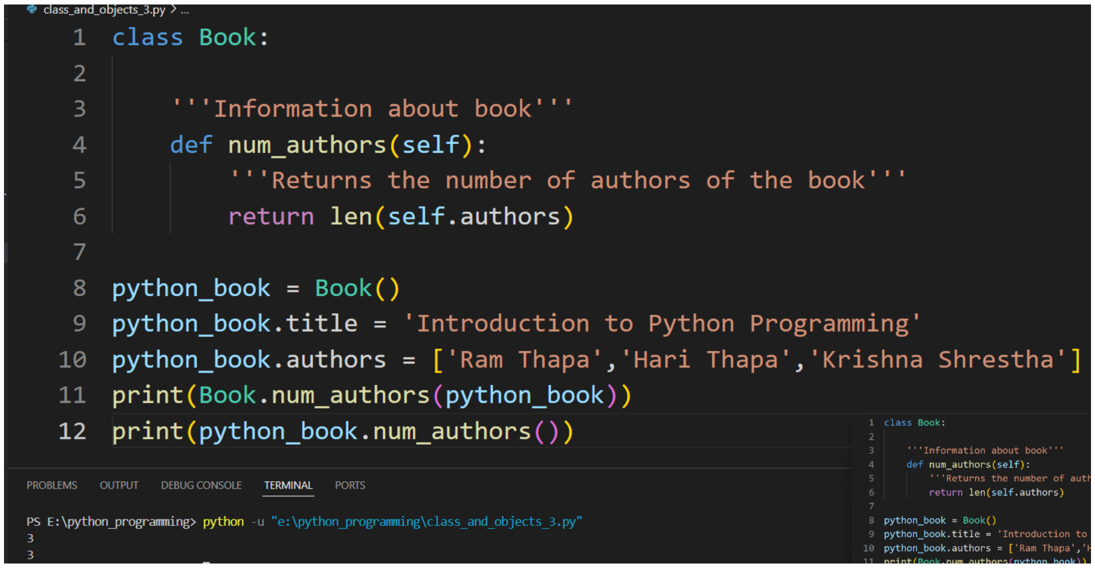
The first assignment statement creates a Book object and then assigns that object to variable python_book. The second assignment statement creates a title variable inside the Book object; that variable refers to the string 'Introduction to Python Programming'. The third assignment statement creates variable authors, also inside the Book object, which refers to the list of strings ['Ram Thapa','Hari Thapa','Krishna Shrestha'].

Variables title and authors are called instance variables because they are variables inside an instance of a class. We can access these instance variables through variable python_book.

Writing a Method in Class Book:

We can define a method num_authors that will return the numbers of authors of the book. Book method num_authors looks just like a function except that it has a parameter called self, which refers to a Book. there are two ways to call a method. One way is to access the method through the class, and the other is to use object-oriented syntax.

- ➔ Book.num_authors(python_book)
- ➔ Python_book.num_authors()



```python
class Book:

    '''Information about book'''
    def num_authors(self):
        '''Returns the number of authors of the book'''
        return len(self.authors)

python_book = Book()
python_book.title = 'Introduction to Python Programming'
python_book.authors = ['Ram Thapa','Hari Thapa','Krishna Shrestha']
print(Book.num_authors(python_book))
print(python_book.num_authors())
```

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_3.py"
3
3
```

Let's take a close look at the first call on method num_authors.

- ➔ Book.num_authors(python_book)

In class Book is method num_authors. The argument to the call, python_book, is passed to parameter self. Python treats the second call on num_authors exactly as it did the first; the first call is equivalent to this one

- ➔ Python_book.num_authors()

The second version is much more common because it lists the object first; we think of that version as asking the book how many authors it has. Thinking of method calls this way can really help develop an object-oriented mentality.

In the python_book example, we assigned the title and list of authors after the Book object was created. That approach isn't scalable; we don't want to have to type those extra assignment statements every time we create a Book. Instead, we'll write a method that does this for us as we create the Book. This is a special method and is called __init__. We'll also include the publisher, ISBN, and price as parameters of __init__:

```
class Book:

