Cheat Sheet

Inheritance - Part 2

How would you design and implement placing order with the details of all the products bought?

Composition

Modelling instances of one class as attributes of another class is called **Composition**

Code

PYTHON

```python
class Product:

    def __init__(self, name, price, deal_price, ratings):
        self.name = name
        self.price = price
        self.deal_price = deal_price
        self.ratings = ratings
        self.you_save = price - deal_price

    def display_product_details(self):
            print("Product: {}".format(self.name))
            print("Price: {}".format(self.price))
            print("Deal Price: {}".format(self.deal_price))
            print("You Saved: {}".format(self.you_save))
            print("Ratings: {}".format(self.ratings))

    def get_deal_price(self):
        return self.deal_price

class ElectronicItem(Product):
    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months

    def get_warranty(self):
        return self.warranty_in_months

class GroceryItem(Product):
    pass

class Order:
```

```
    def __init__(self, delivery_speed, delivery_address):
         self.items_in_cart = []
         self.delivery_speed = delivery_speed
         self.delivery_address = delivery_address

    def add_item(self, product, quantity):
        self.items_in_cart.append((product, quantity))

    def display_order_details(self):
        for product, quantity in self.items_in_cart:
            product.display_product_details()
            print("Quantity: {}".format(quantity))

    def display_total_bill(self):
        total_bill = 0
        for product, quantity in self.items_in_cart:
             price = product.get_deal_price() * quantity
             total_bill += price
        print("Total Bill: {}".format(total_bill))

milk = GroceryItem("Milk",40, 25, 3.5)
tv = ElectronicItem("TV",45000, 40000, 3.5)
order = Order("Prime Delivery", "Hyderabad")
order.add_item(milk, 2)
order.add_item(tv, 1)
order.display_order_details()
order.display_total_bill()
```

Collapse

Output

```
Product: Milk
Price: 40
Deal Price: 25
You Saved: 15
Ratings: 3.5
Quantity: 2
Product: TV
Price: 45000
Deal Price: 40000
You Saved: 5000
```

Expand

In the above example, we are modelling **Product** as attribute of **Order**

Overriding Methods

Sometimes, we require a method in the instances of a sub class to behave differently from the method in instance of a superclass.

Code

PYTHON

```python
class Product:

  def __init__(self, name, price, deal_price, ratings):
    self.name = name
    self.price = price
    self.deal_price = deal_price
    self.ratings = ratings
    self.you_save = price - deal_price

  def display_product_details(self):
      print("Product: {}".format(self.name))
      print("Price: {}".format(self.price))
      print("Deal Price: {}".format(self.deal_price))
      print("You Saved: {}".format(self.you_save))
      print("Ratings: {}".format(self.ratings))

  def get_deal_price(self):
    return self.deal_price

class ElectronicItem(Product):

  def display_product_details(self):
    self.display_product_details()
    print("Warranty {} months".format(self.warranty_in_months))

  def set_warranty(self, warranty_in_months):
    self.warranty_in_months = warranty_in_months

  def get_warranty(self):
    return self.warranty_in_months

e = ElectronicItem("Laptop",45000, 40000,3.5)
e.set_warranty(10)
e.display_product_details()
```

Collapse

Output

```
RecursionError: maximum recursion depth exceeded
```

Because

`self.display_product_details()` in ElectronicItem class does not call the method in the superclass.

Super

Accessing Super Class's Method

`super()` allows us to call methods of the superclass (Product) from the subclass.

Instead of writing and methods to access and modify warranty we can override

`__init__`

Let's add warranty of ElectronicItem.

