Python

SARAVANA

Object-Oriented Programming (OOP)

Introduction to OOP

- **Definition**: Object-Oriented Programming is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes) and code in the form of procedures (methods).
- **Purpose**: To increase the modularity, reusability, and maintainability of code.

OOPS Concepts

Class

Objects

Polymorphism

Encapsulation

Inheritance

Parent Class & Child Class

Data Abstraction

Classes and Objects

Class

A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects (instances) will have.

Class: A blueprint for creating objects with a set of methods and attributes.

Objects

An object is an instance of a class. It is created using the class name followed by parentheses.

Object: An instance of a class, containing data and methods defined in the class.

Function & Methods

Function - A function is a block of code designed to perform a specific task. Functions can be defined and called independently of classes.

Functions can take arguments, perform computations,

and return results.

```
def my_function():
    pass
```

Methods - A method is a function that is associated with an object and defined within a class. Methods operate on data contained within the object (i.e., the instance of the class) and can access and modify the object's attributes.

Methods are called on objects and can use or modify

the object's internal state.

```
class Test:
    def my_method(self):
        pass
```

Attributes

Attributes are variables that belong to an object or class. They store data related to the object or class and define its state or properties.

Types:

Instance Attributes: Attributes specific to an instance of a class. Each object can have different values for these attributes.

Class Attributes: Attributes shared among all instances of a class. They have the same value for every object created from the class.

Key Difference

Scope:

Instance Attributes: Unique to each instance. Different instances can have different values.

Class Attributes: Shared among all instances. The same value is accessed by every instance.

Access:

Instance Attributes: Accessed using self.attribute_name inside or outside class methods.

Class Attributes: Accessed using ClassName.attribute_name or self.attribute_name inside class methods.

Modification:

Instance Attributes: Modified independently for each instance.

Class Attributes: Modifying a class attribute affects all instances that haven’t overridden it with an instance attribute of the same name.

Decorators

A class method is a method that is bound to the class and not the instance of the class. It can access and modify class state that applies across all instances of the class. They take `cls` as their first parameter, which refers to the class itself, rather than `self`, wh `@classmethod` istance of the class.

A static method is a method that does not operate on an instance of the class or the class itself. It behaves like a regular function but belongs to the class's namespace. They don't have access to instance-specific data (i.e., they don't take `self` as their first argument) or class-specific data

`@staticmethod`

Property methods allow you to define methods that can be accessed like attributes. They provide a way to customize the behavior of attribute access and modification

`@property`

Private methods are methods intended to be accessed only within the class. They are not part of the public API of the class and should not be accessed from outside.

Decorators

Class Methods: Use the @classmethod decorator. Operate on the class itself and can modify class state. Accessed via cls.

Static Methods: Use the @staticmethod decorator. Do not operate on class or instance state. Used for utility functions. Accessed directly on the class or instance.

Property Methods: Use @property and @<property_name>.setter decorators. Allow methods to be accessed like attributes, useful for encapsulation.

Private Methods: Use double underscores (__) for methods intended for internal use only. Name mangling is applied to make them less accessible from outside the class.

Constructor

A constructor is a special method used to initialize a newly created object. In Python, the constructor is defined using the `__init__` method.

`__init__` is the constructor method that initializes the attributes of the class when a new instance is created.

```
class MyClass:
  def __init__(self):
    # This is the constructor method
    self.attribute = "Initial Value"
```

Inheritance

Inheritance allows a class (the child or derived class) to inherit attributes and methods from another class (the parent or base class).

It promotes code reuse and establishes a natural hierarchy.

Types:

Single Inheritance: A class inherits from one parent class.

Multiple Inheritance: A class inherits from more than one parent class.

Multilevel Inheritance: A class inherits from a derived class.

Hierarchical Inheritance: Multiple classes inherit from a single parent class.

Parent Class & Child Class

It allows us to create a new class from an existing one. known as the superclass (parent or base class).

Parent Class (Base Class)

A parent class (or base class) is a class that provides attributes and methods that are shared by other classes. It serves as a blueprint for other classes.

Child Class (Derived Class)

A child class (or derived class) is a class that inherits from a parent class. It can use the attributes and methods of the parent class and also define additional attributes and methods or override existing ones.

Polymorphism

Polymorphism allows for methods or operators to operate in different ways depending on the object that invokes them.

Run-Time Polymorphism is achieved through method overriding and is determined at runtime.

Compile-Time Polymorphism is achieved through method overloading and operator overloading and is determined at compile time.

Encapsulation

Encapsulation is the concept of wrapping data (attributes) and methods into a single unit (a class) and restricting access to some of the object's components. This is typically achieved through private and public attributes and methods.

Summary

Classes: Blueprints for creating objects, defining their structure and behavior.

Objects: Instances of classes that hold data and can use the methods defined in the class.

Methods: Functions defined within a class that describe the behaviors of an object.

Attributes: Variables defined within a class that store the state of an object.

Constructors: Special methods (e.g., `__init__` in Python) that initialize new instances of a class, setting up initial state.

Inheritance: Mechanism by which one class (subclass) inherits attributes and methods from another class (superclass), enabling code reuse and establishing hierarchical relationships.

Method Overriding: When a subclass provides a specific implementation for a method already defined in its superclass, allowing subclass-specific behavior.

Polymorphism: Ability for different classes to be treated as instances of a common superclass through a common interface, supporting flexible and dynamic behavior.

Encapsulation: Bundling of data (attributes) and methods that operate on the data into a single unit (class), and restricting access to some of the object's components.

Multithreading and Multiprocessing

Understanding Concurrent and Parallel Execution

Multithreading: Concurrent execution of threads within the same process. Best for I/O-bound tasks.

Multiprocessing: Parallel execution of processes, each with its own Python interpreter and memory space. Best for CPU-bound tasks.

Module

A **module** is a single file with a `.py` extension that contains Python definitions and statements. Modules are used to organize code into manageable sections and to facilitate code reuse.

Key Characteristics:

Code Organization: Modules allow you to group related functions, classes, and variables into a single file.

Importing: You can import a module into another script or module using the `import` statement.

Namespace: Each module has its own namespace, meaning the variables and functions defined in a module do not conflict with those in other modules.

Modules

Creating, Importing & Running the Module

The `dir()` function returns a list of attributes and methods of an object, including modules.

Python looks for modules in specific locations which are listed in `sys.path`

Module Namespace - Each module in Python has its own namespace. This means that variables and functions defined in one module are not directly visible to other modules unless explicitly imported. This helps in avoiding naming conflicts.

Packages

A package is a directory that contains multiple modules and an `__init__.py` file. The `__init__.py` file is used to initialize the package and can include initialization code or be empty.

Packages help in organizing code into manageable sections, promoting code reuse, and avoiding naming conflicts by creating separate namespaces.

Packages

Defining a package

Importing a package

Installing third party package

A module is a single file containing Python code. It can define functions, classes, and variables, and it can also include runnable code.

A package is a collection of modules organized in a directory hierarchy.

Introduction to Database Operations

Introduction to MySQL

MySQL: An open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) for database access. It is widely used for web applications and offers a robust, scalable solution for managing data.

Introduction to Oracle

Oracle Database: A powerful and widely-used RDBMS developed by Oracle Corporation. It supports complex queries, transactions, and high levels of concurrency and scalability.

Thanks