1. What is the significance of classes in Python programming, and how do they contribute to object-oriented programming?

Classes in Python serve as blueprints for creating objects. They enable object-oriented
programming (OOP) by encapsulating data and methods into reusable structures. Classes
provide: - Encapsulation: Bundling attributes and methods together. - Inheritance: Reusing existing code via subclasses. - Polymorphism: Allowing different classes to define the same method differently. - Abstraction: Hiding implementation details to simplify code usage.

2. Create a custom Python class for managing a bank account with basic functionalities like deposit and withdrawal.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return f'Deposited {amount}. New balance: {self.balance}'

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance -= amount
        return f'Withdrawn {amount}. Remaining balance: {self.balance}'
```

3. Create a Book class that contains multiple Chapters, where each Chapter has a title and page count. Write code to initialize a Book object with three chapters and display the total page count of the book.

```
class Chapter:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

class Book:
    def __init__(self, title):
        self.title = title
        self.chapters = []
    def add_chapter(self, chapter):
        self.chapters.append(chapter)

    def total_pages(self):
        return sum(chap.pages for chap in self.chapters)

book = Book('Python Programming')
book.add_chapter(Chapter('Introduction', 20))
book.add_chapter(Chapter('OOP Basics', 30))
book.add_chapter(Chapter('Advanced OOP', 50))

print(f'Total pages in the book: {book.total_pages()}')
```

4. How does Python enforce access control to class attributes, and what is the difference between public, protected, and private attributes?

Python enforces access control through naming conventions: - Public attributes: Accessible from anywhere (e.g., `self.name`). - Protected attributes: Indicated by a single underscore (e.g., `_age`), implying it should be
accessed within subclasses. - Private attributes: Indicated by a double underscore (e.g., `__salary`), making it harder to
access directly outside the class.

5. Write a Python program using a Time class to input a given time in 24-hour format and convert it to a 12-hour format with AM/PM. The program should also validate time strings to ensure they are in the correct HH:MM:SS format. Implement a method to check if the time is valid and return an appropriate message.

```
class Time:
    def __init__(self, hh, mm, ss):
        self.hh = hh
        self.mm = mm
        self.ss = ss

    def is_valid(self):
        return 0 <= self.hh < 24 and 0 <= self.mm < 60 and 0 <= self.ss < 60

    def convert_to_12_hour(self):
        if not self.is_valid():
            return 'Invalid Time'
        period = 'AM' if self.hh < 12 else 'PM'
        hour = self.hh % 12 or 12
        return f'{hour}:{self.mm:02}:{self.ss:02} {period}'

time = Time(14, 30, 15)
print(time.convert_to_12_hour())
```

6. Write a Python program that uses private attributes for creating a BankAccount class. Implement methods to deposit, withdraw, and display the balance, ensuring direct access to the balance attribute is restricted. Explain why using private attributes can help improve data security and prevent accidental modifications.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.__balance = balance

    def deposit(self, amount):
        self.__balance += amount
        return f'Balance updated: {self.__balance}'

    def withdraw(self, amount):
        if amount > self.__balance:
            return 'Insufficient funds'
        self.__balance -= amount
        return f'Withdrawn {amount}. Balance: {self.__balance}'

account = BankAccount('Alice', 1000)
print(account.deposit(500))
print(account.withdraw(200))
```

7. Write a Python program to simulate a card game using object-oriented principles. The program should include a Card class to represent individual playing cards, a Deck class to represent a deck of cards, and a Player class to represent players receiving cards. Implement a shuffle method in the Deck class to shuffle the cards and a deal method to distribute cards to players. Display each player's hand after dealing. import random

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

```
class Deck:
   def __init__(self):
      self.cards = [Card(suit, value) for suit in ['Hearts', 'Diamonds', 'Clubs',
'Spades'] for value
in range(1, 14)]
      random.shuffle(self.cards)

   def deal(self, num_cards):
      return [self.cards.pop() for _ in range(num_cards)]

deck = Deck()
hand = deck.deal(5)
for card in hand:
   print(f'{card.value} of {card.suit}')
```

8. Write a Python program that defines a base class Vehicle with attributes make and model, and a method display_info(). Create a subclass Car that inherits from Vehicle and adds an additional attribute num_doors. Instantiate both Vehicle and Car objects, call their display_info() methods, and explain how the subclass inherits and extends the functionality of the base class.

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

   def display_info(self):
      return f'Vehicle: {self.make} {self.model}'

class Car(Vehicle):
   def __init__(self, make, model, num_doors):
      super().__init__(make, model)
      self.num_doors = num_doors

car = Car('Toyota', 'Camry', 4)
print(car.display_info())
```

9. Write a Python program demonstrating polymorphism by creating a base class Shape with a method area(), and two subclasses Circle and Rectangle that override the area() method. Instantiate objects of both subclasses and call the area() method. Explain how polymorphism simplifies working with different shapes in an inheritance hierarchy.

```
import math
```

```
class Shape:
   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, width):
      self.length = length
      self.width = width

   def area(self):
      return self.length * self.width

shapes = [Circle(5), Rectangle(4, 6)]
for shape in shapes:
   print(f'Area: {shape.area()}')
```

10. Implement the CommissionEmployee class with __init__, earnings, and __repr__ methods. Include properties for personal details and sales data. Create a test script to instantiate the object, display earnings, modify sales data, and handle data validation errors for negative values.

```
class CommissionEmployee:
   def __init__(self, name, sales, commission_rate):
      if sales < 0 or commission_rate < 0:
         raise ValueError('Sales and commission rate must be non-negative.')
      self.name = name
      self.sales = sales
      self.commission_rate = commission_rate

   def earnings(self):
      return self.sales * self.commission_rate

   def __repr__(self):
      return f'CommissionEmployee({self.name}, Sales: {self.sales}, Earnings:
{self.earnings()})'
```

```
emp = CommissionEmployee('Alice', 5000, 0.1)
print(emp)
```

11. What is duck typing in Python? Write a Python program demonstrating duck typing by creating a function describe() that accepts any object with a speak() method. Implement two classes, Dog and Robot, each with a speak() method. Pass instances of both classes to the describe() function and explain howduck typing allows the function to work without checking the object's type.

```
class Dog:
    def speak(self):
        return 'Bark'

class Robot:
    def speak(self):
        return 'Beep'

def describe(obj):
    return obj.speak()

dog = Dog()
robot = Robot()
print(describe(dog))
print(describe(robot))
```

12. WAP to overload the + operator to perform addition of two complex numbers using a custom Complex class.

```
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 __str__(self):
        return f'{self.real} + {self.imag}i'

c1 = Complex(2, 3)
c2 = Complex(4, 5)
print(c1 + c2)
```

13. WAP to create a custom exception class in Python that displays the balance and withdrawal amount when an error occurs due to insufficient funds? class InsufficientFunds(Exception):

```
    def __init__(self, balance, amount):
        super().__init__(f'Attempted to withdraw {amount}, but balance is only
{balance}')

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFunds(balance, amount)
    return balance - amount

try:
    withdraw(1000, 1500)
except InsufficientFunds as e:
    print(e)
```

14. Write a Python program using the Card data class to simulate dealing 5 cards to a player from a shuffled deck of standard playing cards. The program should print the player's hand and the number of remaining cards in the deck after the deal.

```
from dataclasses import dataclass
import random

@dataclass
class Card:
    suit: str
    value: str

class Deck:
    def __init__(self):
        self.cards = [Card(suit, value) for suit in ['Hearts', 'Diamonds', 'Clubs', 'Spades']
for value
in range(1, 14)]
        random.shuffle(self.cards)

    def deal(self, num_cards):
        return [self.cards.pop() for _ in range(num_cards)]

deck = Deck()
hand = deck.deal(5)
for card in hand:
    print(card)
```

15. How do Python data classes provide advantages over named tuples in terms of flexibility and functionality? Give an example using python code.

```
from dataclasses import dataclass
@dataclass
class Point:
x: int
y: int
p = Point(3, 4)
print(p)
```

16. Write a Python program that demonstrates unit testing directly within a function's docstring using the doctest module. Create a function add(a, b) that returns the sum of two numbers and includes multiple test cases in its docstring. Implement a way to automatically run the tests when the script is executed.

```
def add(a, b):
"""
>>> add(2, 3)
5
>>> add(-1, 1)
0
"""
return a + b
if __name__ == '__main__':
import doctest
doctest.testmod()
```

17. Scope Resolution: object's namespace → class namespace→ global namespace → built-in names- pace.

```
Species = Global Species
class Animal:
species = Class Species
def __init__ (self, species):
self.species = species
def display_species(self):
print("Instance species: self.species)
print("Class species:", Animal.species)
print("Global species:", globals()['species '])
a = Animal("Instance Species")
a.display _species()
```

What will be the output when the given scope resolution program is executed? Explain the scope resolution process step by step.

Explanation of Scope Resolution:
1. Instance attributes take precedence: `self.species` refers to the instance-level attribute.
2. Class attributes come next: If an instance attribute does not exist, `Animal.species` is used.
3. Global variables are checked last: If neither the instance nor class has the attribute, the
global `species` variable is used.
Expected Output:
```
Instance species: Instance Species
Class species: Class Species
Global species: Global Species
```

18. Write a Python program using a lambda function to convert temperatures from Celsius to Kelvin, store the data in a tabular format using pandas, and visualize the data using a plot.

```
import pandas as pd
import matplotlib.pyplot as plt
temps_celsius = [0, 10, 20, 30, 40]
temps_kelvin = list(map(lambda c: c + 273.15, temps_celsius))
df = pd.DataFrame({'Celsius': temps_celsius, 'Kelvin': temps_kelvin})
print(df)
df.plot(x='Celsius', y='Kelvin', kind='line')
plt.show()
```