OOP

LECT.

### 1. Object-Oriented Programming (OOP) - Abstraction

Abstraction is one of the four fundamental principles of object-oriented programming (OOP), alongside encapsulation, inheritance, and polymorphism. It is the process of simplifying complex systems by modeling classes appropriate to the problem and working at the most relevant level of inheritance for a particular aspect of the problem. Here’s a detailed look at abstraction:

**What is Abstraction?**

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. It focuses on what an object does instead of how it does it. By doing this, abstraction reduces complexity and allows the programmer to focus on interactions at a high level rather than getting bogged down by low-level details.

**Benefits of Abstraction**

* **Reduces Complexity**: By abstracting away complex details, developers can manage larger codebases more effectively.
* **Enhances Maintainability**: Changes to the implementation details of a method or class don’t affect other parts of the program that depend on the abstracted method or class.
* **Improves Reusability**: Abstract classes and interfaces can be reused across different programs and projects, promoting code reuse.

2. Inheritance is a core principle of object-oriented programming (OOP) that allows a new class to inherit properties and behaviors (methods) from an existing class. The primary purpose of inheritance is to promote code reusability and establish a natural hierarchy between classes.

**Key Concepts of Inheritance**

1. **Base Class (Parent Class)**: This is the class whose properties and methods are inherited by another class. It is also known as the superclass.
2. **Derived Class (Child Class)**: This is the class that inherits from another class. It is also known as the subclass.
3. **Single Inheritance**: A derived class inherits from only one base class.
4. **Multiple Inheritance**: A derived class inherits from more than one base class.

**Benefits of Inheritance**

* **Code Reusability**: Allows the reuse of existing code, reducing redundancy.
* **Hierarchy**: Establishes a relationship between base and derived classes, reflecting a real-world hierarchy.
* **Maintenance**: Simplifies code maintenance, as changes in the base class automatically propagate to derived classes.

### 3. Object-Oriented Programming (OOP) - Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to bundling data (attributes) and methods (functions) that operate on the data into a single unit, or class. It restricts direct access to some of an object's components to prevent accidental interference and misuse. This concept is crucial for creating a clear structure and protecting the internal state of objects.

**Key Concepts of Encapsulation**

1. **Data Hiding**: Restricting access to certain details of an object’s state, typically using access modifiers.
2. **Access Modifiers**: Keywords used to set the access level for classes, methods, and variables:
   * public: Accessible from any other class.
   * private: Accessible only within its own class.
   * protected: Accessible within its own class and by derived class instances.

**Benefits of Encapsulation**

* **Improved Security**: Protects an object’s internal state from unintended interference and misuse.
* **Reduced Complexity**: By exposing only necessary parts of an object, reduces complexity and increases ease of maintenance.
* **Enhanced Flexibility**: Allows the internal implementation of a class to be changed without affecting other parts of the program.

### 4. Object-Oriented Programming (OOP) - Polymorphism

Polymorphism is a key concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class rather than their actual class. The term "polymorphism" means "many shapes" in Greek, and it enables a single interface to be used for different underlying forms (data types).

**Key Concepts of Polymorphism**

1. **Method Overloading**: Allows a class to have more than one method with the same name, provided their parameter lists are different. Method overloading is a form of compile-time polymorphism.
2. **Method Overriding**: Allows a subclass to provide a specific implementation of a method that is already defined in its superclass. Method overriding is a form of runtime polymorphism.

**Benefits of Polymorphism**

* **Flexibility**: Allows the same interface to be used for different data types.
* **Code Reusability**: Promotes the reuse of methods and reduces code duplication.
* **Maintainability**: Simplifies the addition of new features and functionality.

