CLASS VARIABLES LAB 9

1. create a class

In [34]:
```
class Employee:
    no_of_emp=0 #class variables
    raise_amount=1.04

    def __init__(self,first,last,pay):
        self.first=first #instance variables
        self.last=last
        self.pay=pay

    no_of_emp+=1
    def apply_raise(self):
        self.pay=int(self.pay*Employee.raise_amount)
```

2.to create an instance and use class variables to track the number of instances

In [19]:
```
class Employee:
    raise_amount = 1.04
    no_of_emp = 0

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

        # Increment the class variable whenever a new instance is created.
        Employee.no_of_emp += 1

    def apply_raise(self):
        # Access the raise amount using the class itself or the instance.
        self.pay = int(self.pay * self.raise_amount)

# Create employee instances
emp_1 = Employee('Corey', 'Schafer', 50000)
```

```
emp_2 = Employee('Test', 'User', 60000)

# Check the total number of employees
print(Employee.no_of_emp)

# Apply a raise to an employee's salary
emp_1.apply_raise()
print(emp_1.pay)
```

```
2
52000
```

3.accessing attributes through instance variables

In [14]:
```
e1=Employee("reena","sri","70000") # creating an instance passing variables
```

In [15]:
```
print(e1.pay) #accessing pay instance variable
```

```
70000
```

4.print class variable before and after creating or incrementing through instance

In [16]:
```
print(e1.raise_amount)  # check if instance variable can access class variables
print(Employee.raise_amount) # accessing class variable wrt class
```

```
1.04
1.04
```

In [10]:
```
print(e1.__dict__) # displays the namespace of the instance
```

```
{'first': 'reena', 'last': 'sri', 'pay': '70000'}
```

In [17]:
```
e1.raise_amount=1.05 # assigning new value to the class variable through instance variable
```

after updation

In [18]:
```
print(e1.raise_amount) # it gets updated to the new value
print(Employee.raise_amount) # it doesnt get updated to the new value
```

```
1.05
1.04
```

changes the scope of the variable;

In [13]:
```
print(e1.__dict__) # displays the namespace of the instance, now it includes the class variable after it modifies
```

```
{'first': 'reena', 'last': 'sri', 'pay': '70000', 'raise_amount': 1.05}
```

BASIC INHERITANCE

In [20]:
```
class Parent: #creating a simple class
    def greet(self):
        print("Hello from the parent")
class Child(Parent): # inheritance syntax
    pass #skeleton not defined without any syntax errors
obj=Child() #creating an instance
obj.greet() #child class accessing the functin defined in parent class
```

```
Hello from the parent
```

1. create a class that inherits Employee class

In [21]:
```
class Child_Emp(Employee):
    pass
```

In [24]:
```
obj2=Child_Emp("hi","heloo",90000) #can pass arguments for parent class through child
print(obj2.raise_amount) #accessing the class variable of parent class , but cannot be accessed without calling ini
```

```
1.04
```

2. using super () function to access the parent class

In [33]:
```
class Parent: #creating a simple class
    def greet(self):
        print("Hello from the parent")
class Child(Parent): # inheritance syntax
```

```
    def greets(self):
        super().greet() # calling the menthod of parent class within a child class
        print("Hello from the Child")

obj=Child() #creating an instance
obj.greet() #calling method of parent class
```

```
Hello from the parent
```

In [32]:
```
obj.greets() #calling method of child class
```

```
Hello from the parent
Hello from the Child
```

3. try to access the apply_raise function using super() function

In [35]:
```
class Child_Emp(Employee):
    def access(self):
        super().apply_raise() #accesing apply_raise function using super() function
```

In [36]:
```
obj3= Child_Emp("michel","jackson",500000)
obj3.access()
print(obj3.pay)
```

```
520000
```

super() func is useful where the child and parent can have the functions with same name

MULTIPLE INHERITANCE

In [37]:
```
class A:
    def method_a(self): #parent 1
        print("Method A")

class B:
```

```
    def method_b(self): #parent 2
        print("Method B")

class C(A,B): #syntax for multiple inheritance
    pass

obj=C()
obj.method_a() # accessing class A method with obj C
obj.method_b() # accessing class B method with obj C
```

```
Method A
Method B
```

4.) try to make the multi level inheritance like this

In [38]:
```
class A:
    def method_a(self): #parent of B
        print("Method A")

class B(A):
    def method_b(self): #parent of C
        print("Method B")
class C(B): #syntax for multi-level inheritance
    pass
obj=C()
obj.method_a() # accessing class A method with obj C
obj.method_b() # accessing class B method with obj C
```

```
Method A
Method B
```

understanding the uni-directional property of inhert classes

In [42]:
```
class Contact: # parent class
    all_contacts=[]

    def __init__(self,name,email):
        self.name=name
```

```
        self.email=email
        Contact.all_contacts.append(self)
```

In [43]:
```
class Supplier(Contact): #child class
    def order(self,order):
        print("send""{} order to {}".format(order,self.name))
```

In [44]:
```
c=Contact("Some Body","somebody@example.net") #creating an instance of parent class
s=Supplier("Sup Plier","supplier@example.net") #creating an instance of child class
print(c.name,c.email,s.name,s.email) #printing values of respectve classes
```

```
Some Body somebody@example.net Sup Plier supplier@example.net
```

In [45]:
```
c.all_contacts # accessing the class variable
```

Out[45]:
```
[<__main__.Contact at 0x7165fcae4830>, <__main__.Supplier at 0x7165fcae5340>]
```

In [46]:
```
c.order("pliers") # it is a single way and parent cannot access the methods of child
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[46], line 1
----> 1 c.order("pliers")

AttributeError: 'Contact' object has no attribute 'order'
```

In [47]:
```
s.order("plier") #counter-measure
```

```
sendplier order to Sup Plier
```

EXTENDING THE BUILD-IN CLASS

In [7]:
```
class ContactList(list):
    def search(self,name):
        """ return all contact that contain the search value in their name."""
        matching_contacts=[]
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
            return matching_contacts
```

In [10]:
```
class Contact:
    all_contacts=ContactList() # Shared list of all contacts
    def __init__(self,name,email):
        self.name=name
        self.email=email
        self.all_contacts.append(self) #Add self to the shared contact list
```

In [12]:
```
c1=Contact("John A","johna@example.net")
c2=Contact("John B","johnb@example.net")
c3=Contact("Jenna C","jennac@example.net")
[c.name for c in Contact.all_contacts.search('John ')]
```

Out[12]:
```
['John A']
```

In [ ]: