**Week 4 python**

**What is Inheritance?**

Inheritance allows a class to inherit attributes and methods from another class, creating a parent-child relationship.

**Types of Inheritance**

Python supports single, multiple, and multilevel inheritance, allowing for code reuse and specialization.

**Overriding Methods**

Child classes can override inherited methods to modify their behavior and customize functionality.

PYTHON INHERITANCE

Python inheritance and its various types, syntax, working with attributes and methods, and provide real-life examples. Discover how inheritance can improve code reusability and organization and gain a deeper understanding of this fundamental feature in Python programming.

What is Inheritance?

Inheritance is a powerful concept in Python that allows you to create a new class (child class) based on an existing class (parent class). It enables code reuse, reduces redundancy, and promotes the hierarchical organization of code. Through inheritance, child classes inherit attributes and methods from its parent, thereby inheriting their behavior and functionality.

Types of Inheritance in Python

Single Inheritance

A child class inherits from a single parent class. It follows a simple one-to-one relationship hierarchy, where the child class extends the capabilities of the parent class.

Multiple Inheritance

A child class inherits from multiple parent classes. This allows the child class to inherit attributes and methods from multiple sources, resulting in a versatile, yet complex, inheritance structure.

Multilevel Inheritance

A child class inherits from another child class, which further inherits from a parent class. This creates a multilevel hierarchy where each child class extends the functionality of its parent class.

Hierarchical Inheritance

Multiple child classes inherit from a single parent class. This structure promotes specialization, as different child classes can focus on specific attributes and methods inherited from the parent.

Syntax of Inheritance in Python

To define a parent class in Python, simply create a class with the desired attributes and methods. To define a child class that inherits from the parent, add the parent class as a parameter in the child class definition. For example:

class ParentClass:

# Parent class attributes and methods go here

...

class ChildClass(ParentClass):

# Child class attributes and methods go here

...

Working with Inherited Attributes and Methods

Accessing Inherited Attributes

Child classes can access inherited attributes directly using the same syntax as accessing their own attributes. This allows child classes to utilize the attributes defined in the parent class.

Overriding Inherited Methods

Child classes can override inherited methods by redefining the method with the same name and modifying its behavior. This provides flexibility to customize functionality based on specific needs.

Adding New Attributes and Methods

Child classes can add new attributes and methods that are unique to them, enhancing the functionality inherited from the parent class. This promotes extensibility and customization.

Example of Inheritance in Python

Let's create a real-life example to demonstrate inheritance in action. We'll define a parent class "Animal" with common attributes and methods and create child classes "Dog" and "Cat" with their specific attributes and overridden methods. This example will showcase the power of inheritance in organizing and reusing code.

What are Iterators?

Iterators are objects that allow sequential access to elements of a collection.

They provide an interface for fetching elements one by one.

Python built-in objects such as lists and strings are iterable, meaning they can be used with iterators.

Creating Custom Iterators

The \_\_iter\_\_ Method

The \_\_iter\_\_ method is used to create an iterator object and returns the iterator itself.

The \_\_next\_\_ Method

The \_\_next\_\_ method is responsible for returning the next element in the iteration.

Examples of Iterators

Iterating over a list of names, characters in a string, or numbers in a range are some common examples of using iterators in Python.

What are Generators?

Generators are a type of iterator that can be defined with simple functions.

They use the yield keyword to pause execution and return a value, without terminating the function.

Generators are memory-efficient and can handle large data sets since they generate values on the fly.

Creating Custom Generators

The yield Keyword

The yield keyword is used to yield a value from the generator and pause the execution of the function.

Examples of Generators

Generating Fibonacci sequence, parsing large log files, and reading data from a network stream are some examples of using generators in Python.

Benefits of Generators

Generators provide a clean and efficient way to handle large datasets, simplify code, and conserve memory.

Conclusion and Summary

Iterators and generators are powerful tools in Python that enable efficient and flexible iteration. They enhance code readability, promote reusability, and optimize memory usage.

Types of Inheritance in Python

Single Inheritance

A child class inherits from a single parent class. It follows a simple one-to-one relationship hierarchy, where the child class extends the capabilities of the parent class.

