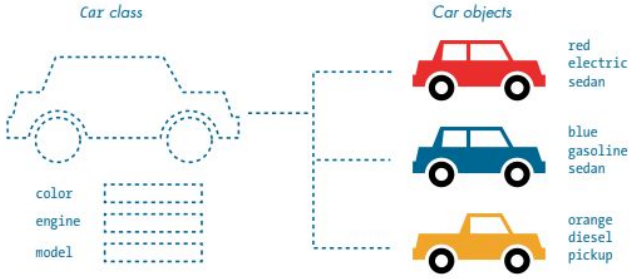
Object-Oriented Programming (OOP) in Python

Prepared By
Md. Tarek Hasan
Lecturer, Department of CSE
Email: tarek@cse.uiu.ac.bd

What is OOP?

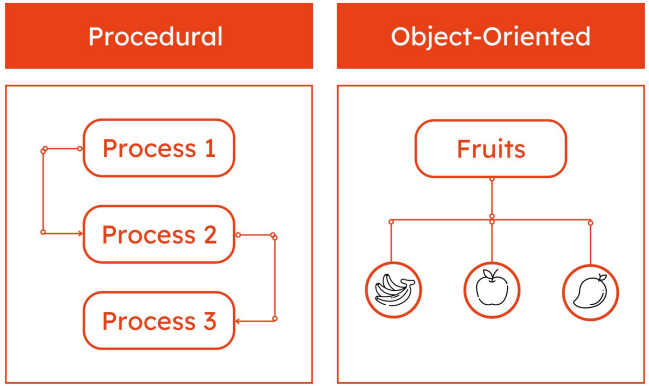
- OOP is a programming paradigm that organizes code into reusable objects.
- Objects contain data (attributes/variables) and behaviors (methods/functions).
- Four key principles of OOP:
  - **Encapsulation** – Bundling data and methods together
  - **Inheritance** – Reusing and extending existing code
  - **Polymorphism** – Different behaviors for the same interface
  - **Abstraction** – Hiding implementation details

- Example: OOP in real life: `A "Car" object with attributes (color, engine, model) and methods (drive(), brake()).`



Why Use OOP?

- Code reusability through **classes** and **objects**
- **Scalability** – Handle complex software easily
- **Better maintainability** – Modular and structured code
- **Encapsulation** improves data security
- Example: Comparing procedural vs. OOP approach

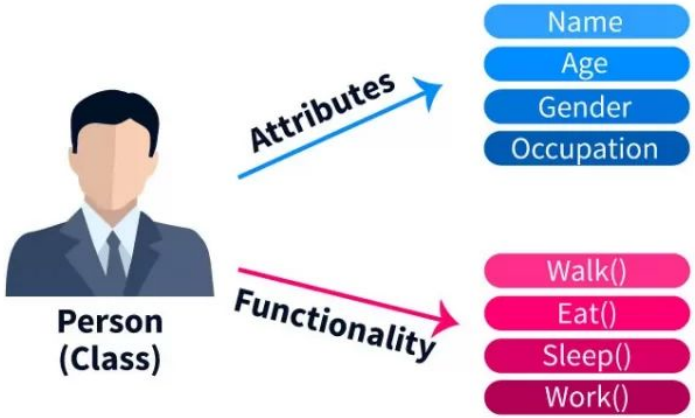


Understanding Classes and Objects

- **Class**: A blueprint/template for creating objects
- **Object**: An instance of a class with actual data

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

my_car = Car("Toyota", "Corolla")
print(my_car.brand, my_car.model)  # Output: Toyota Corolla
```



Creating a Class and an Object in Python

- Defining a class with attributes and methods
- Creating an instance (object) from the class
- Accessing attributes and calling methods

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

    def greet(self):
        return f“Hello, my name is {self.name} and I am {self.age} years old.”

john = Person(“John”, 30)
print(john.greet())
```

Understanding Instance Variables

- Instance variables are unique to each object.
- Class variables are shared across instances.

```
class Student:
    school = “XYZ School”  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

s1 = Student(“Alice”)
print(s1.name, s1.school)  # Output: Alice XYZ School
```

Understanding Scope and Namespaces

- Scope Levels in Python (LEGB Rule):
  - Local: Inside a function
  - Enclosing: Inside a nested function
  - Global: Defined at the script level
  - Built-in: Defined by Python itself

```
global_var = “I am Global”

def outer():
    enclosing_var = “I am Enclosing”

    def inner():
        local_var = “I am Local”
        print(local_var, enclosing_var, global_var)

    inner()

outer()
```

Built-in (Python's predefined scope)
└── Global (Defined at the module level)
└── Enclosing (Functions inside functions)
└── Local (Variables inside a function)

Methods in Classes

- Instance Methods – Operate on instance attributes
- Class Methods – Operate on class-level attributes
- Static Methods – Independent of class and instance

```python
class MathOperations:
    def instance_method(self, x):
        return x ** 2

    @classmethod
    def class_method(cls, x):
        return x ** 3

    @staticmethod
    def static_method(x):
        return x ** 4

obj = MathOperations()
print(obj.instance_method(2))  # Output: 4
print(MathOperations.class_method(2))  # Output: 8
print(MathOperations.static_method(2))  # Output: 16
```

Constructors and Destructors

- Constructor (__init__) – Initializes attributes
- Destructor (__del__) – Cleans up before object deletion

```
class Sample:
    def __init__(self):
        print("Constructor is called!")

    def __del__(self):
        print("Destructor is called!")

obj = Sample()
del obj
```

Object Created --> __init__() Called --> Object in Use --> __del__() Called --> Object Destroyed

Encapsulation in OOP

- Hiding data from direct access
- Using getter & setter methods

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def get_balance(self):
        return self.__balance

acc = BankAccount(1000)
print(acc.get_balance())  # Output: 1000
```

Access Control (Public, Protected, Private)

- Public Attributes (**`self.var`**) – Accessible anywhere
- Protected Attributes (**`self._var`**) – Suggests limited access
- Private Attributes (**`self.__var`**) – Strongly restricts access

```
class Example:
    def __init__(self):
        self.public = "I am Public"
        self._protected = "I am Protected"
        self.__private = "I am Private"

obj = Example()
print(obj.public)
print(obj._protected)
# print(obj.__private)  # Throws AttributeError
```

Name Mangling in Python

- Prevents accidental overriding

```
class Example:
    def __init__(self):
        self.__mangled = “Mangled Name”

obj = Example()
print(obj._Example__mangled)  # Output: Mangled Name
```


kulfyapp.com
Thank You.
CBS NEWS
2017 PRESIDENTIAL INAUGURATION
CBSN
LIVE

Thank you