Object Oriented Programming

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Recap

- Class defines types; objects can be created of this class type
- A class definition specifies attributes that keep the state of an object, and defines methods that can be invoked on objects
- So far we have looked at class definitions that use attributes of standard python types
- We will now look at composing classes

Combining Classes

- So far, we have looked at defining new types through a class definition - using mostly objects of standard python types
    - We have looked at how to make a class definition more general so it can be used for objects of different types using dunder methods
- A class can also use objects of other classes - this allows *composition* and ability to handle more complex problems
    - It enables creating complex types by combining objects of other types
    - Objects of a Composite class can contain an object of another class
- In composition, a class definition uses objects of other user defined classes
- Large problems often will have some Class defined for some type of objects, and then other classes will use these definitions

Composition...

- Composition of classes requires properly defining classes, so they can be easily used in other classes
- Creating a *composite* class using *component* classes, requires understanding of the component class attributes/methods (but not the implementation)
- Creating composite classes is a common approach for large applications using OOP - it also aids developing OO thinking
- We will illustrate composition through a few examples

Examples – Book Collection

```
class Book:
        def  __init__(self,  title,
author, price):
        self.title = title
        self.author = author
        self.price = price

class Member:
    def __init__(self, name, books):
        self.name = name
        self.books = books
    def add_book(self, book):
        self.books.append(book)
    def remove_book(self, book):
        self.books.remove(book)
```

```
class BookClub:
    def __init__(self, name, members):
        self.name = name
        self.members = members
    def add_member(self, member):
        self.members.append(member)
    def remove_member(self, member):
        self.members.remove(member)
    def find_book(self, book):
        for member in self.members:
            if book in member.books:
                return member
      def  transfer_book(self,  m1,  m2,
book):
        m1.remove_book(book)
        m2.add_book(book)
```

Book Collection

- We have a class Books - to define objects of type
- Have defined another class Member - to represent people who will be members of the book club - they own the books (so we have add / remove books for them)
- These two are indep classes - we can create objects of these types
- A BookClub has a name and a list of members - and has methods to add and remove members
- A BookClub allows one to search which of the members may have a book (so you can contact him/her), and also transfer of books from one member to another

Examples – Book Collection

```
if __name__ == "__main__":
    book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 10)
    book2 = Book("The Catcher in the Rye", "J. D. Salinger", 8)
    book3 = Book('The 4-Hour Workweek', 'Tim Ferriss', 15)
    book4 = Book('The Lean Startup', 'Eric Ries', 10)
    book5 = Book('The 7 Habits of Highly Effective People', 'Stephen Covey', 5)
    book6 = Book('The Business School', 'Robert Kiyosaki', 25)

    member1 = Member("David", [book2])
    member2 = Member("Aaron", [book3, book4, book5])
    member3 = Member("Emily", [book1, book6])

    book_club = BookClub("IIITD Book Club", [])
    book_club.add_member(member1)
    book_club.add_member(member2)
    book_club.add_member(member3)

    book = book1
    member = book_club.find_book(book)
    print(member.name) # Emily
    book_club.transfer_book(member3, member2, book)
    member = book_club.find_book(book)
    print(member.name) #Aaron
```

Example – Social Media

```
class Person:
    def __init__(self, username, age, bio, interests):
        self.username = username
        self.age = age
        self.bio = bio
        self.interests = interests

    def update_bio(self, new_bio):
        self.bio = new_bio

    def add_interest(self, new_interest):
        self.interests.append(new_interest)

    def remove_interest(self, interest):
        self.interests.remove(interest)

    def __str__(self):
        return f"username: {self.username}\nAge:
{self.age}\nBio: {self.bio}\nInterests:
{self.interests}"
```

```
class SocialGraph:
    def __init__(self):
        self.people = {}
    def add_person(self, person):
        self.people[person] = []
    def add_connection(self, person1, person2):
        if person1 in self.people:
            self.people[person1].append(person2)
        else:
            self.people[person1] = [person2]
        if person2 in self.people:
            self.people[person2].append(person1)
        else:
            self.people[person2] = [person1]
    def remove_connection(self, person1, person2):
        if person1 in self.people:
            self.people[person1].remove(person2)
        if person2 in self.people:
            self.people[person2].remove(person1)
    def get_common_friends(self, person1, person2):
        common_friends = []
        if person1 in self.people:
            for friend in self.people[person1]:
                if friend in self.people[person2]:
                    common_friends.append(friend)
        return common_friends
```