Multiple Inheritance

A child class inherits from multiple parent classes. This allows the child class to inherit attributes and methods from multiple sources, resulting in a versatile, yet complex, inheritance structure.

Multilevel Inheritance

A child class inherits from another child class, which further inherits from a parent class. This creates a multilevel hierarchy where each child class extends the functionality of its parent class.

Hierarchical Inheritance

Multiple child classes inherit from a single parent class. This structure promotes specialization, as different child classes can focus on specific attributes and methods inherited from the parent.

Syntax of Inheritance in Python

To define a parent class in Python, simply create a class with the desired attributes and methods. To define a child class that inherits from the parent, add the parent class as a parameter in the child class definition. For example:

class ParentClass:

# Parent class attributes and methods go here

...

class ChildClass(ParentClass):

# Child class attributes and methods go here

...

Working with Inherited Attributes and Methods

Accessing Inherited Attributes

Child classes can access inherited attributes directly using the same syntax as accessing their own attributes. This allows child classes to utilize the attributes defined in the parent class.

Overriding Inherited Methods

Child classes can override inherited methods by redefining the method with the same name and modifying its behavior. This provides flexibility to customize functionality based on specific needs.

Adding New Attributes and Methods

Child classes can add new attributes and methods that are unique to them, enhancing the functionality inherited from the parent class. This promotes extensibility and customization.

Example of Inheritance in Python

Let's create a real-life example to demonstrate inheritance in action. We'll define a parent class "Animal" with common attributes and methods, and create child classes "Dog" and "Cat" with their specific attributes and overridden methods. This example will showcase the power of inheritance in organizing and reusing code.

Python inheritance is a fundamental concept that enhances code reusability, promotes organization, and encourages modular design patterns. By utilizing inheritance, you can create hierarchical relationships between classes, preserve code structure, and streamline development.

**Datetime**

* From datetime import datetime

Current\_time = datetime.now()

Where date represents dates(year, month, day)

Time represents (hours, minutes, second )

* The **datetime** module in Python is part of the standard library and provides classes for working with dates and times. It offers a set of powerful and flexible tools to handle various aspects of date and time representation. Here are some key components and functionalities of the **datetime** module:
* **datetime Class:**
* The **datetime** class is the core component, representing a combination of date and time.
* It has attributes such as year, month, day, hour, minute, second, and microsecond.
* Instances of the **datetime** class can be created to represent specific points in time.

**Output formatting**

Output formatting refers to the method of presenting a program's outcomes in a well-structured and easily readable manner, all without delving into the underlying code. Its primary objective is to enhance clarity and readability. This involves proper utilization of indentation, spacing, and alignment to facilitate better understanding.

Structured Presentation: The organization of output in a structured manner is crucial, especially for intricate data. This may encompass grouping related information, incorporating headers or labels, and establishing a logical flow.

Consistent Formatting: Maintaining consistency in formatting significantly enhances the overall appearance of the output. Employing a uniform style for information presentation assists users in comprehending and interpreting the data more effectively.

Numeric Formatting: Numeric values, such as currency or percentages, need to be formatted correctly. This involves specifying the number of decimal places, utilizing commas for large numbers, and ensuring uniformity in numeric representation.

Logical Grouping: Logical grouping of related information is essential. In tabular data, for example, columns should be aligned, and headers must clearly convey the content of each column. This logical arrangement aids in swift comprehension.

Avoiding Redundancy: Efforts should be made to eliminate unnecessary repetition of information. The output should deliver essential details without inundating the user with redundant data.

Certainly! Let's break down each sentence:

1. Introduction to Python Classes:

• Python classes are a fundamental part of object-oriented programming.

• Explanation: Object-oriented programming (OOP) is a programming paradigm that uses classes and objects. Python classes are a key feature of OOP, providing a way to structure and organize code.

• Classes provide a blueprint for creating objects, allowing for code organization and reuse.

• Explanation: Classes act as templates or blueprints for creating objects. They help in organizing code by encapsulating data and methods, and they support code reuse through the creation of multiple instances (objects) based on the same class.

2. Defining a Class:

• Use the class keyword followed by the class name and a colon.