**Method Overloading vs. Method Overriding**

**Method Overloading**

* **Definition**: Method overloading allows multiple methods in the same class to have the same name but different parameters (number, type, or order).
* **Compile-time Polymorphism**: The method to be called is determined at compile time based on the method signature.
* **Not Supported Directly in Python**: Python does not support method overloading directly like some other languages (e.g., Java). However, it can be achieved using default arguments or variable-length arguments

**Method Overriding**

* **Definition**: Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
* **Runtime Polymorphism**: The method to be called is determined at runtime based on the object type.
* **Supported in Python**: Method overriding is directly supported in Python and is commonly used in inheritance.

#### Conclusion

Polymorphism in OOP is a powerful mechanism that allows objects of different classes to be treated as objects of a common superclass. Method overloading and method overriding are two ways to achieve polymorphism. Method overloading enables multiple methods in the same class with the same name but different parameters, while method overriding allows a subclass to provide a specific implementation of a method defined in its superclass. Both techniques enhance code flexibility, reusability, and maintainability.

### 5. Object-Oriented Programming (OOP) - Class vs Instance (or Object)

In object-oriented programming (OOP), classes and instances (or objects) are fundamental concepts. Understanding the distinction between them is crucial for effective programming in any OOP language.

**Class**

* **Definition**: A class is a blueprint or template for creating objects. It defines a set of attributes (data) and methods (functions) that the created objects will have.
* **Purpose**: Classes allow you to define the properties and behaviors that objects of that class will possess.
* **Syntax**: In Python, a class is defined using the class keyword.

**Instance (or Object)**

* **Definition**: An instance (or object) is a specific realization of a class. When a class is instantiated, an object is created, with its own unique set of attributes defined by the class.
* **Purpose**: Objects represent individual entities that hold data and can interact with other objects.
* **Creation**: An object is created by calling the class as if it were a function.

**Differences between Class and Instance**

1. **Blueprint vs. Object**: A class is a blueprint for creating objects, while an instance is a specific object created from that class.
2. **Memory Allocation**: A class does not allocate memory for its attributes; memory is allocated when an instance is created.
3. **Shared vs. Unique**: Attributes and methods defined in a class are shared by all instances, but the attribute values are unique to each instance.
4. **Initialization**: A class defines how its instances will be initialized and behave, but an instance is initialized with specific values.

### 6. Object-Oriented Programming (OOP) - Access Modifiers: Public, Private, Protected

Access modifiers are keywords used in object-oriented programming (OOP) to set the accessibility of classes, methods, and attributes. They are crucial for implementing encapsulation, a fundamental concept of OOP that restricts access to certain components to protect the internal state of an object. The main access modifiers are public, private, and protected.

**Public**

* **Definition**: Members (attributes or methods) declared as public are accessible from any other class.
* **Purpose**: Used for methods and attributes that need to be accessible from outside the class.
* **Syntax**: In Python, all members are public by default.

**Private**

* **Definition**: Members declared as private are accessible only within the class they are declared. Private members are not accessible from outside the class.
* **Purpose**: Used to hide the internal details and protect the integrity of the data.
* **Syntax**: In Python, private members are indicated by a double underscore prefix (\_\_).

**Protected**

* **Definition**: Members declared as protected are accessible within the class they are declared and in subclasses (derived classes). However, they are not accessible from outside these classes.
* **Purpose**: Used to allow controlled access to members for subclasses, but not for other classes.
* **Syntax**: In Python, protected members are indicated by a single underscore prefix (\_).

**Conclusion**

Access modifiers are essential for implementing encapsulation in OOP. They control the visibility and accessibility of class members:

* **Public**: Accessible from any other class. Suitable for methods and attributes that need to be widely accessible.
* **Private**: Accessible only within the class they are declared. Used to protect the internal state and hide implementation details.
* **Protected**: Accessible within the class they are declared and in subclasses. Used for controlled access in class hierarchies.

By properly using access modifiers, developers can create robust and maintainable code that safeguards the integrity of objects and their data.

1. OOP - Static vs instance properties.

### Instance Properties

Instance properties (or instance variables) belong to individual instances (objects) of a class. Each object has its own copy of these properties, meaning that changes to the propert.ies of one object do not affect the properties of another object of the same class.