Example – Social Media

```
if __name__ == "__main__":
    graph = SocialGraph()

    user1 = Person("johan", 28, "I am a programmer", ["football", "coding"])
    user2 = Person("joe", 25, "I am a doctor", ["chess", "singing", "football"])
    user3 = Person("jane", 23, "I am a lawyer", ["dancing", "singing", "football"])
    user4 = Person("jim", 21, "I am a student", ["painting", "basketball"])

    graph.add_person(user1)
    graph.add_person(user2)
    graph.add_person(user3)
    graph.add_person(user4)

    graph.add_connection(user1, user3)
    graph.add_connection(user1, user4)
    graph.add_connection(user2, user3)
    graph.add_connection(user4, user2)

    user1.update_bio("I am a programmer and I love to code")
    user2.add_interest("skiing")
    query = graph.get_common_friends(user1, user2)
```

Example – Calendar

```
class Event:
    def __init__(self, name, date, time,
location):
        self.name = name
        self.date = date
        self.time = time
        self.location = location

    def update_date_time(self, date, time):
        self.date = date
        self.time = time

    def __str__(self):
        return self.name+" "+self.date+"
"+self.time+" "+self.location
```

```
class Calendar:
    def __init__(self):
        self.events = {
            "birthdays": [],
            "holidays": [],
            "meetings": [],
            "other": []
        }

    def add_event(self, event, category):
        if category in self.events:
self.events[category].append(event)

    def get_events(self, date):
        events = []
        for category in self.events:
            for event in
self.events[category]:
                if event.date == date:
                    events.append(event)
        return events
```

Example – Calendar...

```
if __name__ == "__main__":
    calendar = Calendar()
    event1 = Event("Rahul's Birthday", "12/12/22",
"12:00", "Home")
    event2 = Event("Christmas", "25/12/22",
"12:00", "Home")
    event3 = Event("Board Meeting", "7/12/22",
"17:00", "IIITD")
    event4 = Event("ML Seminar", "12/12/22",
"14:00", "IIITD")
    calendar.add_event(event1, "birthdays")
    calendar.add_event(event2, "holidays")
    calendar.add_event(event3, "meetings")
    calendar.add_event(event4, "other")
    query = calendar.get_events("12/12/22")
    for event in query:
        print(event)
```

Output

```
Rahul's Birthday
12/12/22 12:00 Home
ML Seminar 12/12/22
14:00 IIITD
```

Summary – Composition

- Classes define new Types - simple classes use objects of types defined in Python
- Classes can, however, also use objects of other class types - this allows composition, where a composite class is defined using objects of component types
- This allows for progressively building higher levels of abstraction / complexity using components
- Is a useful method for developing large OO programs

Quiz

Which of the following is/are true about composite classes in Python ?

(a) It combines the functionality of multiple classes into a single class.

(b) It reduces the complexity of the code and makes the execution faster.

(c) To create an object of a composite class, we need to call the constructor of each of the classes that make up the composite class.

(d) To create an object of a composite class, we need to call the constructor of the composite class.

Quiz : Solution

Which of the following is/are true about composite classes in Python ?

(a) It combines the functionality of multiple classes into a single class. ✅

(b) It reduces the complexity of the code and makes the execution faster. ❌

(c) To create an object of a composite class, we need to call the constructor of each of the classes that make up the composite class.❌

(d) To create an object of a composite class, we need to call the constructor of the composite class.✅

Solution : (a), (d)

Saving Object State

- Sometimes, we might want to save the current state of an object and retrieve it later for further processing
- Example - Progress in a Video Game
- We can do so by saving the object in a binary file.
- Binary files are are not human readable, unlike text files.
- Instead of “w” and “r”, we use “wb” and “rb” for reading binary files in Python. Eg:

```
open("my_file", 'wb')
open("my_file", 'rb')
```

- Pickle module is commonly used to dump/read a Python object to/from a binary file.