    """Information about a book, including title, list of authors publisher, ISBN, and price.
    """

    def __init__(self, title, authors, publisher, isbn, price):
        """Create a new book entitled title, written by    the people in authors,
        published by publisher, with ISBN isbn and costing price dollars.
        >>> python_book = Book( \
            'Practical Programming', \
            ['Campbell', 'Gries', 'Montojo'], \
            'Pragmatic Bookshelf', \
            '978-1-6805026-8-8', \
            25.0)
        """
        self.title = title
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price
    def num_authors(self):
        '''Returns the number of authors of the book'''
        return len(self.authors)

python_book = Book(
    'Practical Programming',
    ['Campbell', 'Gries', 'Montojo'],
    'Pragmatic Bookshelf',
    '978-1-6805026-8-8',
    25.0)

print(python_book.ISBN, python_book.title, python_book.authors)
```

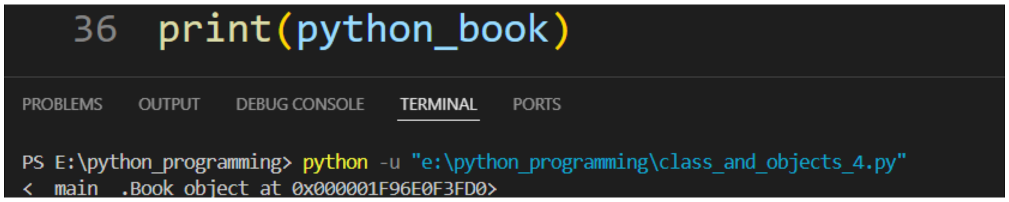
Method __init__ is called whenever a Book object is created. Its purpose is to initialize the new object; this method is sometimes called a constructor. Here are the steps that Python follows when creating an object:

1. It creates an object at a particular memory address.
2. It calls method __init__, passing in the new object into the parameter self.
3. It produces that object's memory address.

Note: Methods belong to classes. Instance variables belong to objects. If we try to access an instance variable as we do a method, we get an error.

Another special method: __str__()

When we call print(python_book), then python_book.__str__() is called to find out what string to print. The output Python produces when we print a Book isn't particularly useful:

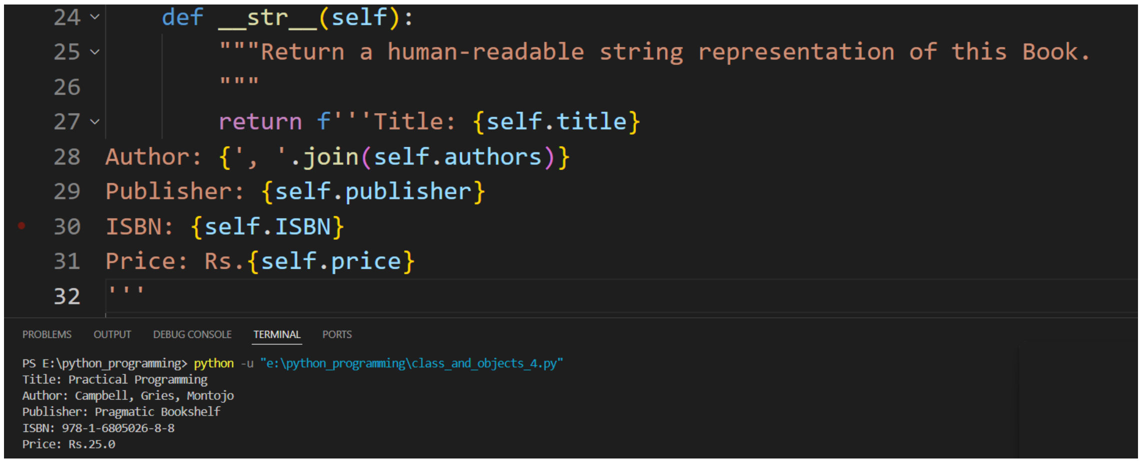


```
print(python_book)
```

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_4.py"
<__main__.Book object at 0x000001F96E0F3FD0>
```

This is the default behavior for converting objects to strings: it just shows us where the object is in memory. This is the behavior defined in class object's method __str__, which our Book class has inherited.