• Explanation: In Python, the class keyword is used to define a new class. The class name follows it, and a colon indicates the beginning of the class body.

• Define attributes and methods within the class.

• Explanation: Inside the class body, you can define attributes (variables) and methods (functions). These collectively define the properties and behaviors of the objects created from the class.

3. Creating Objects (Instances):

• Instantiate a class to create objects.

• Explanation: To create objects (instances) of a class, you use the class name followed by parentheses. This process is known as instantiation.

• Objects encapsulate data and behavior defined by the class.

• Explanation: Objects encapsulate both the data (attributes) and behavior (methods) defined in the class. Each object is an independent instance with its own state and functionality.

4. Inheritance:

• Create subclasses to inherit attributes and methods from a parent class.

• Explanation: Inheritance allows the creation of a new class (subclass) that inherits properties and behaviors from an existing class (parent class).

• Subclasses can extend or modify inherited functionalities.

• Explanation: Subclasses have the option to add new attributes or methods, override existing methods, or extend the functionality inherited from the parent class.

5. Method Overriding:

• Subclasses can provide their own implementation of methods inherited from the parent class.

• Explanation: Method overriding allows a subclass to provide a specific implementation for a method that it inherits from the parent class.

• Enables customization and specialization.

• Explanation: This customization allows subclasses to specialize in certain behaviors while maintaining a shared structure with the parent class.

6. Multiple Inheritance:

• Inherit from multiple parent classes to combine their features.

• Explanation: A class can inherit from more than one parent class, allowing it to combine features from multiple sources.

• Follows a specific method resolution order.

• Explanation: When a method is called on an instance of a class with multiple inheritance, Python follows a specific order to determine which method to execute.

7. Advanced Topics:

• The section covers more advanced concepts related to classes.

8. Class Variables vs. Instance Variables:

• Class variables are shared among all instances.

• Explanation: Class variables are attributes shared by all instances of a class.

• Instance variables are unique to each object.

• Explanation: Instance variables are specific to each instance of a class and can have different values for each object.

9. Class Methods vs. Instance Methods:

• Class methods operate on the class itself.

• Explanation: Class methods can access and modify class-level attributes and are defined using the @classmethod decorator.

• Instance methods operate on individual instances.

• Explanation: Instance methods can access and modify instance-specific attributes and are the most common type of methods in a class.

10. Static Methods:

• Independent functions within a class.

• Explanation: Static methods are defined using the @staticmethod decorator and are not bound to class or instance state.

• Do not access or modify class or instance state.

• Explanation: They are self-contained and don't rely on or modify class or instance-specific data.

11. Encapsulation and Data Hiding:

• Use access modifiers to control visibility and access to attributes and methods.

• Explanation: Access modifiers like private (\_\_) can be used to control the visibility of attributes and methods.

• Enhances code security and maintainability.

• Explanation: Encapsulation helps protect the internal implementation details of a class, making it easier to maintain and less prone to external interference.

12. Use Cases and Examples:

• Demonstrations of how to apply classes in practical scenarios.

13. Best Practices:

• Recommendations for writing effective and maintainable class-based code.

14. Naming Conventions:

• Follow conventions for class and method names.

• Explanation: Adhering to naming conventions improves code readability and consistency.

15. Keeping Classes Small:

• Divide complex systems into focused, manageable classes.

• Explanation: Smaller classes with specific responsibilities are easier to understand, maintain, and reuse.

16. Effective Inheritance Use:

• Employ inheritance judiciously for code reuse and hierarchy organization.

• Explanation: Use inheritance when it logically makes sense and promotes code reuse, avoiding unnecessary complexity.

17. Testing and Debugging:

• Thoroughly test and debug classes for robustness.

• Explanation: Rigorous testing ensures that classes behave as expected and helps identify and fix any issues.

18. Conclusion:

• Python classes are essential for creating structured, reusable, and maintainable code.

• Explanation: Classes are a powerful tool in Python for creating organized and reusable code structures.

• Mastering classes enhances your ability to design and implement complex systems.

• Explanation: Proficiency in using classes allows developers to create sophisticated and well-organized software systems.