Saving Object State – Pickle

```python
class Student:
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score
        self.hobbies = []

    def add_hobby(self, hobby):
        self.hobbies.append(hobby)

    def remove_hobby(self, hobby):
        self.hobbies.remove(hobby)

    def update_score(self, score):
        self.score = score

    def __str__(self):
        return f"{self.name} is {self.age}
years old, has a score of {self.score} and
hobbies - {self.hobbies}"
```

```python
import pickle

Student1 = Student("David", 20, 70)
Student1.add_hobby("Cricket")
Student1.add_hobby("Football")
print(Student1)
pickle.dump(Student1, open("Student1.pkl", "wb"))

# After some time
Student1 = pickle.load(open("Student1.pkl", "rb"))
Student1.remove_hobby("Football")
Student1.update_score(80)
print(Student1)
pickle.dump(Student1, open("Student1.pkl", "wb"))
```

Note : The class definition should be present or imported in the Python file where the pickle file of Student1 is loaded

Accessing Object Attributes

Object oriented programming also promotes data encapsulation

I.e. data is encapsulated in an object and from outside you can only perform operations on the object

You do not need to know the nature of data of an object - or how the data is organized

In ideal OOP, the attributes (the data) are contained in the object and not directly visible to the user of the object

Some languages impose it, i.e. you cannot access the state of an object from outside

Accessing attributes…

How do you then get the value of an attribute or change the value

We can initialize them with __init__, but to change?

To get the value/state of an attribute, you need to have methods to return the value - often called getter methods

To set the value of an attribute - you need to have methods to set the value - called setter methods

You have to write these methods

Though python provides a method

Accessing attributes

Python allows access directly to users

E.g. c1 = Complex(2, 4)

Can access internals of c1 directly by

c1.imag or c1.real

Can set them also, e.g. c1.real = 6, c1.imag = 8

C1["imag"] = 8

C1["real"] = 6

Class vs Dictionary

Since you can access the object attributes, objects start looking more like dictionary

What you can do with class and objects - you can also do with dictionaries

The main difference is in style of programming - OO abstraction vs data structures and functions

In OO - you define a type, and then declare as many objects of that type as you need

Operations are part of the object

In dictionaries, each dictionary object will be separate, and you will have to write functions to work with them

Summary – Classes and Objects

- We have seen composite types can be created by using objects of component types - i.e. a new class is defined using other class definitions
  - With this, an object of composite class has objects of component classes
  - The relationship between the types is ‘***has’*** , i.e. a class C has objects of class D

```
student = {
    "rollno": "1234",
    "name": "Shyam",
    "sem1": [("m101", 4, 9),
("cs101", 4, 8), ("com101", 4,
10)],
    "sem2": [("m102", 4, 8),
("cs102", 4, 9), ("ee102", 4, 8)],
    "sem3": [("m202", 2, 10),
("cs201", 4, 8), ("elect1", 4, 10)],
    }
```

Inheritance (Just for Familiarization)

- Motivation - we have a class definition; we want to define a similar class which has more methods and more attributes
- We can define a new class, then the two classes are independent - and we will have to code the new class independently. Changes made in one class on parts that are common to both need to be copied over in the other class
- We can take the existing class, and "refine" it: by borrowing attributes and methods, and adding some more
- This helps in reusing the definitions of the existing class - a very useful property when writing big code
- Inheritance provides this facility

Inheritance

- With inheritance we define a new class using an existing class definition and specifying some modifications/enhancements to it
- I.e. we derive a new type (class) from an existing type (class)
- The new class is called derived class, child class or subclass, and the one from which it inherits is called the base class or parent/super class

Inheritance Syntax

```
class BaseClass:
    <body of base class>
    # attributes and methods

class DerivedClass (BaseClass):
    <body of derived class>
```

- The DerivedClass specifies the BaseClass it is inheriting from
- On inheriting, all attributes and methods of BaseClass are available to this class
- DerivedClass can add new features and new methods
- DerivedClass is said to *extend* the BaseClass

Objects of Base/Derived class

- A derived class can define additional attributes and methods
- An object of derived class has all attributes of base class, and can perform all operations of base class; + it has attributes of derived class and all its methods (*extends* the base class)
- So an object of derived class is also on object of base class - as has all the base class attributes and can perform those methods
- But an object of base class is not an object of derived class
- This defines an **"is-a"** relationship between the classes - an object of derived class is an object of the base class also, but not the other way around

