July - 4th

Python - tutorial 7

- **Encapsulation:** (to hide the implementation of actual method)
  input → output (when you want to hide method).

① composition:

```
class Tyre:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __str__(self):
        return("Tyres: \n\t
                a:" + self.a + "\n\t
                b:" + self.b)
class Engine:
    def __init__(self, c, d)
        self.c = c
        self.d = d
    def __str__(self):
        return("Engine: \n\t
                c:" + self.c + "\n\t)
                d:" + self.d
class Body:
    def __init__(self, size)
        self.size = size
    def __str__(self)
        return("Body: \n\t
                size:" + self.size
```

```
class Car:
    def __init__(self, tyre, engine, body):
        self.tyre = tyre
        self.body = body
        self.engine = engine
    def __str__(self):
        return str(self.tyre) + "\n"
               + str(self.engine) + "\n"
               + str(self.body)

t = Tyre('tyre1', 'tyre2')
e = Engine('engine1', 'engine2')
b = Body('size1')
c = Car(t, e, b)
print(c)
```

← (:: directly calling object → concept of encapsulation)

```
Tyres:
    a: tyre1
    b: tyre2

Engine:
    c: engine1
    d: engine2

Body:
    size: size1
```

[if all argument passed are object] composition

[if some of arguments are objects while other are variables or something] dynamic extension
String/int

~~Now, we perform same thing~~ by 'Encapsulation'

② Dynamic Extension:

```
class Dog:
    def __init__(self, name, dob, breed):
        self._name = name
        self._dob = dob
        self._breed = breed
    def __str__(self)
        return "%s is a %s born in %d."
            %(self._name, self._breed,
              self._dob)
kudrjavka = Dog("Kudrjavka", 1954, "Laika")
print(kudrjavka)

class Student:
    def __init__(self, anagraphic, student_id):
        self._anagraphic = anagraphic
        self._student_id = student_id
    def __str__(self):
        return str(self._anagraphic) +
            " student ID": %d" % self._student_id
alec_student = Student("MADHU", 1)
kudrjavka_Student = Student(kudrjavka, 2)
                            variable
                    (object of another class)
```

Hide implementation of one class & use it in another → encapsulation.

```
print(alec_student)
print(kudrjavka_student)
```

```
MADHV Student ID: 1
Kudrjavka is a laika born in 1954.
Student ID: 2
```

(*)

```
class C1:
    def __init__(self,a1,a2,a3):
        self.a1 = a1
        self.a2 = a2
        self.a3 = a3
    def m1(self):
        return (f'{self.a1} + " " + {self.a2} +
                " " + {self.a3}')
```

```
class C2:
```

same as C1 (use b instead of a in variables)

```
class C3:
```

" " " " " "

```
class C4:
```

" " " " " "

```
    def m4(self):
        return (f'{self.d1.m1()} + " " + {self.d2.m2()}
                + " " + {self.d3.m3()}')
```

```
a = C1(1,2,3)
b = C2(4,5,6)
c = C3(7,8,9)
d = C4(a,b,c)
d.m4()
'1'+" "+2+" "+3+" "+4+" "+5+" "+6+" "+7+" "
   +8+" "+9'
```

Abstraction + Inheritance

```
class C1:
    def __init__(self,a,b,c):
        self.a=a          (Public)
        self._b=b         (Protected)
        self.__c=c        (private)
class C2(C1):             Inherit
    pass
obj = C2(1,2,3)
print(obj.a)
print(obj._b)
print(obj._C1__c)
```

1
2
3

we can print PPP variables of C1 ~~outside~~ by using obj of C2 only when the are aware of their abstraction.

Abstraction + Encapsulation

```
class C1:
    def __init__(self,a,b,c):
        self.a=a
        self._b=b
        self.__c=c
    def __str__(self):
        return str(self.a)+
               str(self.b)+
str(self._C1__c)
        ~~str(self.c)~~
obj = C1(1,2,3)
print(obj)
```

456

```
class C2:
    def __init__(self,a):
        self.a=a
    def __str__(self):
        return str(self.a)
obj1 = C2(obj) | print(obj1) → 123
```

Public, Private and Protected exists as well as don't exists, we can understand from previous example. As by using inheritance & encapsulation we can access private/protected.

- Polymorphism And Duck Typing:
(ability to use same syntax for objects of different types)

```
def summer(a,b)
    return a+b
print(summer(1,1))
print(summer(["a","b","c"],["d","e"]))
print(summer("MA","DHU"))
```

operator is same but performs addition, list contraction, string

```
2
['a','b','c','d','e']
MADHU
```

- Files:

① Creating a txt-file

```
%%writefile text21.txt
Hello, this is how to create a txt-file.
```

② Opening a file

```
my_file = open('text21.txt')
my_file
something ...
my_file.read()
'Hello, this is how to create a txt-file'
```

But again if we write read command?

my-file.read() ⟶ ''

(∵ becoz whenever we try to read a file, it will be done from starting cursor-wise & once the reading is done, cursor will reach to end point & when we call read() again then now since cursor is at end, so blanck.)

[seek - try to place the cursor as required].

```
my-file.seek(0)   (∵ set cursor at starting)
↳ 0
my-file.read()    (∵ and now read again)
↳ 'Hello, this is how to create a txt-file.'
my-file.readlines()  (∵ list of lines in file)
↳ ['Hello, this is how to create a txt-file.']
```