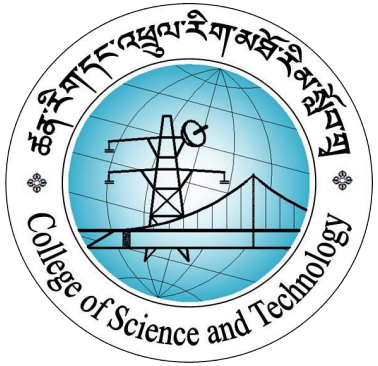


Royal University of Bhutan

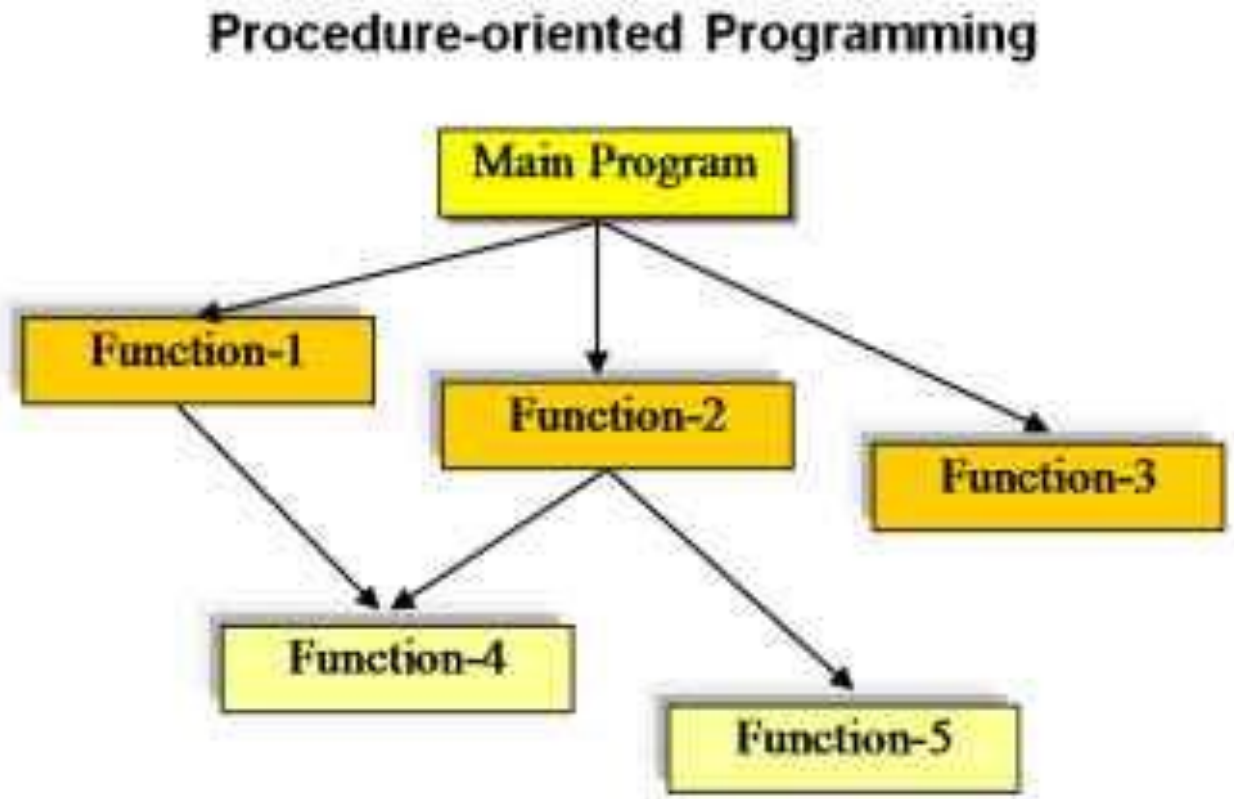
Unit V:
Principles of Object Oriented Programming