Objects of Base/Derived class

- Creating an object of any base class is as with regular class
- Creating an object of derived class - the attributes of the base class automatically get defined, and methods become available
    - Often __init__() of base class called to initialize that part of state
- When an attribute is accessed/method performed on the object, python first searches in the derived class - if it exists that is used
- If it does not exist, then it searches in the base class - if it exists then it is used (this is applied recursively)
- (So, attributes/methods defined in derived class get selected over those defined in base class, if they have same names)

Inheritance

- Inheritance allows a new class to be defined by inheriting all properties of the base class and extending them
- It is used a lot in OO Programming
- In python, function-based programming is often used - it is more designed for this type of use
- Classes and objects are used when using libraries, packages, etc
  - E.g. strings, lists, etc - they are objects and we use many operations on them (all ops of the type <l>.op() are methods)
- Inheritance used only in bigger applications; some packages expect programmers to inherit classes and define new classes
- In IP you dont need to use inheritance

Quiz

What will be the output of the given code ?

Note: end='' is used when we do not want a new line after the print statement.

For example :

```
print("A", end='')
print("B")
```

```
Output: AB
```

```
class Parent:
    def __init__(self):
        print("Parent")
class Child(Parent):
    def __init__(self):
        print("Child", end='')
def main():
    obj1 = Child()
    obj2 = Parent()
main()
```

Quiz : Solution

What will be the output of the given code ?

Note: end='' is used when we do not want a new line after the print statement.

For example :

```
print("A", end='')
print("B")
```

```
Output: AB
```

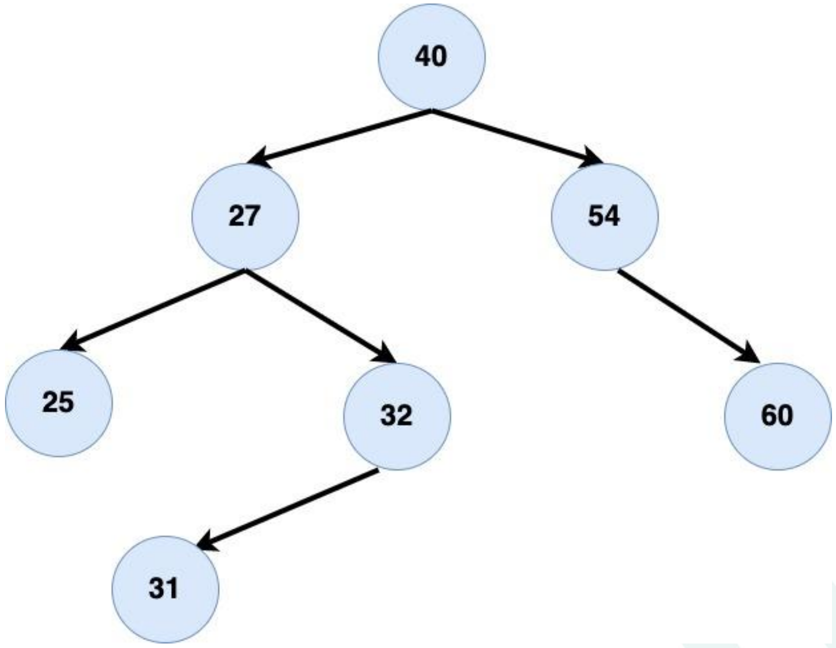
Solution: ChildParent

```
class Parent:
    def __init__(self):
        print("Parent")
class Child(Parent):
    def __init__(self):
        print("Child", end='')
def main():
    obj1 = Child()
    obj2 = Parent()
main()
```

Extra Slides

Example – Binary Search Tree

- Binary search tree - a root with value, and links to a tree on left and right
- Left sub-tree has values smaller than the root
- Right subtree has values greater than the root
- Searching for an element with binary search tree is very efficient - recursion is natural here
- Printing sorted items is easy



Implementing Binary Search Tree

- A binary search tree can be viewed as the root node of the tree
- Attributes: Value (of the root), ltree, rtree (None if no subtree)
- On creating a tree by init - we will create a root node
- Operations we desire: Insert a value, search for a value, print tree in sorted order

```
class Tree:
#Attributes: data, ltree, rtree
    def __init__(self, val):
        self.data = val
        self.ltree = None
        self.rtree = None
```