#### Characteristics:

1. **Belong to Objects**: Instance properties are tied to individual instances of a class.
2. **Unique to Each Instance**: Each object has its own set of instance properties.
3. **Initialized by Constructor**: Usually initialized in the constructor (\_\_init\_\_ method in Python) of the class.
4. **Accessed Through the Instance**: Accessed and modified using the instance of the class.

### Static Properties

Static properties (or class variables) are shared among all instances of a class. They are defined at the class level and are not tied to any specific instance. This means that all instances of the class share the same value for these properties.

#### Characteristics:

1. **Belong to Class**: Static properties are associated with the class itself rather than any particular instance.
2. **Shared by All Instances**: All instances of the class share the same value of the static property.
3. **Defined in the Class Scope**: Declared directly within the class, outside any instance methods.
4. **Accessed Through Class or Instance**: Can be accessed using the class name or an instance, but typically accessed through the class name to avoid confusion.

### Key Differences

* **Scope**: Instance properties are specific to each object, while static properties are shared across all instances of a class.
* **Initialization**: Instance properties are usually initialized within the constructor, whereas static properties are defined at the class level.
* **Access**: Instance properties are accessed and modified through object instances. Static properties can be accessed through both the class and its instances, but typically through the class to indicate their shared nature.
* **Memory Allocation**: Each instance has its own copy of instance properties, leading to separate memory allocation for each object. Static properties are stored once at the class level, saving memory when many instances are created.

8. OOP - Constructors.

OOP - Constructors.

In Object-Oriented Programming (OOP), constructors are special methods used for initializing objects of a class. Constructors are typically called automatically when an object is created. They allow you to set up initial values for an object's attributes or perform any necessary setup tasks.

Constructors in Python

In Python, constructors are defined using the \_\_init\_\_ method. This method is called automatically when an object is created. Here's an example:

python

Copy code

class MyClass:

def \_\_init\_\_(self, parameter1, parameter2):

self.attribute1 = parameter1

self.attribute2 = parameter2

# Creating an object of MyClass

obj = MyClass("value1", "value2")

In this example:

\_\_init\_\_ is the constructor method.

self is a reference to the current instance of the class.

parameter1 and parameter2 are parameters passed to the constructor.

self.attribute1 and self.attribute2 are attributes of the object being created, initialized with the values of parameter1 and parameter2, respectively.

Default Constructor

If you don't define a constructor explicitly, Python provides a default constructor that takes no arguments. However, it's common practice to define your own constructor even if it doesn't take any parameters to set up initial values or perform other setup tasks.

python

Copy code

class MyClass:

def \_\_init\_\_(self):

# Default constructor

pass

Overloading Constructors

Python doesn't support method overloading in the traditional sense. However, you can achieve similar behavior by using default parameter values or multiple constructors with different names.

python

Copy code

class MyClass:

def \_\_init\_\_(self, parameter1=None, parameter2=None):

if parameter1 is None and parameter2 is None:

# Default constructor

pass

else:

# Constructor with parameters

self.attribute1 = parameter1

self.attribute2 = parameter2

Summary

Constructors are special methods used for initializing objects of a class.

In Python, constructors are defined using the \_\_init\_\_ method.

Constructors are called automatically when an object is created.

You can define default constructors and overload constructors using default parameter values or multiple constructor methods with different names.

Constructors are essential for setting up initial states of objects and ensuring proper initialization before objects are used.

9. OOP vs Procedural programming.

Object-Oriented Programming (OOP) and Procedural Programming are two different paradigms used for writing computer programs. Here's a comparison between them:

Object-Oriented Programming (OOP)

Focus: OOP focuses on objects and data rather than procedures and functions. It emphasizes encapsulation, inheritance, and polymorphism.

Objects: Programs are organized around objects, which represent real-world entities or concepts. Objects have attributes (data) and methods (functions) that operate on the data.