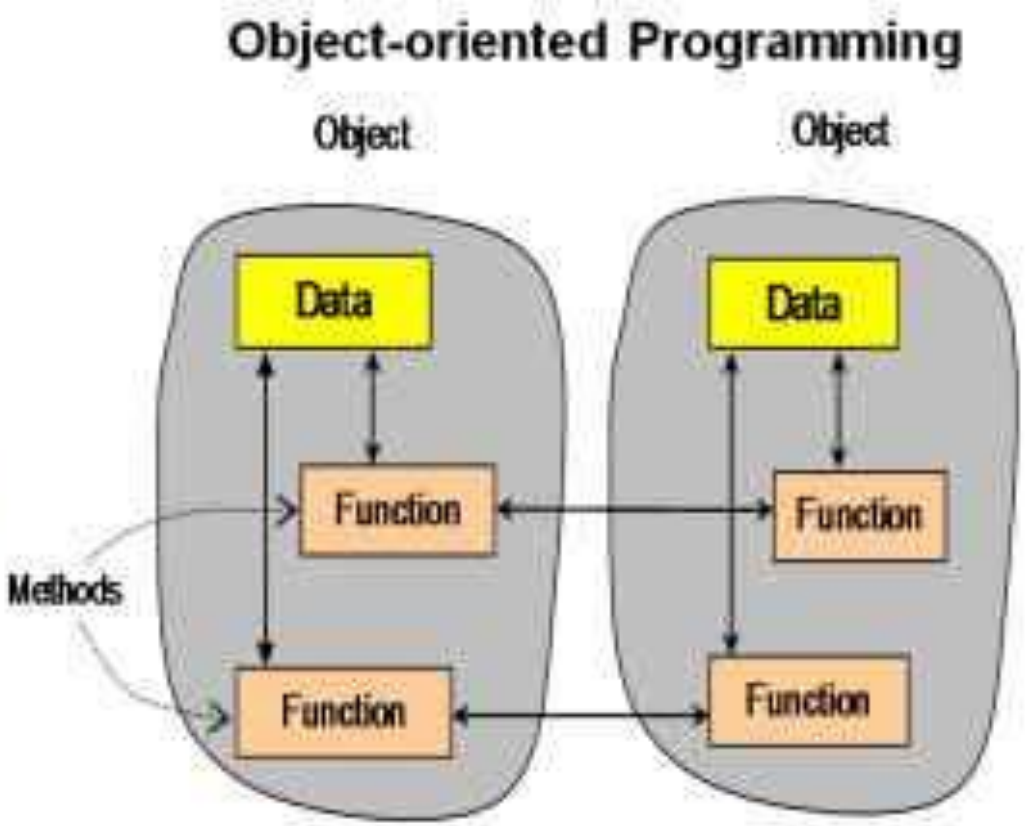
Programming Methodology (CSF101)

Outline

- Object, Class, Method
- Constructors and Destructors
- Abstraction, Encapsulation, Inheritance, Polymorphism
- Operator Overloading
- Type Conversions