Binary Search Tree...

```
class Tree:
    #Attributes: data, ltree, rtree
    def __init__(self, val):
        self.data = val
        self.ltree = None
        self.rtree = None

    def insert(self, val):
        if val == self.data:
            return
        if val < self.data:
            if self.ltree is None:
                self.ltree = Tree(val)
            else:
                self.ltree.insert(val)
        elif val > self.data:
            if self.rtree is None:
                self.rtree = Tree(val)
            else:
                self.rtree.insert(val)
```

```
    def ispresent (self, val):
        if val == self.data:
            return  True
        if val < self.data:
            print("Search L subtree")
            if self.ltree is None:
                return False
            else:
                return self.ltree.ispresent(val)
        elif val > self.data:
            print("Search R subtree")
            if self.rtree is None:
                return False
            else:
                return self.rtree.ispresent(val)
```

Binary Search Tree – Inorder Traversal

```
# Print the values in tree in sorted order
def inorder(self):
    if self.ltree:
        self.ltree.inorder()
    print(self.data)
    if self.rtree:
        self.rtree.inorder()
```

Note: *An inorder traversal first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree).*

```
t = Tree(12)
t.insert(6)
t.insert(6)
t.insert(14)
t.insert(3)
t.insert(18)
print("Search 3", t.ispresent(3))
t.inorder()
```

```
Search L subtree
Search L subtree
Search 3 True
3
6
12
14
18
```

Binary Search Tree – Exercise

- A very common data structure used commonly in any operations requiring searching, sorting, etc
- Traversing the tree - like inorder() , we also have
  - preorder() - the root node visited first, then left sub tree, then right
  - postorder() - left subtree; right subtree; then the node is visited

- Extend the Class to add these methods (you can cut and paste the code in pythontutor or an IDE and then add)

Quiz

Which of these will print the contents of the tree in sorted order

A. inorder() # left first, node , then right
B. preorder() # node first, left, right
C. postorder() # right, node, left
D. None of the above

Quiz – Answer

Which of these will print the contents of the tree in sorted order

Ans: inorder() - it will print the smaller numbers first, then the root number, then the larger numbers

postorder() will print the tree contents sorted in reverse order

Searching

- We often have a collection of objects (integers, strings/words, word-meaning pairs, students, books, …)
  - We have seen List, Sets, Dictionaries data structures to store a collection of objects
  - We can create new ones also - e.g. a binary tree, a queue, …
- Searching if an item/object exists in a collection of objects is a very common function - e.g. search for a name, search for a word, …
  - All built-in data structures provide the *in* operation, or a method
- For a set or a dictionary, as they internally use hashing, searching is fast - you don't really have to "search" - you just check directly if the element is present (using hashing)
- Let us briefly look at how searching can be done when the collection of objects is kept as a list

Searching and Sorting

- If the list is not sorted - then we will have to search for an *item* by traversing the *lst* from start to end
  - Eg. for i in lst, if item==i, return True … return False at loop end
- Very slow - up to len(list) comparisons
- If we can sort the list - then searching is faster: we check with the middle element, if not found, search the left or the right
  - Can write this as a loop
  - Or write a recursive function for it
- Python provides a index function for finding the index - let us understand how they may work by writing our own

Searching in a sorted list

```
def search(arr, x):
    low, mid = 0
    high = len(arr) - 1
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1
```

Algorithm

- Find the mid-point of list, with low and high as end points
- If item is at mid - found
- If item < mid, search from low till mid
- If item > mid, search from mid to hig
- Repeat this till low and high are same
- Can write a recursive equivalent

Sorting

- Searching is facilitated if the list is sorted
- Hence, if we want to search for items frequently, we should keep the list sorted
- Python provides sort() method to sort the list
- Let us understand how sorting works - by looking at a simple approach to sort

Sorting

Algorithm

- Traverse the list till n-1 item
- Compare i, i+1 items - swap the bigger one to right
  - In one traversal biggest item will be the right most
- Then traverse list again but till n-2; keep repeating till we reach the start
- This algo is implemented as a nested loop

```
#bubble sort
def mysort(lst):
    n = len(lst)
    for i in range(n-1):
        for j in range(n-i-1):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
```

Sorting

- There are many sorting algorithms - it was an active area of research in the early days
- They require the use of nested loop - so sorting takes time
- There are some recursive algorithms also
- But now, sorting is provided in many languages (or libraries)
- It is best to use in-built sorted function
- If there are many insert or delete operations to be performed, as well as search - list will not be very efficient
    - After every insert/remove, we will have to sort
    - In this case organizing them as binary search tree is better

