In [1]:
```
# 1. What are the five key concepts of Object-Oriented Programming (OOP)?
```

oops(obejcts oriented programing) basically it consists of class and objects and help us to create the code.It enhance the code saftey modularity and efficiency. oops have many concepts some of them are: 1 class 2 inheritance 3 encaplsuation 4 coupling 5 modularity 6 constructor.

In [31]:
```
# 2. Write a Python class for a `Car` with attributes for `make`, `model`, and `year`. Include a method to display the car's information.

class Car():
    def __init__(self, make, model, year):
        self.car_make = make
        self.car_model = model
        self.car_year = year
    def car1_det_make_model_year(self):
        print(self.car_make, self.car_model, self.car_year)
```

In [32]:
```
Car1 = Car("VERNA", "i5", "2021")
Car1.car1_det_make_model_year()
```

```
VERNA i5 2021
```

In [33]:
```
#  3. Explain the difference between instance methods and class methods. Provide an example of each.
```

Insatnce method is used to access or modify the object state.If we use instance variblae inside a method,these methods are called instance method. these methods have "sellf" parameters to refer to the current object # class method is used to access or modify the class state. In method implementation, if we use only class varibles,such types of method we should declare as a class method. it have a "cls" parameter which refer to the class.

In [23]:
```
# EX

class Animals:
    home = "zoo"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def animals_home(cls, home):
        cls.home = home
```

In [26]:
```
animal1 = Animals("lion", 5)
print(f'the original home is{animal1.home}')
Animals.animals_home("jungle")
print(f'the new animal home is{animal1.home}')
```

```
the original home iszoo
the new animal home isjungle
```

In [33]:
```
# ex of instance method

class Animals:
    home = 'zoo'
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def animals_home(cls, home):
        cls.home = home
```

In [34]:
```
def insta_method (self):
    self.home = "jungle"
    return  f'name: {self.name}, age: {self.age}, ' \
            f'location: {self.home}'
```

In [37]:
```
animal1 = Animals ("lion", 4)
```

In [38]:
```
print(animal1.insta_method())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[38], line 1
----> 1 print(animal1.insta_method())

AttributeError: 'Animals' object has no attribute 'insta_method'
```

In [39]:
```
# 4. How does Python implement method overloading? Give an example.
```

In [43]:
```
class Animal:
    def sound(self):
        print("sound of animal")

class cat(Animal):
    def sound(self):
        print("sound of cat")
```

In [46]:
```
anm = Animal()
anm.sound()
```

```
sound of animal
```

In [47]:
```
cat = cat()
cat.sound()
```

```
sound of cat
```

In [48]:
```
# 5. What are the three types of access modifiers in Python? How are they denoted?
```

3 types of modifiers are : public private protected

In [50]:
```
#public modifiers:

class Animals:
    home = "zoo"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In [53]:
```
a1 = Animals("monkey", 5)
a1.name
```

Out[53]:
```
'monkey'
```

In [64]:
```
# private modifiers: private data

class Animals:
    home = "zoo"
    def __init__(self, name, living):
        self.name = name
        self.living = living
    def show(self):
        print('name', self.name, 'living', self.__living)
```

In [65]:
```
a1 = Animals("monkey", "jungle")
a1.name
```

Out[65]:
```
'monkey'
```

In [67]:
```
a1.living
```

Out[67]:
```
'jungle'
```

In [69]:
```
a1.show()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[69], line 1
----> 1 a1.show()

Cell In[64], line 9, in Animals.show(self)
      8 def show(self):
----> 9     print('name', self.name, 'living', self.__living)

AttributeError: 'Animals' object has no attribute '_Animals__living'
```

In [70]:
```
# private method
```

In [71]:
```
class Animals:
    home = "zoo"
    def __init__(self, name, living):
        self.name = name
        self.living = living
    def show(self):
         print('name', self.name, 'living', self.__living)
    def __private_method(self):
        print("this is a private method")
```

In [73]:
```
a1 = Animals("monkey", "jungle")
a1.private_method()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[73], line 2
      1 a1 = Animals("monkey", "jungle")
----> 2 a1.private_method()