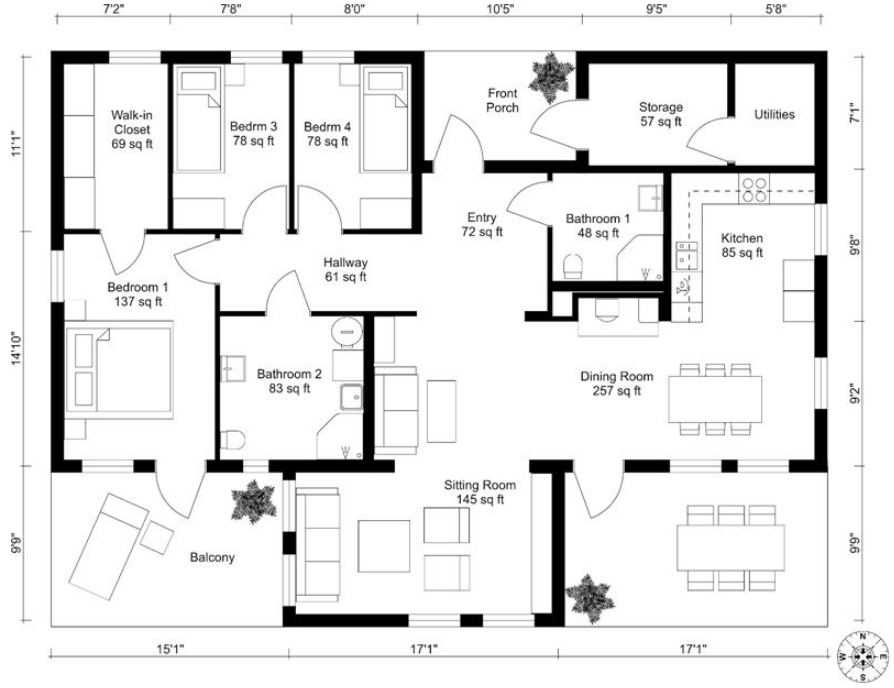
Why?

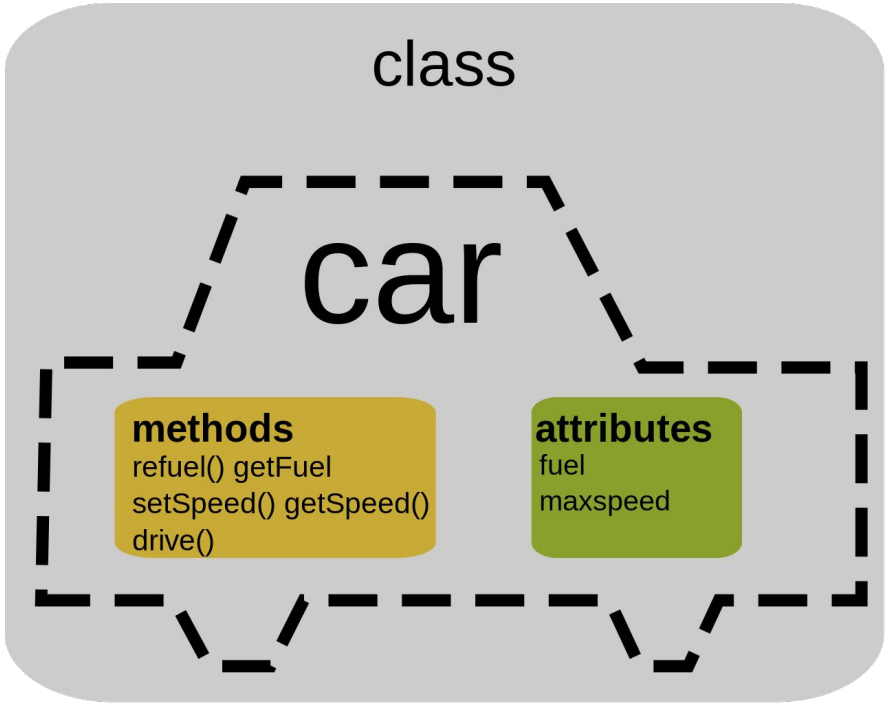

Procedure-oriented Programming
Main Program
Function-1
Function-2
Function-3
Function-4
Function-5


Object-oriented Programming
Object
Object
Data
Data
Function
Function
Methods
Function
Function

Class

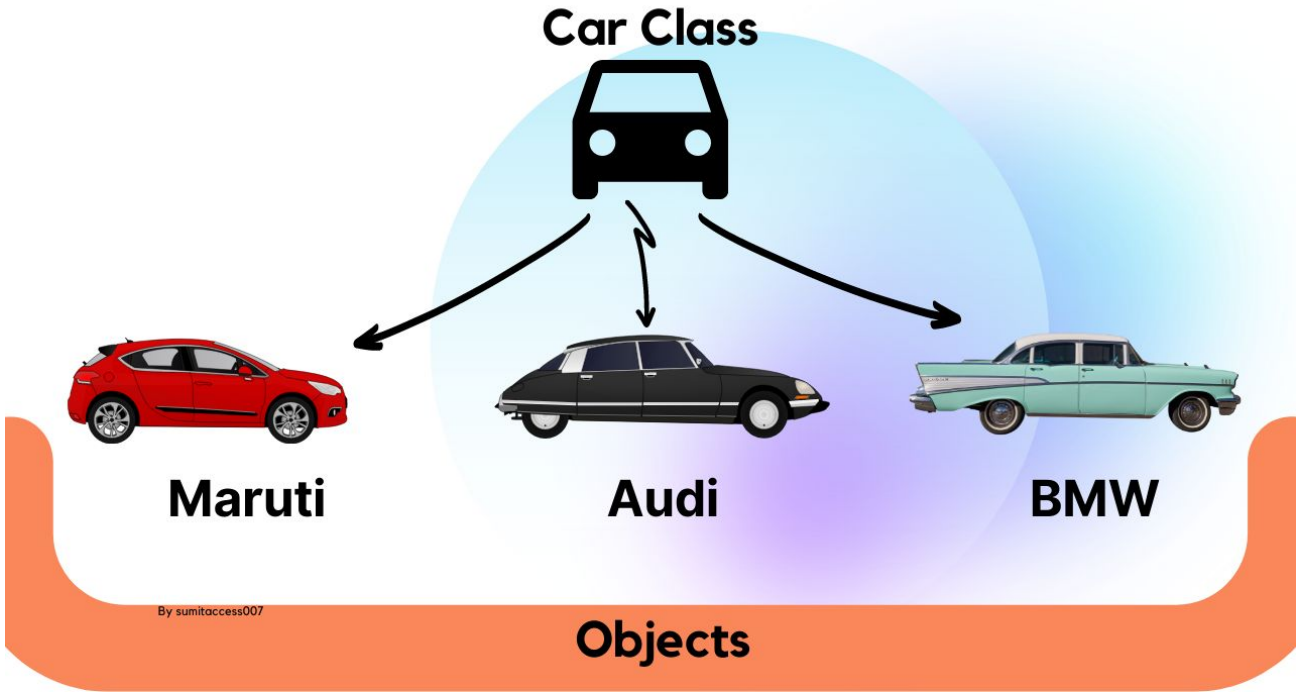
- a blueprint or **template** for **creating objects**. It defines the attributes and methods that all objects of that class will share.