Encapsulation: OOP promotes encapsulation, where data and methods that operate on the data are bundled together within objects. This helps in hiding the internal implementation details and exposing only the necessary interfaces.

Inheritance: OOP allows classes (blueprints for objects) to inherit attributes and methods from other classes. This promotes code reusability and enables the creation of hierarchical relationships between classes.

Polymorphism: OOP supports polymorphism, which allows objects of different classes to be treated as objects of a common superclass. This enables more flexible and generic code.

Example Languages: Common examples of OOP languages include Java, Python, C++, and C#.

Procedural Programming

Focus: Procedural programming focuses on procedures or routines (functions) rather than objects. Programs are organized around procedures that operate on data.

Data and Functions: Data and functions are separate entities in procedural programming. Functions are defined to perform specific tasks on data.

Modularity: Procedural programming promotes modularity by breaking down a program into smaller, manageable procedures. Each procedure focuses on a specific task.

Top-down Approach: Procedural programs often follow a top-down approach, where the main program calls various procedures to accomplish a task.

Limited Reusability: While procedural programming supports reusability through functions, it often lacks the level of code reuse provided by inheritance in OOP.

Example Languages: Common examples of procedural programming languages include C, Fortran, and Pascal.

Comparison

Abstraction: OOP provides a higher level of abstraction through encapsulation, inheritance, and polymorphism, making it easier to model complex systems.

Code Reusability: OOP promotes code reusability through inheritance and polymorphism, leading to more maintainable and scalable codebases.

Complexity Handling: OOP is better suited for handling complex systems with multiple interacting objects and relationships.

Procedural Efficiency: In some cases, procedural programming may be more efficient for simple tasks or low-level system programming.

Ease of Understanding: Procedural programming can be easier to understand for beginners due to its linear, step-by-step execution flow.

In practice, the choice between OOP and procedural programming depends on factors such as the nature of the problem, project requirements, team expertise, and performance considerations.

1. OOP Design patterns - Creation patterns.

Object-Oriented Programming (OOP) design patterns are reusable solutions to common problems encountered in software design. Creation patterns are a category of design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Here are some common creation patterns:

1. Singleton Pattern

Intent: Ensure that a class has only one instance and provide a global point of access to it.

Example: Used in logging classes, database connection classes, and thread pool implementations.

2. Factory Method Pattern

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. It defines a method that must be implemented by subclasses to create objects.

Example: Used in frameworks where subclasses need to decide the type of object to create, such as UI frameworks.

3. Abstract Factory Pattern

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Example: Used in GUI toolkits to provide a way to create families of related UI components (e.g., buttons, checkboxes) without specifying their concrete types.

4. Builder Pattern

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Example: Used when creating objects with many optional parameters or configuration settings, such as creating HTML or XML documents.

5. Prototype Pattern

Intent: Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

Example: Used when the instantiation process is expensive or complex, and many similar objects are needed.

Summary

Creation patterns provide solutions to object creation problems, addressing issues such as ensuring single instances, providing flexible object creation mechanisms, and managing complex instantiation processes. By understanding and applying these patterns appropriately, developers can improve code reusability, maintainability, and flexibility in their object-oriented designs.

1. OOP Design patterns - Structural patterns.

Structural patterns are a category of design patterns in Object-Oriented Programming (OOP) that focus on the composition of classes and objects to form larger structures. They help to define relationships between classes or objects, making it easier to design flexible and efficient systems. Here are some common structural patterns:

1. Adapter Pattern

Intent: Convert the interface of a class into another interface clients expect. It allows incompatible interfaces to work together.

Example: Used when integrating new components or libraries into existing systems with different interfaces.

2. Bridge Pattern

Intent: Decouple an abstraction from its implementation so that the two can vary independently. It involves creating a bridge interface that decouples an abstraction from its implementation so that the two can vary independently.

Example: Used in GUI frameworks to separate platform-independent GUI code from platform-specific implementations.

3. Composite Pattern

Intent: Compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.

Example: Used in UI frameworks to represent hierarchical user interface elements like menus and tree structures.

4. Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

Example: Used in I/O classes to add functionalities like buffering, compression, or encryption to streams.

5. Facade Pattern

Intent: Provide a unified interface to a set of interfaces in a subsystem. It simplifies a complex system by providing a higher-level interface that makes it easier to use.

Example: Used to provide a simplified interface to a complex subsystem, like a software library or API.

6. Proxy Pattern

Intent: Provide a surrogate or placeholder for another object to control access to it. It allows for the creation of a representative object that controls access to another object.

Example: Used in scenarios where there is a need for controlled access to a resource, such as lazy initialization, access control, or logging.

Summary

Structural patterns help to design systems by defining relationships between classes and objects, enabling flexibility, and promoting code reuse. By understanding and applying these patterns appropriately, developers can create more modular, maintainable, and scalable software systems.

1. OOP Design patterns - Behavioral patterns.

Behavioral patterns are a category of design patterns in Object-Oriented Programming (OOP) that focus on communication between objects and how they operate together to achieve common behavioral patterns. They help to define interactions between objects, making it easier to design flexible and maintainable systems. Here are some common behavioral patterns:

### 1. Observer Pattern

\*\*Intent\*\*: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

\*\*Example\*\*: Used in event handling systems, GUI frameworks, and distributed systems for implementing publish-subscribe mechanisms.

### 2. Strategy Pattern

\*\*Intent\*\*: Define a family of algorithms, encapsulate each one, and make them interchangeable. It allows the algorithm to vary independently from clients that use it.

\*\*Example\*\*: Used in sorting algorithms, payment processing systems, and game development for defining different strategies or behaviors.

### 3. Command Pattern

\*\*Intent\*\*: Encapsulate a request as an object, thereby parameterizing clients with queues, requests, and operations. It allows for the decoupling of sender and receiver of a request.

\*\*Example\*\*: Used in GUI frameworks, transactional systems, and remote procedure call (RPC) mechanisms for implementing undo/redo functionality, queuing requests, and logging commands.

### 4. Iterator Pattern

\*\*Intent\*\*: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

\*\*Example\*\*: Used in collection classes like lists, arrays, and trees to traverse and access elements without knowing the internal structure.

### 5. Chain of Responsibility Pattern

\*\*Intent\*\*: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. It chains the receiving objects and passes the request along the chain until an object handles it.

\*\*Example\*\*: Used in event handling systems, logging frameworks, and authentication mechanisms for processing requests in a chain of handlers.

### 6. Interpreter Pattern

\*\*Intent\*\*: Define a grammar for interpreting sentences in a language and provide a way to evaluate sentences in a language.

\*\*Example\*\*: Used in compilers, interpreters, and natural language processing systems for parsing and evaluating expressions or statements.

### Summary

Behavioral patterns focus on defining interactions between objects to achieve common communication patterns and behaviors. By understanding and applying these patterns appropriately, developers can design more flexible, modular, and maintainable software systems.

1. OOP - What is SOLID.

SOLID is an acronym that represents five principles of object-oriented programming and design aimed at creating robust, maintainable, and scalable software systems. These principles were introduced by Robert C. Martin, also known as Uncle Bob, and have become fundamental concepts in software engineering. Here's what each letter in SOLID stands for:

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one responsibility or job.

Key Concept: Encourages high cohesion by ensuring that each class is responsible for a single, well-defined aspect of the system. This makes classes easier to understand, maintain, and extend.

2. Open/Closed Principle (OCP)

Definition: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Key Concept: Encourages software design that allows for adding new features or functionality without modifying existing code. This is typically achieved through abstraction, inheritance, and polymorphism.

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Key Concept: Ensures that subtypes must be substitutable for their base types without altering the correctness of the program. This principle is essential for maintaining behavioral compatibility and preventing unexpected behaviors in polymorphic code.

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use.

