Principles of OOP

Created by: Florensia Widjaja (122040013) for CSC1001 Tutorial 9 Fall 2023 Semester
A good resource worth-checking-out: https://jeemariyana.medium.com/oop-concepts-with-real-world-examples-cda1cd277f4f

Abstraction

You know there are so many interesting features in your phone, such as making phone calls, downloading many different varieties of applications. But do you know **HOW** your phone is able to do all of those cool features? No right? That's basically what abstraction is.

Another example is when you are driving a car, do you know how the brakes or gas pedals work? No. But just by **KNOWING THE FUNCTION** of the brakes and the gas pedals, you are already prepared to use the car. Same thing with abstraction. Imagine that

- Class : All cars
- Your Tesla car : one instance/objects of the Car Class
- The gas and pedals inside your car: Methods/Functions of the Car Class

Encapsulation

Encapsulation is packing up data and methods to one single unit (class).

In abstraction, we explain that you only need to learn the FUNCTION of each methods in a class, which makes it easier right? but sometimes it is also beneficial to the companies who intentionally want to hide the data. Therefore, the users can only change and retrieve the instances attributes they have access to.

Basically they cannot just get their password directly by name.password but they must have the **permission** (which is by defining the getter to become a public method) by the company to have access to their password

```python
class Account:
    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def setUsername(self, newUsername): # this is what's called setter
        self.__username = newUsername
    def setPassword(self, newPassword):
        self.__password = newPassword

    def getUsername(self):  # this is what is called getter
        return self.__username
    def getPassword(self):
        return self.__password
    def __getPassword(self):  # if the getter is set to be PRIVATE then the external users cannot have access to the passwords
        return self.__password
```

Inheritance (the is-a relationship)

For example, you have make users for the employees in your store. There is person who works as a cashier, as a manager, or as a sales person. Every one of these people have the ability to eat, sleep, have to come to work at the same time. Therefore, instead of making three 'eat', 'sleep', and 'come_to_work_ontime' methods for each classes, **we define an 'employee' class, and let the 'manager', 'sales', and 'cashier' class INHERIT from this 'employee' class**.

Example code in below of method overriding

Dynamic Binding

The same method implemented in many classes. Dynamic binding **DETERMINES WHICH METHOD TO INVOKE**

```
class Car:
    def drive(self):
        self.go()
    def go(self):
        print("go in car")

class Tesla(Car):
    def go(self):
        print("go in Tesla")

def main():
    d = Car()
    d.go()
    d.drive()

    t = Tesla()
    t.go()  # t inherits the go method in Car
    t.drive()  # this is dynamic binding. It calls the INHERITED
METHOD DRIVE and then this Drive method CALLS GO FROM TESLA

main()
```

```
go in car
go in car
go in Tesla
go in Tesla
```

Polymorphism (One kind of dynamic binding)

Polymorphism is the ability of an object to take on many forms.
a.k.a the object of different classes can be passed as arguments to the same function

Overriding

One such example of dynamic binding.

The superclass (parent class) and subclass both has a method call x() then if subclass.x() is called --> the method in the subclass gonna be run instead of the one in superclass

come_to_work method below illustrates overriding

```
class Employee(object):  # know that any classes always inherit from
the Object class
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def eat(self):
        print(self.name + " has eaten today")
    def sleep(self):
        print(self.name + ' has gotten their well-deserved sleep :)')
    def come_to_work(self, time):
        if time > 8:  # work starts at 8 am so if he comes more than
8, he is late
            print(self.name + ' comes to work late :(')
        else:
            print(self.name + " comes to work on time")


class Sales(Employee):
    # because we don't define another initialization/constructor, the
sales will inherit
    # __init__ function from employee class
    def make_sales_events(self, time):
        print(self.name + " will make a sales event at " + time)
    def salary(self):
        print(self.name + " has 70.000 yuan salary")

class Cashier(Employee):
    # because we don't define another initialization/constructor, the
cashier will inherit
    # __init__ function from employee class
    def report_income(self, income):
        print(self.name + "collected " + income + " yuan today.")
    def salary(self):
        print(self.name + "has 80.000 yuan salary")

class Manager(Sales):  # if it inherits from sales, then this is a
SALES MANAGER
    # because we don't define another initialization/constructor, the
managers will inherit
    # __init__ function from employee class
    def make_meetings(self, time):
        print("this manager will make a meeting at " + time)
    def salary(self):
        print(self.name + "has 100.000 yuan salary")
    def come_to_work(self, time):
        if time > 10:  # work starts at 8 am so if he comes more than
8, he is late
            print(self.name + ' comes to work late :(')
            return -1 # -1 indicates late
        else:
            print(self.name + " comes to work on time")
            # 1 indicates ontime

def compare_employee(emp1,emp2):  # POLYMORPHISM: WE DON'T KNOW WHAT
EMP1 AND EMP2 IS, THEY CAN BE MANAGER, SALES, OR CASHIER.
    if emp1.come_to_work > emp2.come_to_work:
        print(emp1.name + " is better than " + emp2.name)
    elif emp1.come_to_work < emp2.come_to_work:
        print(emp2.name + " is better than " + emp1.name)
    else:
        print(emp1.name + " have the same performance as " +
emp2.name)
def main():
    mike = Cashier('Mike', 22)
    britany = Manager('Britany', 24)
    mike.salary()
    print(britany.salary())

    # METHOD OVERRIDING (because it has the SAME NAME AND SAME METHOD
ARGUMENT)
    # DYNAMIC BINDING is deciding which come_to_work method should be
invoked BASED ON THE INHERITENCE CHAIN
    britany.come_to_work(9)  # runs the method in the subclass and not
in the superclass
    mike.come_to_work(9)  # runs method of superclass

main()
```

```
Mikehas 80.000 yuan salary
Britanyhas 100.000 yuan salary
None
Britany comes to work on time
Mike comes to work late :(
```