Code

PYTHON

```python
class Product:

  def __init__(self, name, price, deal_price, ratings):
    self.name = name
    self.price = price
    self.deal_price = deal_price
    self.ratings = ratings
    self.you_save = price - deal_price

  def display_product_details(self):
      print("Product: {}".format(self.name))
      print("Price: {}".format(self.price))
      print("Deal Price: {}".format(self.deal_price))
      print("You Saved: {}".format(self.you_save))
      print("Ratings: {}".format(self.ratings))

  def get_deal_price(self):
    return self.deal_price

class ElectronicItem(Product):

  def display_product_details(self):
    super().display_product_details()
    print("Warranty {} months".format(self.warranty_in_months))

  def set_warranty(self, warranty_in_months):
    self.warranty_in_months = warranty_in_months

  def get_warranty(self):
    return self.warranty_in_months
```

```
31
32  e = ElectronicItem("Laptop",45000, 40000,3.5)
33  e.set_warranty(10)
34  e.display_product_details()
```

Collapse

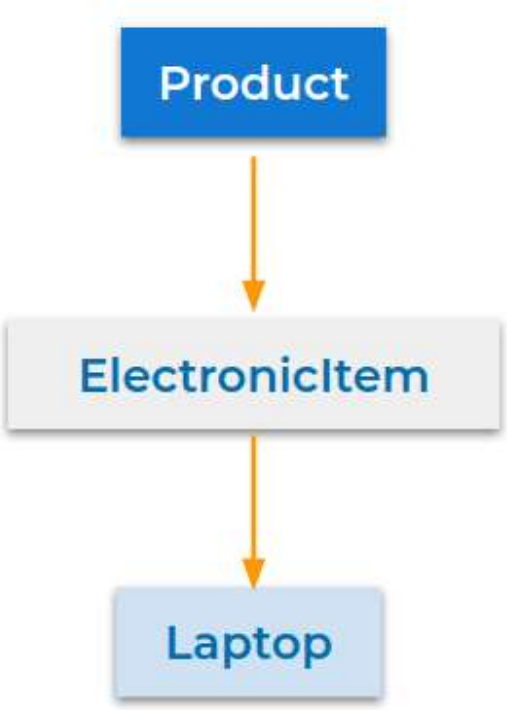
Output

```
Product: Laptop
Price: 45000
Deal Price: 40000
You Saved: 5000
Ratings: 3.5
Warranty 10 months
```

MultiLevel Inheritance

We can also inherit from a subclass. This is called **MultiLevel Inheritance**.

We can continue such inheritance to any depth in Python.



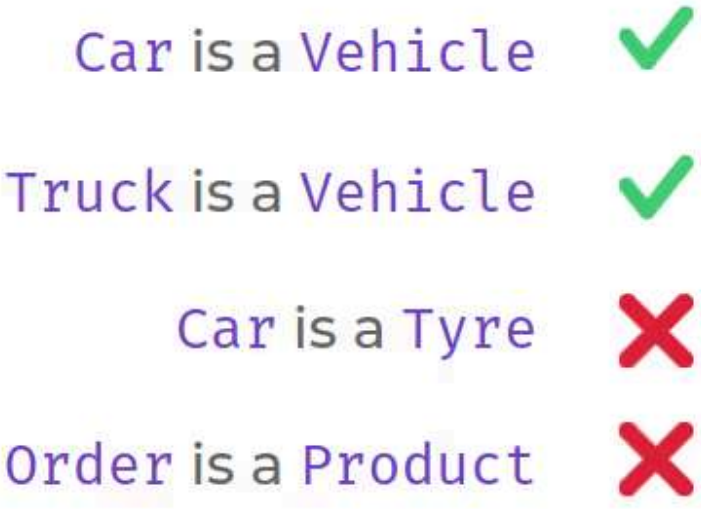
PYTHON

```python
class Product:
    pass

class ElectronicItem(Product):
    pass

class Laptop(ElectronicItem):
    pass
```

Inheritance & Composition

When to use Inheritance?

Prefer modeling with inheritance when the classes have an **IS-A** relationship.



When to use Composition?

Prefer modeling with inheritance when the classes have an **HAS-A** relationship.

Submit Feedback