Let's define method Book.__str__ to provide useful output; this method goes inside class Book, along with __init__ and num_authors:



```
    def __str__(self):
        """Return a human-readable string representation of this Book.
        """
        return f'''Title: {self.title}
Author: {', '.join(self.authors)}
Publisher: {self.publisher}
ISBN: {self.ISBN}
Price: Rs.{self.price}
'''
```

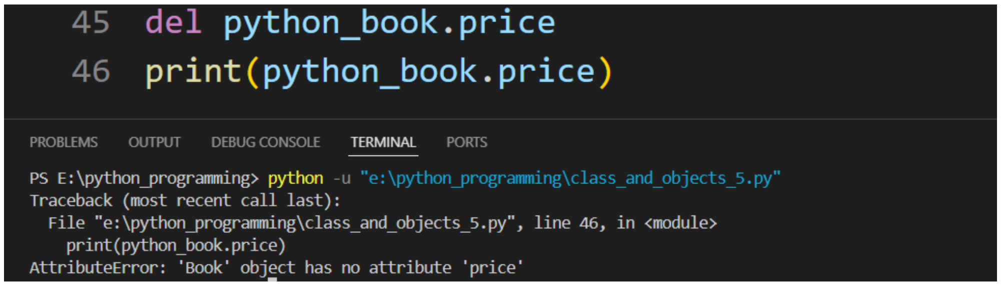
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_4.py"
Title: Practical Programming
Author: Campbell, Gries, Montojo
Publisher: Pragmatic Bookshelf
ISBN: 978-1-6805026-8-8
Price: Rs.25.0
```

The result is displayed when print(python_book)

Deleting Object Properties

To delete an attribute from an object, you can use the `del` statement followed by the object's attribute.

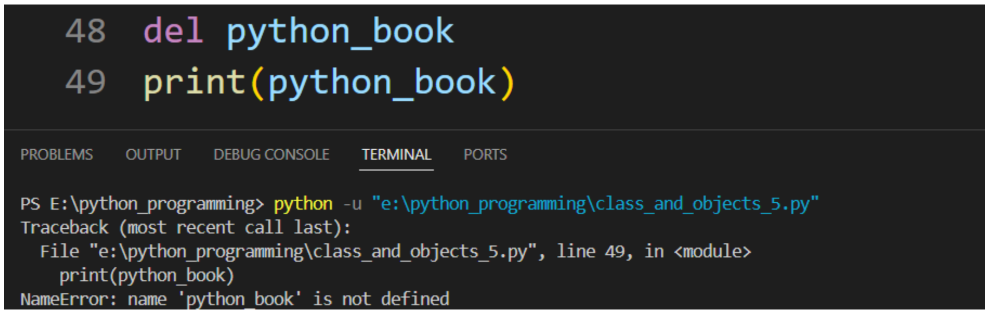


```
del python_book.price
print(python_book.price)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_5.py"
Traceback (most recent call last):
  File "e:\python_programming\class_and_objects_5.py", line 46, in <module>
    print(python_book.price)
AttributeError: 'Book' object has no attribute 'price'
```

Deleting Objects



```
del python_book
print(python_book)
```

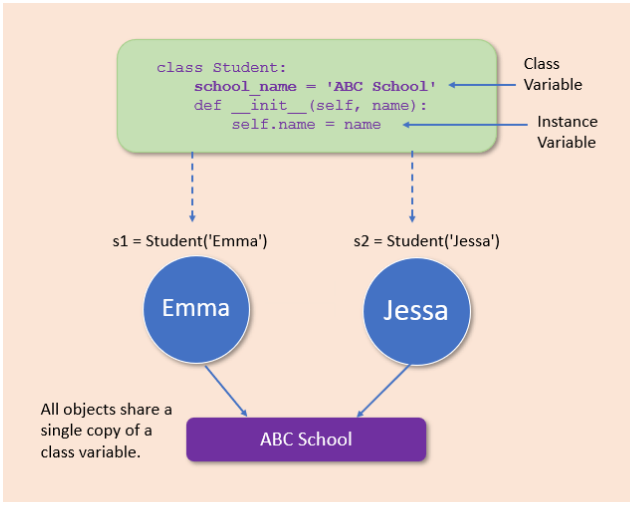
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

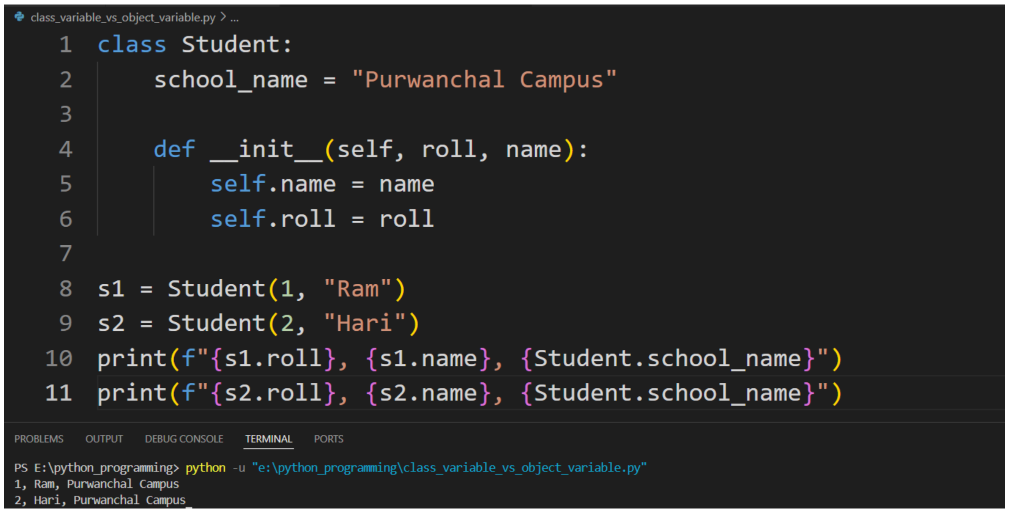
```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_5.py"
Traceback (most recent call last):
  File "e:\python_programming\class_and_objects_5.py", line 49, in <module>
    print(python_book)
NameError: name 'python_book' is not defined
```

Class Variable vs Instance Variable:

Class Variable: Class variables are variables that are shared among all instances of a class. They are defined within the class but outside any methods.

Instance Variable: Instance variables are variables that are specific to each instance of a class. They are typically defined within the __init__ method of the class.


```
class Student:
    school_name = 'ABC School'
    def __init__(self, name):
        self.name = name
```


class_variable_vs_object_variable.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```python
class Student:
    school_name = "Purwanchal Campus"

    def __init__(self, roll, name):
        self.name = name
        self.roll = roll

s1 = Student(1, "Ram")
s2 = Student(2, "Hari")
print(f"{s1.roll}, {s1.name}, {Student.school_name}")
print(f"{s2.roll}, {s2.name}, {Student.school_name}")
```

```
PS E:\python_programming> python -u "e:\python_programming\class_variable_vs_object_variable.py"
1, Ram, Purwanchal Campus
2, Hari, Purwanchal Campus
```

In the Student class, school_name is a class variable, meaning it is shared among all instances of the class and is accessed using the class name (Student.school_name). This variable is common to all students and does not change from one instance to another. On the other hand, name and roll are instance variables, defined within the __init__ method, and are unique to each instance of the class. When s1 and s2 instances are created, they have their own name and roll values (s1 has roll 1 and name "Ram", s2 has roll 2 and name "Hari"), but both share the same school_name "Purwanchal Campus".

Considering a Car example.

```
class Car:
    car_count = 0

    def __init__(self, name, mileage, color):
        self.name = name
        self.mileage = mileage
        self.color = color
        Car.car_count += 1

    def display_car(self):
        print(f"Name = {self.name}\nMileage = {self.mileage}\nColor = {self.color}")

    def update_car_name(self, name):
        self.name = name

    @classmethod
    def display_car_count(cls):
        print(f"Total Cars = {cls.car_count}")

car1 = Car('Marcedes Benz', 2222, 'blue')
car1.display_car()
car2 = Car('Ford', 1111, 'red')
car2.display_car()
car1.update_car_name('Mercedes Benz')
car1.display_car()
Car.display_car_count()
del car1.name
print(car1.__dict__)

del car1
print(car1.__dict__)
```

In the Car class, car_count is a class variable that tracks the total number of car instances, incremented each time a new Car object is created. The __init__ method initializes instance variables name, mileage, and color, and increments the car_count. The display_car method prints the car's details, while update_car_name allows updating the car's name. The display_car_count class method prints the total number of cars created. When car1 and car2 are created, their details are displayed. The name of car1 is updated from 'Marcedes Benz' to 'Mercedes Benz' and displayed again. Car.display_car_count() shows the total number of cars. Deleting car1.name removes only the name attribute from car1, while deleting car1 itself makes car1 undefined, leading to an error if accessed afterward. The __dict__ method is used to display the attributes of an object before deletion. In the Car class, **@classmethod** is a decorator used for the display_car_count method, which allows it to access the class variable car_count and operate on the class itself rather than on instances of the class.