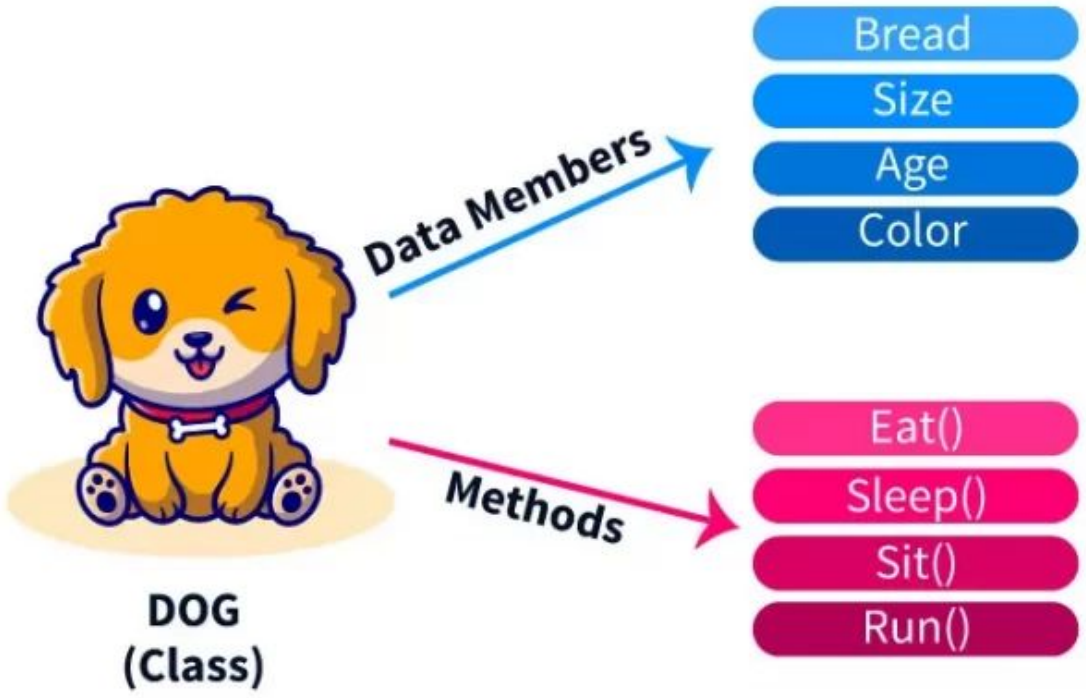
Object

- Created from class.
- self-contained entity that encapsulates **property** and related **operations** that act on that data.



Properties/Data Members & Behaviours/Methods

- Property: The facts/information about the object. AKA Attributes.
- Behaviour: Something that the object does



Class

- Example: *Student* class having attributes/properties *name* and *age*

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

Constructor: *__init__()*

Destructor: *__del__()*

Object

- Create Student objects by calling the Student() class:

```
Sonam = Student("Sonam Lhamo", 20)
Karma = Student("Karma Dorji", 23)
print(Sonam.name) #Sonam Lhamo
print(Karma.age)  #23
```

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

Sonam = Student("Sonam Lhamo", 20)
Karma = Student("Karma Dorji", 23)

print(Sonam.name)
print(Sonam.age)
print(Karma.age)
```

Exercise

- Create a class called *Dog*
- Give it 3 attributes of:
  - name
  - color
  - Age
- Make 3 dog objects from the Dog class. And name the variable that will hold the Dog objects as:
  - dawa
  - nima
  - blacky
- Print all the properties/attributes of all the dog objects

Method/Behaviour

- Classes can also contain methods/behaviors which are **functions defined inside the class.**

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

    def greeting(self):
        print("Hello, My name is ", self.name)

Sonam = Student("Sonam Lhamo", 20)
Sonam.greeting() #Hello, My name is Sonam Lhamo
```

Principles of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Inheritance

- Creating new classes from existing classes.
- New class → derived class (or child class)
- Existing class → base class (or parent class)

Example: Let's create a *HighSchoolStudent* class that inherits from *Student*

```python
class Student: #Base class
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greeting(self):
        print("Hello, My name is ", self.name)

class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
print(Sonam.grade)
```

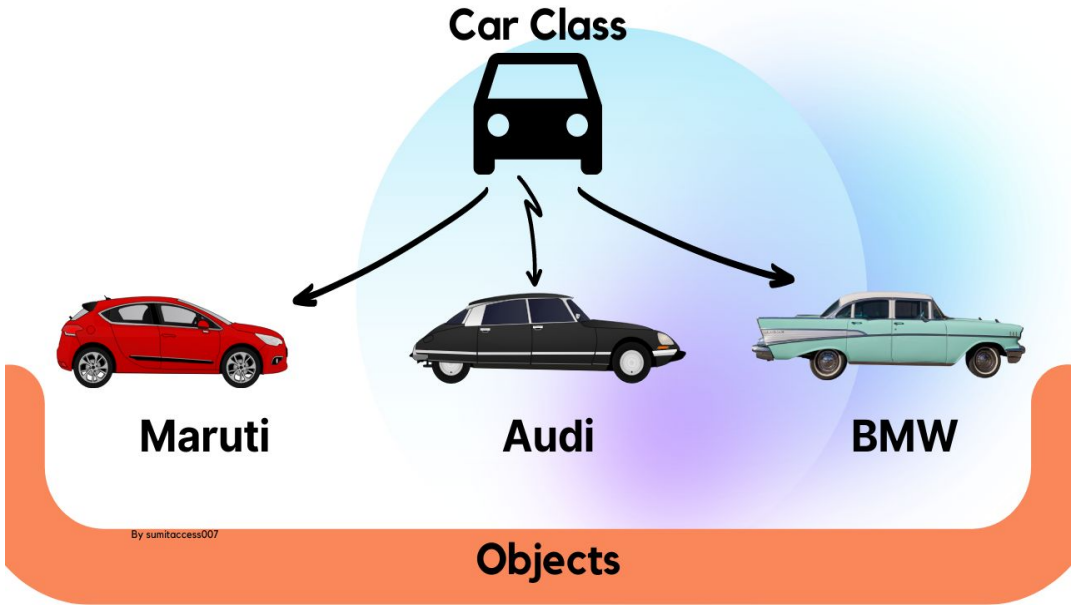
Inheritance

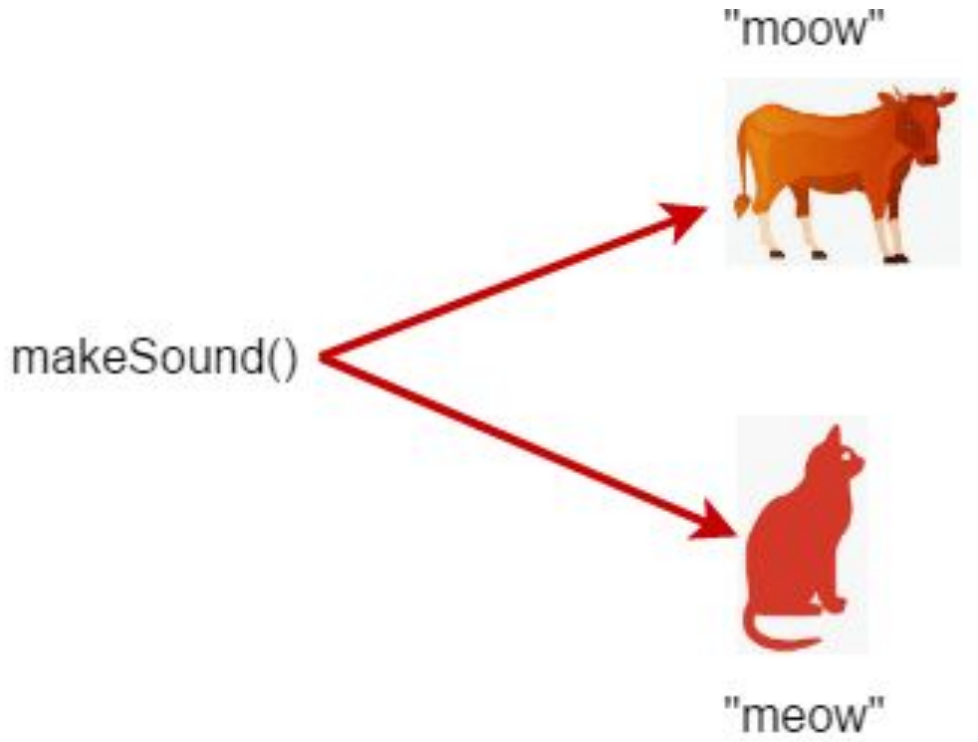
- We can also override methods in the derived class:

```
class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

    def greeting(self):
        print("Hey, I am ", self.name)

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
Sonam.greeting() #Hey, I am Sonam Lhamo
```

Polymorphism

- children can be of different forms
- Ability to use common operations in different forms for different data inputs.





