**🔍 Theory of Inheritance in Python**

* **Parent/Base class**: The class whose properties and methods are inherited.
* **Child/Derived class**: The class that inherits from the parent class.
* A child class can:
  + Use parent class methods and attributes.
  + Override parent class methods.
  + Add new methods and attributes.

**📘 Types of Inheritance**

1. **Single Inheritance**: One child class inherits from one parent class.
2. **Multiple Inheritance**: One child class inherits from multiple parent classes.
3. **Multilevel Inheritance**: A child class becomes a parent for another class.
4. **Hierarchical Inheritance**: Multiple child classes inherit from the same parent.
5. **Hybrid Inheritance**: Combination of two or more types of inheritance.

**📜 Syntax**

class Parent:

# parent class

def method1(self):

print("This is method1 from Parent")

class Child(Parent):

# child class

def method2(self):

print("This is method2 from Child")

**🧪 Example Programs**

**1. Single Inheritance**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def bark(self):

print("Dog barks")

# Object of Dog

d = Dog()

d.speak() # Inherited method

d.bark() # Child's own method

**2. Multilevel Inheritance**

class Animal:

def eat(self):

print("Animal eats")

class Mammal(Animal):

def walk(self):

print("Mammal walks")

class Dog(Mammal):

def bark(self):

print("Dog barks")

d = Dog()

d.eat()

d.walk()

d.bark()

**3. Multiple Inheritance**

python

CopyEdit

class Father:

def show\_father(self):

print("Father's property")

class Mother:

def show\_mother(self):

print("Mother's property")

class Child(Father, Mother):

def show\_child(self):

print("Child's own")

c = Child()

c.show\_father()

c.show\_mother()

c.show\_child()

**4. Method Overriding**

class Parent:

def greet(self):

print("Hello from Parent")

class Child(Parent):

def greet(self): # Overriding the method

print("Hello from Child")

c = Child()

c.greet()

**🔧 Use of super()**

**The super() function allows you to call a method from the parent class.**

class Parent:

def show(self):

print("Parent's show")

class Child(Parent):

def show(self):

super().show()

print("Child's show")

c = Child()

c.show()

**✅ Benefits of Inheritance**

* Code Reusability
* Readability and structure
* Easier Maintenance and Scalability
* Promotes DRY principle (Don't Repeat Yourself)

**QUESTIONS**

For the following program, draw a UML class diagram that shows these classes and the relationships among them.

class PingPongParent:

pass

class Ping(PingPongParent):

def \_\_init\_\_(self, pong):

self.pong = pong

class Pong(PingPongParent):

def \_\_init\_\_(self, pings=None):

if pings is None:

self.pings = []

else:

self.pings = pings

def add\_ping(self, ping):

self.pings.append(ping)

pong = Pong()

ping = Ping(pong)

pong.add\_ping(ping)

Q-2> Write a Deck method called deal\_hands that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.

**🧪 Sample Code Implementation**

import random

class Card:

suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

ranks = [None, 'Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']

def \_\_init\_\_(self, suit=0, rank=2):

self.suit = suit

self.rank = rank

def \_\_str\_\_(self):

return f"{Card.ranks[self.rank]} of {Card.suits[self.suit]}"

class Deck:

def \_\_init\_\_(self):

self.cards = [Card(suit, rank) for suit in range(4) for rank in range(1, 14)]

random.shuffle(self.cards)

def pop\_card(self):

return self.cards.pop()

def add\_card(self, card):

self.cards.append(card)

def deal\_hands(self, num\_hands, cards\_per\_hand):

hands = [Hand(label=f"Hand {i+1}") for i in range(num\_hands)]

for i in range(cards\_per\_hand):

for hand in hands:

if self.cards:

hand.add\_card(self.pop\_card())

return hands

class Hand:

def \_\_init\_\_(self, label=''):

self.cards = []

self.label = label

def add\_card(self, card):

self.cards.append(card)

def \_\_str\_\_(self):

return f"{self.label}: " + ", ".join(str(card) for card in self.cards)

# Example usage:

deck = Deck()

hands = deck.deal\_hands(3, 5) # 3 players, 5 cards each

for hand in hands:

print(hand)

**📝 Output (example)**

Hand 1: 4 of Clubs, Jack of Hearts, ...

Hand 2: Ace of Diamonds, 6 of Spades, ...

Hand 3: 9 of Hearts, 2 of Clubs, ...

Q-2> The following are the possible hands in poker, in increasing order of value and decreasing order of probability:

pair: two cards with the same rank

two pair: two pairs of cards with the same rank

three of a kind: three cards with the same rank

straight: five cards with ranks in sequence (aces can be high or low, so Ace-2-3-4-5 is a straight and so is 10-Jack-Queen-King-Ace, but Queen-King-Ace-2-3 is not.)

flush: five cards with the same suit

full house: three cards with one rank, two cards with another

four of a kind: four cards with the same rank

straight flush: five cards in sequence (as defined above) and with the same suit

If you run PokerHand.py, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on. 3. Add methods to PokerHand.py named has\_pair, has\_twopair, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for “hands” that contain any number of cards (although 5 and 7 are the most common sizes). 4. Write a method named classify that figures out the highest-value classification for a hand and sets the label attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled “flush”. 5. When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in PokerHand.py that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear. 6. Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at

Soultion:-[Answer](https://thinkpython.com/code/PokerHandSoln.py%20.)