Extra slides – more examples

Additional Example: Online Shopping

- **Customer** can :
  - view products in the online store (all details excluding the quantity of stock available in the store)
  - add products to the shopping cart.
  - view their shopping cart.
  - remove products from the shopping cart.
  - checkout, clear the shopping cart, reset password,
  - edit profile information …
- The **system** should:
  - create a catalogue for the customer
  - compute bill on checkout
  - update the count of available products in store after purchase

Example : Online Shopping

```python
class Cart:
 def __init__(self):
    self.items = {} # product id : count

 def add_item(self, product_id, count):
    if product_id not in self.items:
      self.items[product_id] = 0
    self.items[product_id] =
self.items[product_id]+count

 def remove_item(self, product_id, count):
   if (self.items[product_id]-count)>0:
      self.items[product_id] =
self.items[product_id]-count
    else:
      del self.items[product_id]

 def get_items(self):
    return self.items

 def set_items(self, items):
    self.items = items
```

```python
class Product:
 def __init__(self, id, name, price):
    self.id = id
    self.name = name
    self.price = price

  def get_id(self):
    return self.id

  def set_id(self, id):
    self.id = id

  def get_name(self):
    return self.name

  def set_name(self, name):
    self.name = name

  def get_price(self):
    return self.price

  def set_price(self, price):
    self.price = price

  def __str__(self):
    return f'Product ID: {self.id}, Name:
{self.name}, Price: {self.price}'
```

Example : Online Shopping

```
class Customer:
 def __init__(self, id, name, phone, address, passwd):
    self.id = id
    self.name = name
    self.phone = phone
    self.address = address
    self.password = passwd
    self.cart = Cart() # Each customer has a shopping cart

  # View products available in the online store
 def viewProducts(self, catalogue):
      for product_id in catalogue:
        name = catalogue[product_id][0]
        price = catalogue[product_id][1]
        print(f'Product ID : {product_id}, Name : {name}, Price : Rs.
{price}')

  # Customer can view their shopping cart
 def viewCart(self, catalogue):
    cart_items = self.cart.get_items()
    for product_id in cart_items:
      name = catalogue[product_id][0]
      price = catalogue[product_id][1]
      count = cart_items[product_id]
      print(f'Product ID: {product_id}, Name:{name}, Price: Rs.
{price}, Count: {count}')

  #Customer can add products to shopping cart.
 def add_product2Cart(self, product_id, count):
      self.cart.add_item(product_id,count)
```

```
  # Customer class continued …
  # Given product id and count, remove from cart.
Here count means total count of the product to be
removed.
 def remove_productFromCart(self, product_id,
count):
    cart_items = self.cart.get_items()
    if product_id not in cart_items:
      print("Item not present in cart")
    elif cart_items[product_id]<count:
      print("These many items not available in
cart")
    else:
      self.cart.remove_item(product_id, count)


 def checkout(self):
    return self.cart

 # Getter-setter methods for attributes …

 def __str__(self):
    return f"Customer ID: {self.id}, Name:
{self.name}, Phone: {self.phone}, Address:
{self.address}"
```

Example : Online Shopping

```python
class ShoppingSystem:

 def __init__(self, products, customers):
    self.products = products # product details
    self.customers = customers # Customer details

 # Create a catalogue for customers.
 def create_catalogue(self):
    catalogue = {} # {product_id : [name, price]}
    for item in self.products:
      product_id = self.products[item][0].get_id()
      name = self.products[item][0].get_name()
      price = self.products[item][0].get_price()
      catalogue[product_id] = [name, price]
    return catalogue

 # Compute the bill on customer checkout.
 def compute_bill(self, cart):
    cart_items = cart.get_items()
    bill = 0
    for product_id in cart_items:
      item = self.products[product_id][0]
      count = cart_items[product_id]
      bill = bill + count*item.get_price()
    return bill
```

```python
 #ShoppingSystem class continued …

 # Update available stock after purchase
 def update_stock(self, cart):
    cart_items = cart.get_items()
    for product_id in cart_items:
      count = cart_items[product_id]
      self.products[product_id][1] =
self.products[product_id][1] - count

 def get_products(self):
    return self.products

 def set_products(self, products):
    self.products = products

 def get_customers(self):
    return self.customers

 def set_customers(self, customers):
    self.customers = customers
```