Note: The last print(car1.__dict__) will cause an error because car1 has been deleted.

Considering Employee example:

Q. Consider a class Employee with the following instance attributes and methods:

Instance Attributes:

first: The first name of the employee

last: The last name of the employee

email: The email address of the employee, automatically generated in the format first.last@company.com

pay: The base salary of the employee

Class Attributes:

raise_amount: A class attribute that stores the raise percentage

no_of_employees: A class attribute that keeps track of the number of employees created

Instance Methods:

__init__(self, first, last, pay): Initializes the instance attributes and increments the employee count

full_name(self): Prints the full name of the employee

apply_raise(self): Applies the raise to the employee's salary based on the class attribute raise_amount

Class Methods:

set_raise_amt(cls, amount): Sets the class attribute raise_amount to a new value

Given this class definition, create two Employee instances emp_1 and emp_2 with the following details:

emp_1: first name "Hari", last name "Thapa", and pay 50000

emp_2: first name "Sita", last name "Kumari", and pay 20000

Use the class method to set the raise amount to 1.05. Apply the raise to emp_1 and print its __dict__ attribute.

```
class Employee:
    raise_amount = 1.04
    no_of_employees = 0
    def __init__(self,first,last,pay):
        self.first = first
        self.last = last
        self.email = first.lower() +"."+ last.lower() +"@company.com"
        self.pay = pay

        Employee.no_of_employees += 1

    def full_name(self):
        print("Full Name: {} {}".format(self.first,self.last))

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amount = amount

emp_1 = Employee('Hari','Thapa',50000)
emp_2 = Employee('Sita','Kumari',20000)

Employee.set_raise_amt(1.05)

emp_1.apply_raise()
print(emp_1.__dict__)
```

Getter, Setter and Deleter:

GETTER: Function defined within the class to GET the value in a class variable.

SETTER: Function defined within the class to SET the value in the class variable.

DELETER: Function defined within the class to DELETE the value in the class variable.

```
class Car:
    #Initializer/ Constructor
    def __init__(self, name, mileage, color):
        self.name = name
        self.mileage = mileage
        self.color = color

    #Setter
    def set_values(self):
        name = input("Enter the name: ")
        mileage = int(input("Enter the mileage: "))
        color = input("Enter the color: ")
        self.name = name
        self.mileage = mileage
        self.color = color

    #Getter
    def get_values(self):
        return(self.name, self.mileage, self.color)

    #Deleter
    def delete_mileage(self):
        del self.mileage

car1 = Car("BMW", 15, "White")
print(car1.__dict__)

car1.set_values()
name, mileage, color = car1.get_values()
print(name, mileage, color)

car1.delete_mileage()
print(car1.__dict__)
```

The Car class demonstrates the use of getter, setter, and deleter methods in Python. The class initializer __init__ sets initial values for the instance attributes name, mileage, and color. The set_values method is a setter that allows the user to update these attributes via input prompts. The get_values method is a getter that returns the current values of the attributes as a tuple. The delete_mileage method is a deleter that removes the mileage attribute from the instance. In the provided code, an instance car1 is created with initial values "BMW", 15, and "White". The __dict__ method is used to display the instance's attributes. After using set_values to update the attributes, the new values are retrieved and printed using get_values. Finally, delete_mileage is called to delete the mileage attribute, and the updated instance attributes are displayed again using __dict__.

