PYTHON TUTORIAL FOR BEGINNERS

Source: www.youtube.com/@RishabhMishraOfficial

Chapter - 20

OOPs in Python

- What is OOPs
- Why OOP is required
- Class and Object
- Attributes and Methods
- __init__ Method (Constructor)
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OOPs in Python

Two ways of programming in Python:

1) **Procedural** Programming,

2) **OOPs**

OOPs: Object Oriented Programming

A way of organizing code by creating "blueprints" (called **classes**) to represent real-world things like a student, car, or house. These blueprints help you create **objects** (individual examples of those things) and define their behavior.

Class: A class is a blueprint or template for creating objects.

It defines the properties (**attributes**) & actions/behaviors (**methods**) that objects of this type will have.

Object: An object is a specific instance of a class.

It has actual data based on the blueprint defined by the class.

OOPs Example in Python

Example: **Constructing a building**

Class: Blueprint for a floor.

Object: Actual house built from the blueprint. Each house (object) can have different features, like paint color or size, but follows the same blueprint.

Why OOPs?

- **Models Real-World Problems:**
  Mimics real-world entities for easier understanding.
- **Code Reusability:**
  Encourages reusable, modular, and organized code.
- **Easier Maintenance:**
  OOP organizes code into small, manageable parts (classes and objects). Changes in one part don't impact others, making it easier to maintain.
- **Encapsulation:**
  Encapsulation **protects data** integrity and **privacy** by bundling data and methods within objects.
- **Flexibility & Scalability:**
  OOP makes it easier to add new features without affecting existing code.

OOPs – Question

Write a Python program to:

1. Define a Student class with attributes name, grade, percentage, and team.
   - Include an __init__ method to initialize these attributes.
   - Add a method student_details that prints the student's details in the format: "<name> is in <grade> grade with <percentage>%, from team <team>".
2. Create two teams (team1 and team2) as string variables.
3. Create at least two student objects, each belonging to one of the teams.
4. Call the student_details method for each student to display their details.

Class and Object Example

Class: A class is a blueprint or template for creating objects.

Object: An object is a specific instance of a class.

Example

```
class Student:
    pass
# Create an object
student1 = Student()
print(type(student1))
# Output: <class '__main__.Student'>
```

Attributes and Methods

Attributes: Variables that hold data about the object.

Methods: Functions defined inside a class that describe its behavior.

Example

```
class Student:
    def __init__(self, name, grade):
        self.name = name  # Attribute
        self.grade = grade  # Attribute
    def get_grade(self):  # Method
        return f"{self.name} is in grade {self.grade}."
```

```
# Object creation
student1 = Student("Madhav", 10)
print(student1.get_grade())  # Output: Madhav is in grade 10.
```

The __init__ Method (Constructor)

Whenever we create/construct an object of a class, there is an inbuilt method **__init__** which is automatically called to **initialize** attributes.

The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

Example

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
# Initialize object with attributes
student1 = Student("Madhav", 10)
print(student1.name)  # Output: Madhav
```

Abstraction in Python: Hiding unnecessary details

Abstraction hides implementation details and shows only the relevant functionality to the user.

Example

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

    def is_honors(self):  # Abstracting the logic
        return self.percentage > 90  # Logic hidden
# Abstract method in use
student1 = Student("Madhav", 10, 98)
print(student1.is_honors())  # Output: True
```

Encapsulation in Python: Restricting direct access to attributes & methods

Encapsulation restricts access to certain attributes or methods to protect the data and enforce controlled access.

Example

```
class Student:
    def __init__(self, name, grade, percentage):
        self.name = name
        self.grade = grade
        self.__percentage = percentage  # Private attribute
(hidden)
    def get_percentage(self):  # Public method to access the
private attribute
        return self.__percentage

# Creating a student object
student1 = Student("Madhav", 10, 98)

# Accessing the private attribute using the public method
print(f"{student1.name}'s percentage is
{student1.get_percentage()}%.")
print(student1.__percentage) # error
```

Inheritance in Python: Reusing Parent's prop & methods

Inheritance (parent-child), allows one class (child) to reuse the properties and methods of another class (parent). This avoids duplication and helps in code reuse.

Example

```
class Student:
    def __init__(self, name, grade, percentage):
        self.name = name
        self.grade = grade
        self.percentage = percentage
    def student_details(self):  # method
        print(f'{self.name} is in {self.grade} grade with
{self.percentage}%')
```

```
class GraduateStudent(Student):  # GraduateStudent inherits from Student

    def __init__(self, name, grade, percentage, stream):
        super().__init__(name, grade, percentage)  # Call parent class initializer
        self.stream = stream  # New attribute specific to GraduateStudent

    def student_details(self):
        super().student_details()
        print(f"Stream: {self.stream}")

# Create a graduate student
grad_student = GraduateStudent("Vishakha", 12, 94, "PCM")
    # Vishakha is in 12 grade with 94%
grad_student.student_details()     # Stream: PCM
```

Polymorphism in Python: Same method but different output

Polymorphism allows methods in different classes to have the same name but behave differently depending on the object.

Example

```
class GraduateStudent(Student):
    def student_details(self):  # Same method as in parent class
        print(f"{self.name} is a graduate student from final year.")

# Polymorphism in action
student1 = Student("Madhav", 10, 98)
grad_student = GraduateStudent("Sudevi", 12, 99, "PCM")
student1.student_details()
# Output: Madhav is in 10 grade with 98%
```

```
grad_student.student_details()
# Output: Sudevi is a graduate student from final year.
```

Python Tutorial Playlist: **Click Here**

https://www.youtube.com/playlist?list=PLdOKnrf8EcP384Ilxra4UlK9BDJGwawg9