Exercises

Question 1

```
## Question 1.1.
class A:
    def __new__(self):
        print("A's __new__() invoked")
    def __init__(self):
        print("A's __init__() invoked")
```

```
class B(A):  # B inherits A
    def __new__(self):
        print("B's __new__() invoked")
    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main()
```

```
B's __new__() invoked
A's __new__() invoked
```

```
## Question 1.2.
class A:
    def __new__(self):
        self.__init__(self)
        print("A's __new__() invoked")
    def __init__(self):
        print("A's __init__() invoked")

class B(A):  # B inherits A
    def __new__(self):
        self.__init__(self)
        print("B's __new__() invoked")
    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main()
```

```
B's __init__() invoked
B's __new__() invoked
A's __init__() invoked
A's __new__() invoked
```

```
## Question 1.1.
class A:
    def __new__(self):
        self.__init__(self)
        print("A's __new__() invoked")
    def __init__(self):
        print("A's __init__() invoked")

class B(A):  # B inherits A
    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main()

B's __init__() invoked
A's __new__() invoked
A's __init__() invoked
A's __new__() invoked
```

Question 2

```
class A:
    def __init__(self,i=0,j=0):
        self.__i=i
        self.j=j
    def __m1(self):
        self.__i+=1
    def m2(self):
        return self.__i

class B(A):
    def __init__(self,i=1,j=1):
        super().__init__(i,j)

b=B()
##print(b.__i)    #①
print(b.j)      #②
##print(b.__m1()) #③
print(b.m2())   #④

1
1
```

Question 3

```
from math import sqrt

class Vector:
    def __init__(self,x=1,y=0):
        self.x=x
        self.y=y
    def __str__(self):
        return '('+str(self.x)+','+str(self.y)+')'
    def __add__(self,v):
        Newv=Vector()
        Newv.x=self.x+v.x
        Newv.y=self.y+v.y
        return Newv
    def __sub__(self,v):
        Newv=Vector()
        Newv.x=self.x-v.x
        Newv.y=self.y-v.y
        return Newv
    def __eq__(self,v):
        return self.norm()==v.norm()
    def __gt__(self,v):
        return self.norm()>v.norm()
    def __lt__(self,v):
        return self.norm()<v.norm()
    def norm(self):
        return sqrt(self.x**2+self.y**2)

a=Vector(2,5)
b=Vector(3,4)
print(a)
print(b)
c=a+b
print(c)
print(a.norm())
print(b.norm())
print(a==b)
print(a>b)
print(a<b)
```

```
(2,5)
(3,4)
(5,9)
5.385164807134504
5.0
False
True
False
```

Question 4: Inheritance Tree

Please Refer to the pdf file I sent to the OneDrive Link.

Question 5: Simple Game: Undercut and Question 6: Course Selection System

You can refer to the sample codes given. I don't see anything that can be improved on the sample code they gave so I'd suggest you to also refer to those codes.

```
## This code is exactly the same with the sample code in '5-A simple
game_Undercut.py'
class Player:
    def __init__(self,name='',score=0):
        self.name=name
        self.score=score
    def getName(self):
        return self.name
    def resetScore(self):
        self.score=0
    def increaseScore(self):
        self.score+=1
    def __str__(self):
        return str((self.name,self.score))
    def __repr__(self):
        return 'Player'+str(self)

import random
class Computer(Player):
    def __init__(self):
        super().__init__()
    def getMove(self):
        return random.randint(1,10)
    def __repr__(self):
        return 'Computer'+str(self)

class Human(Player):
    def __init__(self):
        super().__init__()
    def getMove(self):
        while True:
            try:
                n=int(input('Enter an integer:'))
                if n>=1 and n<=10:
                    return n
                else:
                    print('invalid input')
            except:
                print('invalid input')
    def __repr__(self):
        return 'Human'+str(self)

def main():
    c=Computer()
    c.name='MyPython'
    h=Human()
    h.name='Jay'
    print(playUnderCut(c,h))

def playUnderCut(p1,p2):
    p1.resetScore()
    p2.resetScore()
    m1=p1.getMove()
    m2=p2.getMove()
    if m1==m2-1:
        p1.increaseScore()
        return p1.getName()+' moves '+str(m1)+' '\
               +p2.getName()+' moves '+str(m2)+' '\
               +p1.getName()+' wins '
    elif m2==m1-1:
        p2.increaseScore()
        return p1.getName()+' moves '+str(m1)+' '\
               +p2.getName()+' moves '+str(m2)+' '\
               +p2.getName()+' wins '
    else:
        return p1.getName()+' moves '+str(m1)+' '\
               +p2.getName()+' moves '+str(m2)+' '\
               'draw: no winner'

main()
```

```
MyPython moves 10 Jay moves 4 draw: no winner
```

Sending my best wishes for your midterms 😊💪

ps. Don't forget to take care of yourselves too during these stressful period!