OOP

- Concept
- class vs object
- OOP principle
- Initializing with __init__
- Inheritance: single, multiple, multilevel
- Polymorphism & Operator overloading
- Function overriding & Encapsulation

OOP

- If I had to write code to create FIFA game with function-based approach, I need to make use of complex data structure
- With OOP it's easier to model real world problem
- Class based approach is easier as I can create a blue-print, add method and attribute to it
- Analogy: Variable ➔ Attribute, Function ➔ Method

Class

- Blueprint of object
- Allows programmers to create user defined data type that have certain attributes and functions
- On taking analogy class is datatype and object is variable
- Property (data) of class is stored as attributes
- Methods are things object can do
- Example: Mario Villian, FIFA game Player, Student

  Attribute: Name, DOB, Country, Dribble rating

  Methods: Rest, Do Project, Learn, Dribble, Tackle

Object

- Instance of each Class
- Multiple objects can be generated using a single class
- Example: Each villain in Mario, specific play in FIFA, Specific Student of institute

OOP Principle

Inheritance

Polymorphism

Encapsulation

Abstraction

Defining Class

```
class People:
    species = 'mammals'          # class attribute

    def __init__(self, name, age, country='Nepal'): # constructor
        self.name = name         # self keyword
        self.age = age           # object attribute
        self.country = country   # with default argument

    def greet(self):
        print(f"Hello {self.name} from {self.country}")

p_1 = People('a', 30)
p_2 = People('b', 25, 'US')
p_1.greet()
p_2.greet()
```

Constructor

- Use to initialize the object
- Defined in __init__ function
- Automatically invoked on initialization

Inheritance

- Child class derive properties and method from parent
- Single (A → B), Multiple (A, B) → C, Multilevel (A → B → C)

```
class People:
    def __init__(self, name, dob, .....):
        self.name = name
        self.dob = datetime.strptime(dob, "%Y-%m-%d").date()

    @property
    def age(self):
        return (date.today() – self.dob).days // 365

    def is_adult(self):
        return self.age >= 18

class Player(People):   # Student can also inherit it but instead of sport there might be course
    def __init__(self, name, dob, ....., sport):
        People.__init__(self, name, dob, .....)   # self required on initializing with name
        self.sport = sport
```

Polymorphism

```
class Nepali:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"Mero nam {self.name} ho"

class American:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"My name in  {self.name}"

def talk(obj):
    print(obj.speak())

p_1 = Nepali('Hari')
p_2 = American('Harry')
talk(p_1)
talk(p_2)
```

Same function behaving differently – Duck Typing

Method Overwriting

```
class People:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"My name in  {self.name}"

class Nepali(People):
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"Mero nam {self.name} ho"


p_1 = Nepali('Hari')
p_2 = American('Harry')
print(p_1.speak())
print(p_2.speak())
```

Operator Overloading

```
class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}"

    def __gt__(self, other):
        return self.age > other.age

p_1 = People("Hari", 20)
p_2 = People("Harry", 20)

if p_1 > p_2:
    print(f"{p_1} is greater")
elif p_2 > p_1:
    print(f"{p_2} is greater")
else:
    print(f"{p_1} and {p_2} are equal")
```

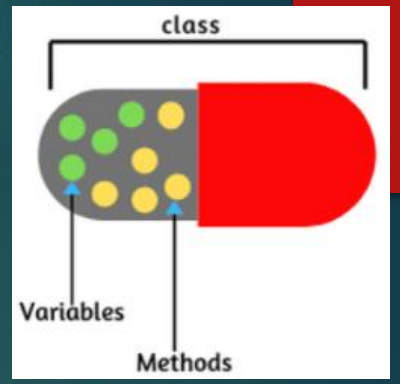
Operator Notation

Function/operator	Notation	Internal
Addition	+	__add__
Add and assign	+=	__iadd__
Subtraction	-	__sub__
Subtract and assign	-=	__isub__
Multiplication	*	__mul__
Power	**	__pow__
Division	/	__truediv__
Floor Division	//	__floordiv__
Remainder(modulo)	%	__mod__
Bitwise left shift	<<	__lshift__
Bitwise right shift	>>	__rshift__
Bitwise and	&	__and__
Bitwise or	|	__or__

Operator Notation

Bitwise xor	^	__xor__
Bitwise not	~	__invert__
Less than	<	__lt__
Less than equal	<=	__le__
Equal to	==	__eq__
Not equal to	!=	__ne__
Greater than	>	__gt__
Greater than equal	>=	__ge__
print	print()	__str__
length	len()	__len__
boolean	bool()	__bool__
Containment	in	__contains__

Encapsulation



- Bundle method & data operating within class.
- Attribute of class is made private using __
- No direct access from outer program
- Data is accessed via special function called getter & setter
- No friend class like C++

Encapsulation

```
class Book:
    def __init__(self, name, author):
        self.name = name
        self.__author = author

    def get_author(self):
        return self.__author

b_1 = Book('Rich Dad Poor Dad', 'Robert Kiyosaki')
print(b_1.name)       # Gives data
print(b_1.__author)   # Says not present
b_1 = 'Hacked'
print(b_1.name)
```

Name Mangling

- Technique to access the private data of the object
- obj._ClassName__field

```
class Book:
    def __init__(self, name, author):
        self.__name = name
        self.__author = author

b_1 = Book('Rich Dad Poor Dad', 'Robert Kiyosaki')
print(b_1._Book__name)
```

Abstraction

- Hiding complex implementation detail and showing only essential feature
- In string we don't know how .lower() works but we know it exist and convert data to lowercase

Abstract class

- Class with at least one abstract method
- Cannot be initiated.
- Child must implement abstract method on it
- No logic only definition on parent

```
from abc import ABC, abstractmethod

class Shape(ABC):       # Inherit ABC class
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

sqr = Square(4)      # Error as perimeter isn't implemented
area = Square.area()
print(area)
```

Exercise