Q. Create a class to verify the password. Define a setter method to ask the user to enter the password and set to the data member. Define a method to check the user entered password by comparing with the class variable. Give three chances to the user to enter the password. If enters correct password then print “Access Granted”, else print “Access Denied”.

```
class PasswordVerify:
    def __init__(self):
        self.password = None

    def set_password(self):
        password = input('Set the password')
        self.password = password

    def verify_password(self):
        attempts = 3
        while attempts > 0:
            entered_password = input('Enter the password')
            if entered_password == self.password:
                print("Access Granted")
                return
            else:
                attempts -= 1
                print(f"Incorrect password. You have {attempts} chances left.")

        print("Access Denied")

user1 = PasswordVerify()
user1.set_password()
user1.verify_password()
```

Iterator in a class

You can create an iterator within a class by implementing two special methods: __iter__ and __next__. The __iter__ method initializes and returns the iterator object, and the __next__ method returns the next item in the sequence.

Example:

```
class Number:
    def __init__(self):
        self.num = None

    def __iter__(self):
        self.num = 1
        return self

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

n1 = Number()
myiter = iter(n1)

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

The given code defines a Number class that implements the iterator protocol. The __init__ method initializes an instance variable num to None. The __iter__ method sets num to 1 and returns the instance itself, making it an iterable object. The __next__ method increments num by 1 and returns the previous value of num. When n1 is instantiated from Number and passed to iter(n1), it returns the instance itself as an iterator. Calling next(myiter) repeatedly invokes the __next__ method, which returns successive integers starting from 1. The output of the print statements will be 1, 2, 3, 4, and 5, demonstrating the iterator's behavior of generating an increasing sequence of numbers.

CostomRange in Python:

```
class CustomRange:
    def __init__(self, stop):
        self.current = 0
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        if self.current >= self.stop:
            raise StopIteration
        else:
            current_value = self.current
            self.current += 1
            return current_value

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

3. Aggregation and Composition

Aggregation is a concept in which an object of one class can own or access another independent object of another class. It represents Has-A's relationship.

Example:

```
class Salary:


    def __init__(self, pay):
        self.pay = pay


    def annual_salary(self):
        return self.pay*12


class Employee:


    def __init__(self, name, age, sal):
        self.name = name
        self.age = age


        self.salary = sal                    # Aggregation


    def total_sal(self):
        return self.salary.annual_salary()


salary = Salary(10000)
emp = Employee('Ram', 25, salary)


print(emp.total_sal())
```

In this code, the Salary class represents the salary details with a monthly pay attribute (pay) and a method to calculate the annual salary (annual_salary). The Employee class represents an employee with attributes like name, age, and a reference to a Salary object (salary). This demonstrates aggregation because the Employee class contains a reference to a Salary object, which exists independently of the Employee object. When an Employee object (emp) is created, it is passed an existing Salary object (salary). The total_sal method in the Employee class calculates the employee's total annual salary by calling the annual_salary method on the Salary object. This separation of concerns allows the Salary object to be reused or independently managed outside of the Employee class.

Composition:

Composition is a type of Aggregation in which two entities are extremely reliant on one another. It is own’s a relationship

Example:

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def annual_salary(self):
        return (self.pay*12)

class Employee:
    def __init__(self, name, age, pay):
        self.name = name
        self.age = age
        self.salary = Salary(pay) # composition

    def total_sal(self):
        return self.salary.annual_salary()

emp = Employee('Ram', 25, 10000)
print(emp.total_sal())
```

In this code, the Salary class represents salary details with a monthly pay attribute (pay) and a method to calculate the annual salary (annual_salary). The Employee class represents an employee with attributes like name, age, and a Salary object (salary). This demonstrates composition because the Employee class creates and owns a Salary object internally, passing the pay parameter to its constructor. The lifecycle of the Salary object is tightly coupled with the Employee object, meaning that when an Employee object (emp) is created, it internally creates a Salary object with the specified pay. The total_sal method in the Employee class calculates the employee's total annual salary by calling the annual_salary method on the internally created Salary object. This strong coupling ensures that each Employee object has its own dedicated Salary object.

4. Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent and child classes:

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