AttributeError: 'Animals' object has no attribute 'private_method'
```

In [75]:
```
# these can be access only by the class

a1._Animals__private_method()
```

```
this is a private method
```

In [88]:
```
class Collage:
    def __init__(self):
        self._collage_name = "mdu"

class student(Collage):
    def __init__(self, name):
        self.name = name
        collage.__init__(self)

    def show(self):
        print("name", self.name, "collage", self._collage_name)
```

In [89]:
```
a1 = student("robin")
a1.name
```

Out[89]:
```
'robin'
```

In [91]:
```
a1.show()
```

```
name robin collage mdu
```

In [92]:
```
# 6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.
```

In [93]:
```
# 1 single inhertance

class father:
    def father_pro(self):
        print("father properties")

class son(father):
    def job(self):
        print("son properties")
```

In [94]:
```
child = son()
```

In [96]:
```
child.job()
```

```
son properties
```

In [97]:
```
child.father_pro()
```

```
father properties
```

In [99]:
```
# 2 multilevel inheritance:

class grandf:
    def pro_grandf(self):
        print("properties of grandfather")
class father(grandf):
    def pro_father(self):
        print("properties of father")
class son(father):
    def pro_son(self):
        print("son properties")
```

In [100…
```
s = son()
```

In [101…
```
s.pro_father()
```

```
properties of father
```

In [102…
```
s.pro_grandf()
```

```
properties of grandfather
```

In [103…
```
s.pro_son()
```

```
son properties
```

In [111…
```
# 3 hierarchical inheritance:
class vehicle:
    def info(self):
        print("information of vehicle are")
class car(vehicle):
    def car_info(self, name):
        print("car information", name)
class truck(vehicle):
    def truck_info(self, name):
        print("truck information", name)
```

In [112…
```
c1 = car()
c1.car_info("jaguar")
```

```
car information jaguar
```

In [120…
```
t1 = truck()
t1.info()
```

```
information of vehicle are
```

In [121…
```
t1.truck_info("tata")
```

```
truck information tata
```

In [123…
```
# 4 hybrid inheritance
class veh:
    def vec_info(self):
        print("vec info")
class car(veh):
    def car_info(self):
        print("car info")
class truck(veh):
    def truck_info(self):
        print("truck info")
class sportscar(car, veh):
    def sportscar_info(self):
        print("scar info")
```

In [133…
```
obj1 = sportscar()
obj1.car_info()
```

```
car info
```

In [136…
```
obj1 = sportscar()
obj1.sportscar_info()
```

```
scar info
```

In [137…
```
obj1.vec_info()
```

```
vec info
```

In [138…
```
# 7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?
```

It occurs when a class inherits from 2 or more class to overcome for this problem python use "MRO" algorithm called l3 linearization. The class that is inherited first in the derived class, that method will be called or executed.

In [149…
```
class veh:
    def vec_info(self):
        print("vec info")
class car(veh):
    def car_info(self):
        print("car info")
class truck(veh):
    def truck_info(self):
        print("truck info")
class sportscar(car, veh):
    def sportscar_info(self):
        print("scar info")
```

In [151…
```
obj1 = sportscar()
obj1.car_info()
```

```
car info
```

In [152…
```
obj1.vec_info()
```

```
vec info
```

In [153…
```
obj1.sportscar_info()
```

```
scar info
```

In [1]:
```
# 8. Create an abstract base class `Shape` with an abstract method `area()`. Then create two subclasses `Circle` and `Rectangle` that implement the `area()` method.
```

In [29]:
```
import abc

class shape():

    @abc.abstractmethod
    def cal_area(self):
        pass


class circle(shape):
    def cal_area(self):
        return "area of circle pi r**2"
class rectangle(shape):
    def cal_area(self):
        return "area of rectangle l*b"
```

In [30]:
```
r = rectangle()
r.cal_area()
```

Out[30]:
```
'area of rectangle l*b'
```

In [31]:
```
c = circle()
c.cal_area()
```

Out[31]:
```
'area of circle pi r**2'
```

In [32]:
```
#  9. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas.
```

In [96]:
```
from abc import ABC, abstractmethod
import math


class shape(ABC):
    @abstractmethod
    def area(self):
        pass

class circle(shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi*(self.radius)**2

class rectangle(shape):
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def area(self):
        return self.length*self.breadth
```

In [97]:
```
def print_area(shape):
    print(f"the area for the shape: {shape.area():.2f}")
```

In [98]:
```
circle = circle(7)
rectangle = rectangle(12, 3)
```

In [99]:
```
print_area(circle)
print_area(rectangle)
```