- Example: we can have a common *study()* method that prints different outputs for different student objects.
- The *study()* method exhibits polymorphic behavior based on the object calling it.

```
class Student: #Base class
    def __init__(self, name, age):…

    def study(self):
        print( self.name, " is studying Python.")

class HighSchoolStudent(Student): #Derived class
    def __init__(self, name, age, grade):…

    def study(self):
        print(self.name, " is studying Mathematics.")

Karma = Student("Karma", 25)
Karma.study() #Karma is studying Python.

Sonam = HighSchoolStudent("Sonam Lhamo", 20, 12)
Sonam.study() #Sonam Lhamo is studying Mathematics.
```

Encapsulation

- bundling of attributes and methods inside a single class.
- Private attribute: restricted in access, it cannot be accessed or modified directly from outside the class.
- Denoted using double underscore prefixes.

For example:

```
class Student:
    def __init__(self):
        self.__id = 123 # private attribute

Sonam = Student()
print(Sonam.__id) # AttributeError
```

- To access or modify private attributes, we use setter and getter methods:

```
class Student:
    def __init__(self):
        self.__id = 123

    def get_id(self):
        return self.__id

    def set_id(self, id):
        self.__id = id

Sonam = Student()
print(Sonam.get_id())  # 123

Sonam.set_id(456)
print(Sonam.get_id())  # 456
```

Abstraction

- Hiding away the details of implementation
- It allows us to focus on what an object does rather than how it does it.

Operator Overloading

- giving extended meaning beyond their predefined operational meaning.

```
# + operator for different purposes.

print(1 + 2) #3

# concatenate two strings
print("Python"+"Python") #PythonPython

# Product two numbers
print(3 * 4) #12

# Repeat the String
print("Python"*4) #PythonPythonPythonPython
```

How to overload operators?

- Using a **special function or magic function.**
- Example:
  When we use + operator, the magic method **__add__** is automatically invoked

```
class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, other):
        return self.a + other.a

ob1 = A(1)
ob2 = A(2)
ob3 = A("Python")
ob4 = A("Code")

print(ob1 + ob2) #3
print(ob3 + ob4) #PythonCode
```

```
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Operator	Magic Method
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Operator	Magic Method
-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)

```python
class A:
    def __init__(self, a):
        self.a = a
    def __lt__(self, other):
        if(self.a<other.a):
            return "ob1 is lessthan ob2"
        else:
            return "ob2 is less than ob1"

    def __eq__(self, other):
        if(self.a == other.a):
            return "Both are equal"
        else:
            return "Not equal"

ob1 = A(2)
ob2 = A(3)
print(ob1 < ob2)

ob3 = A(4)
ob4 = A(4)
print(ob1 == ob2)
```

gy enriched with GNH values

Type Conversion

- converting data of one type to another.
- For example: converting *int* data to *str*.

1. Implicit Conversion - automatic type conversion
2. Explicit Conversion - manual type conversion

Implicit Conversion

- In certain situations, Python automatically converts one data type to another.

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))

#Value: 124.23
#Data Type: <class 'float'>
```

Explicit Conversion (Typecasting)

- users convert the data type of an object to required data type.

```
num_string = '10'
num_integer = 15

print("Data type of num_string before Type Casting:",type(num_string))
#Data type of num_string before Type Casting: <class 'str'>

# explicit type conversion
num_string = int(num_string)

print("Data type of num_string after Type Casting:",type(num_string))
#Data type of num_string after Type Casting: <class 'int'>

num_sum = num_integer + num_string

print("Sum:",num_sum) #25
print("Data type of num_sum:",type(num_sum)) #Data type of num_sum: <class 'int'>
```

Reference

Guttag, J. V. (2013). Introduction to computation and programming using Python. http://cds.cern.ch/record/2622221

Bader, D., Jablonski, J., & Heisler, F. (2021). Python Basics: A Practical Introduction to Python 3. Real Python (Realpython.Com).