جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم
قسم الانظمة الطبية الذكية

Lecture: (3)

Introduction to OOP and its need

Subject: Object oriented programming I

Class: Second

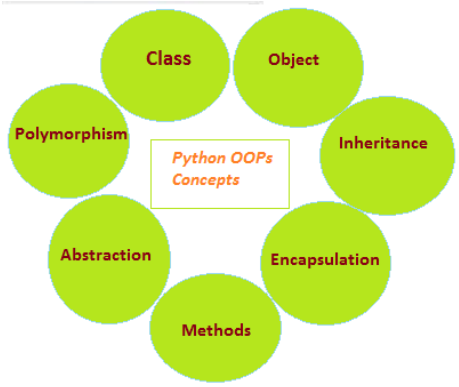
Lecturer: Dr. Maytham N. Meqdad

Introduction to OOP and its need

In Python object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class in Python
- Objects in Python
- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python



- **Class in Python**:

A class is a blueprint for creating objects. It defines properties (attributes) and methods (functions) that describe the behavior of the objects created from it.

- **Objects in Python**:

An object is an instance of a class. It is a specific realization of the class that contains data (attributes) and functions (methods) defined by the class.

- **Polymorphism in Python**:

Polymorphism means that objects of different classes can use the same method name, but each class implements the method differently based on its specific behavior.

- **Encapsulation in Python**:

Encapsulation is the concept of bundling data and methods that operate on the data within a class, while restricting direct access to some of the class's components. This is done by making variables private (using a single or double underscore) and providing getter and setter methods to access or modify them.

- **Inheritance in Python**:

Inheritance is the process by which one class (the child or subclass) can inherit properties and methods from another class (the parent or superclass), allowing code reuse and the extension of functionality.

- **Data Abstraction in Python**:

Data abstraction is the process of hiding complex implementation details and showing only the essential features of an object. This is often achieved using abstract classes and methods, where the specific implementation is left to the subclasses.

==========

Why Do We Need OOP?

1. **Code Reusability**: Through inheritance, OOP allows new classes to reuse existing code. This reduces duplication and makes code easier to maintain.
2. **Modularity**: Code is organized into objects (or classes), which promotes modularity. This makes it easier to understand, maintain, and modify specific parts of a program without affecting the whole system.
3. **Scalability**: OOP systems are easier to scale because objects can be independently developed and extended. As the system grows, new functionality can be added without disrupting the existing code.
4. **Improved Collaboration**: OOP allows multiple developers to work on different parts of a project simultaneously. Each developer can focus on a specific class or object, improving efficiency in team environments.
5. **Maintainability**: Encapsulation ensures that internal object details are hidden, reducing the likelihood of unintended interference from other parts of the code. This improves maintainability and reduces bugs.
6. **Flexibility through Polymorphism**: With polymorphism, you can write more generic and flexible code that can work with different types of objects, reducing code complexity and enhancing extensibility.
7. **Real-world Representation**: OOP allows developers to model real-world systems more naturally, making the design and implementation process intuitive and effective for complex software systems.

Examples

1. Program to Model a Patient's Health Record

This program models a patient's health record, storing basic medical information and allowing updates. The program models a patient's medical record. The class Patient stores basic information (name, age, blood type) and keeps a list of medical records. You can add new records to a patient's history and view all the details.

```
# Define a Patient class
class Patient:
    def __init__(self, name, age, blood_type, medical_history=[]):
        self.name = name  # Patient's name
        self.age = age  # Patient's age
        self.blood_type = blood_type  # Blood type (A, B, O, etc.)
        self.medical_history = medical_history  # List of medical history records

    # Method to add a medical record
    def add_medical_record(self, record):
        self.medical_history.append(record)
        print(f"Medical record added: {record}")

    # Method to display the patient's details
    def display_patient_info(self):
        print(f"Patient Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Blood Type: {self.blood_type}")
        print(f"Medical History: {self.medical_history}")

# Create a patient object and update the medical history
patient1 = Patient("John Doe", 45, "A+")
patient1.add_medical_record("Diabetes diagnosed in 2018")
patient1.add_medical_record("Hypertension diagnosed in 2021")
patient1.display_patient_info()
```

https://www.programiz.com/online-compiler/48ttOpREpDboa

2. Program to Monitor Blood Pressure Readings

This program monitors and stores blood pressure readings for a patient, alerting if the readings are too high. This program models a blood pressure monitor. The class BloodPressureMonitor allows adding systolic and diastolic readings. It checks if a reading is above normal (hypertension) and gives a warning if the pressure is too high

```
# Define a BloodPressureMonitor class
class BloodPressureMonitor:
    def __init__(self, patient_name):
        self.patient_name = patient_name
        self.readings = []  # List to store blood pressure readings

    # Method to add a blood pressure reading
    def add_reading(self, systolic, diastolic):
        reading = {"systolic": systolic, "diastolic": diastolic}
        self.readings.append(reading)
        print(f"Reading added: {systolic}/{diastolic} mmHg")

        # Check if the blood pressure is too high
        if systolic > 140 or diastolic > 90:
            print("Warning: High blood pressure detected!")

    # Method to display all readings
    def display_readings(self):
        print(f"Blood Pressure Readings for {self.patient_name}:")
        for reading in self.readings:
            print(f"{reading['systolic']}/{reading['diastolic']} mmHg")

# Create a blood pressure monitor object and add readings
monitor = BloodPressureMonitor("Alice")
monitor.add_reading(120, 80)  # Normal reading
monitor.add_reading(145, 95)  # High blood pressure
monitor.display_readings()
```

https://www.programiz.com/online-compiler/9gffYUBoGuUYi

3. Program to Track Medication Schedule

This program models a medication tracking system, allowing patients to track and manage their medication schedule.

- This program tracks a patient's medication schedule.
- The `Medication` class holds information about a single medication, including the name, dose, and frequency.
- The `PatientMedicationSchedule` class allows adding medications to a patient's schedule and viewing the complete list.

```
# Define a Medication class
class Medication:
    def __init__(self, name, dose, frequency):
        self.name = name  # Medication name
        self.dose = dose  # Dosage (e.g., mg)
        self.frequency = frequency  # How often to take (e.g., per day)

    # Method to display medication details
    def display_medication(self):
        print(f"Medication: {self.name}, Dose: {self.dose}, Frequency: {self.frequency} times per day")

# Define a PatientMedicationSchedule class
class PatientMedicationSchedule:
    def __init__(self, patient_name):
        self.patient_name = patient_name
        self.medications = []  # List of medications

    # Method to add a medication
    def add_medication(self, medication):
        self.medications.append(medication)
        print(f"Added medication: {medication.name}")

    # Method to display the patient's medication schedule
    def display_schedule(self):
        print(f"Medication Schedule for {self.patient_name}:")
        for med in self.medications:
            med.display_medication()

# Create medication objects and a schedule for a patient
med1 = Medication("Metformin", "500 mg", 2)
med2 = Medication("Lisinopril", "10 mg", 1)

schedule = PatientMedicationSchedule("John Doe")
schedule.add_medication(med1)
schedule.add_medication(med2)
schedule.display_schedule()
```

https://www.programiz.com/online-compiler/82EEgn2X5fMBr

4. Program to Track Patient Vitals

This program models a system to track a patient's vital signs, such as temperature, pulse, and respiratory rate.

- This program tracks a patient's vital signs.
- The class `PatientVitals` stores temperature, pulse, and respiratory rate for a patient.
- You can add new vital sign records and view all recorded vitals for monitoring the patient's condition.

```
# Define a PatientVitals class
class PatientVitals:
    def __init__(self, patient_name):
        self.patient_name = patient_name
        self.vitals = []  # List to store vital sign records

    # Method to add vitals
    def add_vitals(self, temperature, pulse, respiratory_rate):
        vital_signs = {
            "temperature": temperature,
            "pulse": pulse,
            "respiratory_rate": respiratory_rate
        }
        self.vitals.append(vital_signs)
        print(f"Vitals recorded: Temp={temperature}°C, Pulse={pulse} bpm, Resp={respiratory_rate} breaths/min")

    # Method to display all recorded vitals
    def display_vitals(self):
        print(f"Vitals for {self.patient_name}:")
        for vital in self.vitals:
            print(f"Temperature: {vital['temperature']}°C, Pulse: {vital['pulse']} bpm, Respiratory Rate:
{vital['respiratory_rate']} breaths/min")

# Create a vitals tracking object and add readings
vitals_tracker = PatientVitals("Emily")
vitals_tracker.add_vitals(36.8, 72, 16)  # Normal vitals
vitals_tracker.add_vitals(38.5, 90, 20)  # Fever and elevated vitals
vitals_tracker.display_vitals()
```

https://www.programiz.com/online-compiler/3lOO4ixlfX3Sj