Example : Online Shopping

```python
# Create a dictionary of products
products = { 'p01': [ Product('p01', 'Shirt', 750), 100 ],
             'p02': [ Product('p02', 'Jeans', 800), 110 ],
             'p03': [ Product('p03', 'Python Book', 1200), 75 ],
             'p04': [ Product('p04', 'Pens', 20), 500 ],
             'p05': [ Product('p05', 'Cake (1kg)', 1000), 10 ]
             }

# View the products
for item in products:
  print('----------')
  print(products[item][0])
  print(f'Count: {products[item][1]}')
```

```
----------
Product ID: p01, Name: Shirt, Price: 750, Count: 100
----------
Product ID: p02, Name: Jeans, Price: 800, Count: 110
----------
Product ID: p03, Name: Python Book, Price: 1200, Count: 75
----------
Product ID: p04, Name: Pens, Price: 20, Count: 500
----------
Product ID: p05, Name: Cake (1kg), Price: 1000, Count: 10
```

```python
# Create an instance of the customer class
user = Customer(id = 'c01', name = 'Abc', phone =
'9876543210', address = 'IIIT Delhi', passwd = 'user')

# View the customer details
print(user)
```

```
Customer ID: c01, Name: Abc, Phone: 9876543210,
Address: IIIT Delhi
```

```python
# Create a dictionary for customers
customers = {'c01' : user}

# Create an instance of the shopping system
shoppersZone = ShoppingSystem(products, customers)

# Create the Catalogue
catalogue = shoppersZone.create_catalogue()
print(catalogue)
```

```
{'p01': ['Shirt', 750],
 'p02': ['Jeans', 800],
 'p03': ['Python Book', 1200],
 'p04': ['Pens', 20],
 'p05': ['Cake (1kg)', 1000]}
```

Example : Online Shopping

```
# Customer can view products available in the online store.
user.viewProducts(catalogue)
```

```
Product ID : p01, Name : Shirt, Price : Rs. 750
Product ID : p02, Name : Jeans, Price : Rs. 800
Product ID : p03, Name : Python Book, Price : Rs. 1200
Product ID : p04, Name : Pens, Price : Rs. 20
Product ID : p05, Name : Cake (1kg), Price : Rs. 1000
```

```
# Customer can add items to the shopping cart
user.add_product2Cart('p03',1)
user.add_product2Cart('p04',5)
user.add_product2Cart('p05', 1)
user.viewCart(catalogue)
```

```
Product ID: p03, Name:Python Book, Price: Rs. 1200, Count: 1
Product ID: p04, Name:Pens, Price: Rs. 20, Count: 5
Product ID: p05, Name:Cake (1kg), Price: Rs. 1000, Count: 1
```

```
# Customer removes 2 pens from the cart
user.remove_productFromCart('p04', 2)
user.viewCart(catalogue)
```

```
Product ID: p03, Name:Python Book, Price: Rs. 1200, Count: 1
Product ID: p04, Name:Pens, Price: Rs. 20, Count: 3
Product ID: p05, Name:Cake (1kg), Price: Rs. 1000, Count: 1
```

```
# Customer adds 4 more pens to the cart
user.add_product2Cart('p04', 4)
user.viewCart(catalogue)
```

```
Product ID: p03, Name:Python Book, Price: Rs. 1200, Count: 1
Product ID: p04, Name:Pens, Price: Rs. 20, Count: 7
Product ID: p05, Name:Cake (1kg), Price: Rs. 1000, Count: 1
```

```
# Customer can checkout
# Get the shopping cart
cart = user.checkout()
# Compute the bill
print(f'Please pay Rs. {shoppersZone.compute_bill(cart)}')
```

```
Please pay Rs. 2340
```

```
# After the payment is done, update the stock in the store
shoppersZone.update_stock(cart)
# Now the available stock in the store
products = shoppersZone.get_products()
for item in products:
  print('----------')
  print(f'{products[item][0]}, Count: {products[item][1]}')
```

```
----------
Product ID: p01, Name: Shirt, Price: 750, Count: 100
----------
Product ID: p02, Name: Jeans, Price: 800, Count: 110
----------
Product ID: p03, Name: Python Book, Price: 1200, Count: 74
----------
Product ID: p04, Name: Pens, Price: 20, Count: 493
----------
Product ID: p05, Name: Cake (1kg), Price: 1000, Count: 9
```