```
class Person:              #Parent Class


    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname


    def full_name(self):
        return f"{self.fname} {self.lname}"


class Employee(Person):     #child class
    def __init__(self, fname, lname, salary):
        Person.__init__(self, fname, lname)
        self.salary = salary
    def display_sal(self):
        return self.salary

class Student(Person):     #child class
    def __init__(self, fname, lname, percentage):
        Person.__init__(self, fname, lname )
        self.percentage = percentage

    def display_per(self):
        return self.percentage



emp_1 = Employee("Ram", "Thapa", 50000)
stu_1 = Student("Hari", "Thapa", 80.44)
print(f"Emp detailes: {emp_1.full_name()} {emp_1.display_sal()}")
print(f"Student detailes: {stu_1.full_name()} {stu_1.display_per()}")
```

In this code, the Person class is a parent class that holds common attributes fname and lname along with a method full_name to return the full name of the person. The Employee and Student classes are child classes that inherit from Person. The Employee class adds a salary attribute and a method display_sal to return the salary, while the Student class adds a percentage attribute and a method display_per to return the percentage. The constructors of the child classes use Person.__init__(self, fname, lname) to initialize the inherited attributes from the Person class. Instances of Employee and Student are created with specific attributes, and their details are printed using the methods defined in their respective classes.

The super() function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent. By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

```
class Person:           #Parent Class

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def full_name(self):
        return f"{self.fname} {self.lname}"

class Employee(Person):     #child class
    def __init__(self, fname, lname, salary):
        super().__init__(fname, lname)
        self.salary = salary

    def display_sal(self):
        return self.salary

class Student(Person):     #child class
    def __init__(self, fname, lname, percentage):
        super().__init__(fname, lname )
        self.percentage = percentage

    def display_per(self):
        return self.percentage
```

```
emp_1 = Employee("Ram", "Thapa", 50000)
stu_1 = Student("Hari", "Thapa", 80.44)
print(f"Emp detailes: {emp_1.full_name()} {emp_1.display_sal()}")
print(f"Student detailes: {stu_1.full_name()} {stu_1.display_per()}")
```

Method Overriding:

The Python method overriding refers to defining a method in a subclass with the same name as a method in its superclass. In this case, the Python interpreter determines which method to call at runtime based on the actual object being referred to.

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

```
class Person:           # Parent Class

    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def full_name(self):
        return f"{self.fname} {self.lname}"

class Employee(Person):     # Child class
    def __init__(self, fname, lname, salary):
        super().__init__(fname, lname)
        self.salary = salary

    def full_name(self):                              #method overriding
        return f"Employee: {self.fname} {self.lname}"

    def display_sal(self):
        return self.salary
```

```
class Student(Person):     # Child class
    def __init__(self, fname, lname, percentage):
        super().__init__(fname, lname)
        self.percentage = percentage

    def full_name(self):
        return f"Student: {self.fname} {self.lname}"

    def display_per(self):
        return self.percentage



emp_1 = Employee("Ram", "Thapa", 50000)
stu_1 = Student("Hari", "Thapa", 80.44)

print(f"{emp_1.full_name()} {emp_1.display_sal()}")
print(f"{stu_1.full_name()} {stu_1.display_per()}")
```

In this example, we have a parent class Person with a method full_name that returns the full name of a person. There are two child classes, Employee and Student, each inheriting from Person. Both child classes override the full_name method to prepend a specific label ("Employee: " or "Student: ") to the name. They also have additional attributes (salary for Employee and percentage for Student) and corresponding methods (display_sal and display_per) to return these values. When instances of Employee and Student are created and their details are printed, the output shows the customized full names and their specific details (salary and percentage) using the overridden full_name methods. This demonstrates the concept of method overriding in object-oriented programming, allowing child classes to provide specialized behavior for inherited methods.

Forms of Inheritance (Single, Hierarchical, Multiple, Multilevel)

- Single Inheritance: A class inherits from one and only one parent class.
- Hierarchical Inheritance: Multiple classes inherit from a single parent class.
- Multiple Inheritance: A class inherits from more than one parent class.
- Multilevel Inheritance: A class inherits from a parent class, which in turn inherits from another parent class, forming a chain of inheritance.