```
the area for the shape: 153.94
the area for the shape: 36.00
```

In [78]:
```
#  10. Implement encapsulation in a `BankAccount` class with private attributes for `balance` and `account_number`. Include methods for deposit, withdrawal, and balance inquiry
```

In [137…
```
class Bankaccount:
    def __init__(self, account_no, start_balance = 0):
        self.__account_no = account_no
        self.__balance = start_balance

    def deposit (self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"deposted: ${amount:.2f}")
        else:
            print("deposited amount must be something")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"withdrew: {amount:.2f}")
        else:
            print("invalid withdraq")

    def get_balance(self):
        return self.__balance

    def get_account_no(self):
        return self.__account_no
```

In [138…
```
if __name__ == "__main__":
    account = Bankaccount("57625346", 700)

    print(f"account no: {account.get_account_no()}")
    print(f"start balance: {account.get_balance():.2f}")
```

```
account no: 57625346
start balance: 700.00
```

In [139…
```
account.deposit(500)
print(f"new balance: {account.get_balance():.2f}")
```

```
deposted: $500.00
new balance: 1200.00
```

In [140…
```
account.withdraw(200)
print(f"new balance: {account.get_balance():.2f}")
```

```
withdrew: 200.00
new balance: 1000.00
```

In [ ]:
```
#  11. Write a class that overrides the `__str__` and `__add__` magic methods. What will these methods allow you to do?
```

In [159…
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y +other.y)
        return noimplementation
```

In [160…
```
if __name__ == "__main__":
    m1 = Vector(2, 4)
    m2 = Vector(4, 7)

    print(m1)
    print(m2)

    m3 = m1 + m2
    print(m3)
```

```
Vector(2, 4)
Vector(4, 7)
Vector(6, 11)
```

In [ ]:
```
#  12. Create a decorator that measures and prints the execution time of a function.
```

In [ ]:

In [ ]:

In [ ]:
```
import time

def measure_execution_time(func):
    def wraper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        exe_time = end_time - start_time
        print(f"execution of{func.__name__}: {exe_time:4f} seconds")
        return result
    return wraper


@measure_execution_time
def example_function(n):
    total = 0
    for i in range(n):
        total += i
    return total

result = example_function(100)
```

In [ ]:

In [ ]:

In [ ]:
```
#  13. Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?
```

This problem occurs in python in multiple inheritance when a class from 2 or more class, it will lead to ambiguity in calling of method. To overcome from this problem python uses method resolution order(MRO) algorithm called l3 linearization.

In [2]:
```
class a:
    def methof(self):
        return"method a"

class b(a):
    def method(self):
        return"method b"

class c(a):
    def method(self):
        return"method c"

class d(b, c):
    pass
```

In [5]:
```
D = d()
```

In [7]:
```
print(D.method())
print(d.__mro__)
```

```
method b
(<class '__main__.d'>, <class '__main__.b'>, <class '__main__.c'>, <class '__main__.a'>, <class 'object'>)
```

In [8]:
```
#  14. Write a class method that keeps track of the number of instances created from a class.
```

In [15]:
```
class instancecounter:
    instance_count = 0
    def __init__(self):
        instancecounter.instance_count += 1
    @classmethod
    def get_instance_count(cls):
            return cls.instance_count
```

In [16]:
```
if __name__ == "__main__":
    obj1 = instancecounter()
    obj2 = instancecounter()

    print(f"no of instance created: {instancecounter.get_instance_count()}")
```

```
no of instance created: 2
```

In [17]:
```
# 15. Implement a static method in a class that checks if a given year is a leap year.
```

In [18]:
```
class yearutils:
    @staticmethod
    def is_leap_year(year):
        """check if if given year is a leap year"""
        if (year%4==0 and year%100!=0) or (year%400 == 0):
            return True
        return False

if __name__ =="__main__":
    year = 2004
    if yearutils.is_leap_year(year):
        print(f"{year} is a leap year")
    else:
        print(f"{year} is not a leap year")
```

```
2004 is a leap year
```

In [19]:
```
if __name__ =="__main__":
    year = 2010
    if yearutils.is_leap_year(year):
        print(f"{year} is a leap year")
    else:
        print(f"{year} is not a leap year")
```

```
2010 is not a leap year
```

In [ ]:

In [ ]: