Pawel Stroka

R00198912

Q1: Creational Design Pattern (Prototype Design Pattern) [40 Marks]

```
import copy

original_list = [[1,2,3], [4,5,6]]

copy1 = copy.copy(original_list)



print("Original list: ", original_list)

for i in range(0, len(original_list[1])):
    copy1[1][i] += 10

print("\n\nShallow copy example")
print("Copy1: ", copy1)
print("Original list: ", original_list)
```

```
Original list:  [[1, 2, 3], [4, 5, 6]]


Shallow copy example
Copy1:  [[1, 2, 3], [14, 15, 16]]
Original list:  [[1, 2, 3], [14, 15, 16]]
```

In the above snapshots we can see a shallow copy example, copy1 variable contains the same contents as our original list.

We create a for loop that changes the content of the second inner number list by adding 10 to each number in it.

Then we print both variables, original_list and copy1

In this example if we change the contents of copy1, the changes are applied to original_list too, this is because copy1 and original_list point to the same space in memory where our contents are. Or we can say that copy1 is just a variable that points to original_list, they share the same contents.

The same goes for copy3 example.

```
copy2 = copy.deepcopy(original_list)

for i in range(0, len(original_list[1])):
    copy2[1][i] -= 5

print("\nDeep copy example")
print("Copy2: ", copy2)
print("Original list: ", original_list)
```

```
Deep copy example
Copy2:  [[1, 2, 3], [9, 10, 11]]
Original list:  [[1, 2, 3], [14, 15, 16]]
```

In the above snapshots we can see a deep copy of our original_list.

We create a variable copy2 that is a deep copy of our original list, then loop over the second inner list and subtract 5 from each number in it.

Deep copy unlike shallow copy, duplicates our list and assigns it to copy2, creating 2 independent copies of the same content in two different variables. With that even if we alter the contents of one list, the other list will not be affected by change.

Q2: Structural Design Pattern (Decorator Design Pattern) [30 Marks]

```python
class IValue:
    def operation(self, value):
        pass
```

Here we create an interface called IValue with a function “operation”, we leave the function blank, it’s an abstract function.

```python
class Value(IValue)     :
    def __init__(self, number=0):
        self.number = int(number)

    def operation(self, value):
        return self.number + value

    def __repr__(self):
        return str(self.number)
```

In here we create a Value class which implements our previously created interface, aside from the basic functions (init and repr) we have to also implement the operation function because we implemented the interface, the operation function add our input value to our number (which is 0 initially)

```python
class Add(IValue):
    def __init__(self, value):
        self.value = value

    def operation(self, base_value):
        if isinstance(self.value, IValue):
            return base_value + self.value.operation(0)
        else:
            return base_value + self.value


class Sub(IValue):
    def __init__(self, value):
        self.value = value

    def operation(self, base_value):
        if isinstance(self.value, IValue):
            return base_value - self.value.operation(0)
        else:
            return base_value - self.value
```

Then we create our Add and Sub(Subtract) classes which also implement our IValue interface. In both of them the operation function has an If-else statements as to accept integers, a value object, or other add/sub decorators. It checks if the given value is an instance of our IValue interface (that is if it's either our Value object, or any of our decorators since they both are an Instance of the IValue interface), or else for an Integer.

```
number = Value(10)
add = Add(10)
sub = Sub(5)

print("Original number: ", number)

add_result = add.operation(number.operation(0))
print("Add 10:", add_result)
```

Sample test code

And its respective output

```
Original number:  10
Add 10: 20
Sub 5: 5
```

Q3: Behavioral design pattern (Strategy Design Pattern) [30 Marks]

```python
class InvestmentStrategy():
    def buyStock(self, amount, current_price):
        pass
```

We start by creating our Interface class with one abstract function buyStock.

```python
class Aggressive(InvestmentStrategy):
    def buyStock(self, amount, current_price):
        current_price = current_price
        return f'{amount} has been invested at the price of {current_price}'


class Passive(InvestmentStrategy):
    def buyStock(self, amount, current_price):
        amount = amount / 2
        current_price = (current_price / 100) * 90
        return f'{amount} will be invested when the price is equal to {current_price}'
```

Then we create our investment strategies, they both implement our interface class and both have our buyStock function with each doing different logic

```python
class StockBuyer:
    def __init__(self, strategy):
        self.strategy = strategy

    def set_strategy(self, strategy):
        self.strategy = strategy

    def buyStock(self, amount, current_price):
        return self.strategy.buyStock(amount, current_price)
```

Then we create our StockBuyer class, with a strategy as a variable, it also has a buyStock function but it is different from our previous ones. This function uses our strategy variable (which is one of our InvestmentStrategy classes) and uses their corresponding buyStock function, and returns its output.

```
aggr = Aggressive()
passi = Passive()

stock_buyer = StockBuyer(aggr)

aggr_result = stock_buyer.buyStock(2000, 120)

print(aggr_result)

stock_buyer.set_strategy(passi)

passi_result = stock_buyer.buyStock(1500, 100)

print(passi_result)
```

Sample test code

And its sample output

```
2000 has been invested at the price of 120
750.0 will be invested when the price is equal to 90.0
```