```
class Person:  # Parent Class
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def full_name(self):
        return f"{self.fname} {self.lname}"

class Employee(Person):  # Single Inheritance
    def __init__(self, fname, lname, salary):
        super().__init__(fname, lname)
        self.salary = salary

    def full_name(self):
        return f"Employee: {self.fname} {self.lname}"

    def display_sal(self):
        return self.salary

class Student(Person):  # Hierarchical Inheritance
    def __init__(self, fname, lname, percentage):
        super().__init__(fname, lname)
        self.percentage = percentage
    def full_name(self):
        return f"Student: {self.fname} {self.lname}"

    def display_per(self):
        return self.percentage

class Skills:  # Another Parent Class
    def __init__(self, skills):
        self.skills = skills

    def display_skills(self):
        return ", ".join(self.skills)

class WorkingStudent(Person, Skills):  # Multiple Inheritance
    def __init__(self, fname, lname, percentage, skills):
        Person.__init__(self, fname, lname)
        Skills.__init__(self, skills)
        self.percentage = percentage

    def full_name(self):
        return f"Working Student: {self.fname} {self.lname}"

class Intern(Employee):  # Multilevel Inheritance
    def __init__(self, fname, lname, salary, duration):
        super().__init__(fname, lname, salary)
        self.duration = duration

    def full_name(self):
        return f"Intern: {self.fname} {self.lname}"
    def display_duration(self):
        return self.duration

# Single Inheritance
emp_1 = Employee("Ram", "Thapa", 50000)
print(f"{emp_1.full_name()} with salary {emp_1.display_sal()}")

# Hierarchical Inheritance
stu_1 = Student("Hari", "Thapa", 80.44)
print(f"{stu_1.full_name()} with percentage {stu_1.display_per()}")

# Multiple Inheritance
work_stu = WorkingStudent("Sita", "Shrestha", 75.0, ["Python", "Data Analysis"])
print(f"{work_stu.full_name()} with skills {work_stu.display_skills()} and percentage
{work_stu.display_per()}")

# Multilevel Inheritance
intern_1 = Intern("Gita", "Rai", 30000, "6 months")
print(f"{intern_1.full_name()} with salary {intern_1.display_sal()} and duration
{intern_1.display_duration()}")
```

5. Polymorphism and dynamic binding

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example of inbuilt polymorphic functions:

- ➔ print(len("Hari"))   # len() being used for a string
- ➔ print(len([1, 2, 3, 4]))   # len() being used for a list

Dynamic binding, also known as late binding or runtime binding, is a concept where the method or attribute of an object is resolved at runtime rather than at compile time. This allows for more flexible and dynamic code, as the exact method to be invoked is determined only when the program is running. In Python, dynamic binding is a core feature due to its dynamic nature and support for polymorphism.

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def make_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_sound(dog)
make_sound(cat)
```

In the provided code, polymorphism is demonstrated through the make_sound function, which can accept objects of different classes (Dog and Cat) as long as they implement a speak method. This function allows for a single interface to work with different underlying forms, enabling the same function to call the speak method on both dog and cat objects, even though they are instances of different classes. Polymorphism allows make_sound to handle any object that implements the speak method, thus providing a unified way to invoke behavior across different types.

Dynamic binding is exemplified in how the make_sound function resolves the speak method at runtime based on the object's actual class. When make_sound(dog) is called, Python dynamically determines that dog is an instance of the Dog class and invokes the Dog class's speak method, returning "Woof!". Similarly, when make_sound(cat) is called, it invokes the Cat class's speak method, returning "Meow!". This runtime decision-making process, where the appropriate method is selected based on the object's type, highlights the concept of dynamic binding, allowing the same code to work with different object types seamlessly.

Note: Polymorphism in Python is primarily achieved through inheritance, method overriding, and special methods.

Abstract class and concrete class:

Abstract Class: An abstract class is a class that cannot be instantiated directly and is typically used to define a common interface for other classes. It can contain abstract methods that must be implemented by subclasses. Abstract classes are used to define a blueprint for other classes. They ensure that subclasses implement specific methods, enforcing a consistent interface.

To create an abstract class in Python, you use the abc module (Abstract Base Classes). You decorate abstract methods with @abstractmethod.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

    def move(self):
        return "Moving..."

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

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



dog = Dog()
print(dog.speak())
print(dog.move())

cat = Cat()
print(cat.speak())
print(cat.move())
```

Concrete Class: A concrete class is a class that can be instantiated and does not contain any abstract methods. It provides complete implementations for all its methods. In the previous example, Dog and Cat are concrete classes because they implement the speak method from the Animal abstract class and can be instantiated.