Python Fundamentals day 51

Today's Agenda

- Encapsulation

Encapsulation

To know what is encapsulation and is it needed. Let us consider an example

```
class AccountHolder:

    def __init__(self):
        self.bal=10000

ah=AccountHolder()
print(ah.bal)
ah.bal=20000
print(ah.bal)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
20000
```

Let us now try giving negative number and see if the value modifies

```
class AccountHolder:

    def __init__(self):
        self.bal=10000

ah=AccountHolder()
print(ah.bal)
ah.bal=-20000
print(ah.bal)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
-20000
```

But balance in ones' account is personal and should not be easily accessible by the third person. And more importantly we shouldn't be allowed to mishandle the data. Let us see how to resolve this issue

```
class AccountHolder:

    def __init__(self):
        self.bal=10000

    def get_bal(self):
        return self.bal

    def set_bal(self,amt):
        if amt>0:
            self.bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.get_bal())
ah.set_bal(-20000)
print(ah.get_bal())
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
Invalid Amount
10000
```

Encapsulation in Python is the process of wrapping up variables and methods into a single entity. In programming, a class is an example that wraps all the variables and methods defined inside it.

Encapsulation acts as a protective layer by ensuring that, access to wrapped data is not possible by any code defined outside the class in which the wrapped data are defined. Encapsulation provides security by hiding the data from the outside world.

But in the above code we can still access amt by calling bal and can still modify. How to avoid it?

```
class AccountHolder:

    def __init__(self):
        self._bal=10000 # _ represents not to access directly
                        # or symbolises as private variable
    def get_bal(self):
        return self._bal

    def set_bal(self,amt):
        if amt>0:
            self._bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.get_bal())
ah.set_bal(-20000)
print(ah.get_bal())
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
Invalid Amount
10000
```

Although the above declaration of variable is just indication of not using directly, which means any third member can still aces it directly who isn't aware of it. Which says that python doesn't strictly enforce anything it's more of a responsibility of the user and developer. Is there a solution for this? There certainly is let us see

```
class AccountHolder:

    def __init__(self):
        self._bal=10000 # _ represents not to access directly
                        # or symbolises as private variable
    def get_bal(self):
        return self._bal

    def set_bal(self,amt):
        if amt>0:
            self._bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.__dict__)  #checks if bal is stored in dict
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_bal': 10000}
```

We confirmed that bal is stored in dictionary, but still we haven't resolved the situation completely

```
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.__dict__)  #checks if bal is stored in dict
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_AccountHolder__bal': 10000}
```

By adding 2 underscore in front of the variable made the difference. Python internally changed the __bal to _AccountHolder__bal using a concept called as **data mangling**.

There is no strict way to stop the access there are only few tricks which we can work around with.

Now let us see another example of data mangling

```
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
ah.__bal=20000
print(ah.__dict__)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_AccountHolder__bal': 10000, '__bal': 20000}
```

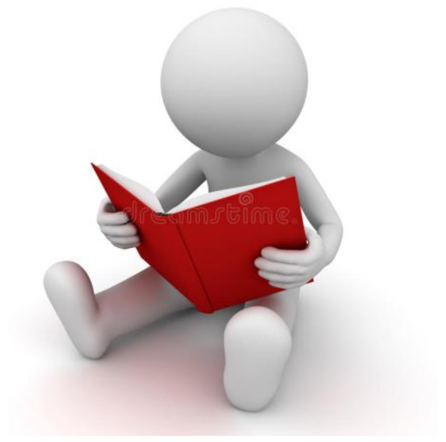
We see that new variable is created with name __bal. This is because mangling is a process which is only applied to instance variables which are created within the class.

This is how internally new variable got created

```
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
ah.__bal=20000 #ah.__dict['__bal']=20000
print(ah.__dict__)
```

The above code will give the same output.

Let us see another example



```python
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        print('get_bal() called')
        return self.__bal

    def set_bal(self,amt):
        print('set_bal called')
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

    bal=property(get_bal,set_bal)

ah=AccountHolder()
print(ah.bal)
ah.bal=20000
print(ah.bal)
```

There's another way to do the same.

```python
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    @property
    def bal(self):
        print('get_bal() called')
        return self.__bal

    @bal.setter
    def bal(self,amt):
        print('set_bal called')
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

    #bal=property(get_bal,set_bal)

ah=AccountHolder()
print(ah.bal)
ah.bal=20000
print(ah.bal)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
get_bal() called
10000
set_bal called
get_bal() called
20000
```