Key Concept: Promotes the creation of specific interfaces for clients, avoiding "fat" interfaces that include methods not relevant to all clients. This reduces the impact of changes and dependencies, leading to more flexible and maintainable code.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions, and abstractions should not depend on details. In other words, dependency should be on abstractions, not concretions.

Key Concept: Encourages decoupling between modules by introducing abstractions that define the interactions between them. This allows for flexible and interchangeable implementations, facilitating easier testing, maintenance, and evolution of the system.

Summary

SOLID principles provide guidelines for designing object-oriented software that is modular, maintainable, and extensible. By adhering to these principles, developers can create code that is easier to understand, modify, and maintain, ultimately leading to more robust and scalable software systems.

1. Relational Databases. What is ACID?.

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. It represents a set of properties that ensure reliability and consistency in database transactions. These properties are fundamental to the design and implementation of relational databases, providing guarantees regarding the integrity and reliability of data operations. Let's delve into each component of the ACID properties:

1. Atomicity

Definition: Atomicity ensures that each transaction is treated as a single, indivisible unit of work. Either all operations within the transaction are completed successfully and committed to the database, or none of them are. There is no intermediate state.

Example: Consider a banking transaction where funds are transferred from one account to another. Atomicity ensures that if the funds are debited from one account, they are also credited to the other account. If any part of the transaction fails (e.g., due to an error or interruption), the entire transaction is rolled back, and the database returns to its original state.

2. Consistency

Definition: Consistency ensures that a transaction brings the database from one valid state to another valid state. It preserves the integrity constraints and rules defined in the database schema.

Example: In a database containing information about students and courses, consistency ensures that if a student enrolls in a course, the course's maximum enrollment limit is not exceeded, and the student's enrollment does not violate any prerequisites or constraints defined in the database schema.

3. Isolation

Definition: Isolation ensures that the execution of multiple transactions concurrently does not result in interference or inconsistency. Each transaction appears to execute in isolation, as if it were the only transaction being executed in the system.

Example: Consider two transactions that involve updating the same row in a database concurrently. Isolation ensures that one transaction's changes are not visible to the other transaction until it completes. This prevents issues such as dirty reads, non-repeatable reads, and phantom reads.

4. Durability

Definition: Durability ensures that once a transaction is committed, its changes are permanent and survive system failures, crashes, or restarts. Committed transactions persist even in the event of power loss or hardware failures.

Example: If a transaction updates a record in a database and then commits the changes, durability ensures that the updated data remains intact even if the database server crashes immediately after the transaction commits. Upon recovery, the changes made by the committed transaction are still present in the database.

Summary

ACID properties provide a framework for ensuring the reliability, consistency, and durability of database transactions. By adhering to these properties, relational databases maintain data integrity and reliability, even in the face of concurrent operations, system failures, or other unexpected events. This ensures that the database remains a trusted source of information for applications and users.

1. Relational Databases. What is indexing and how it works.

Indexing in relational databases is a technique used to improve the performance of data retrieval operations, such as searching, sorting, and filtering, by creating special data structures known as indexes. These indexes provide a faster way to locate and access specific rows or ranges of rows within a database table. Here's how indexing works:

1. Creation of Indexes

Selection of Columns: Database administrators or developers select one or more columns of a table to be indexed based on the types of queries frequently executed on the table.

Data Structure: An index is created for each selected column, which contains a sorted list of values from that column along with pointers to the corresponding rows in the table.

2. Types of Indexes

Single-Column Index: An index created on a single column of a table.

Composite Index: An index created on multiple columns of a table, allowing for faster retrieval of data based on combinations of values from these columns.

3. How Indexing Works

Binary Search: Indexes are typically implemented using data structures like B-trees or hash tables, which allow for efficient searching and retrieval of data. When a query is executed that involves a column with an index, the database engine uses a binary search algorithm to locate the desired rows based on the indexed values.

Lookup Optimization: Instead of scanning the entire table sequentially, the database engine uses the index to quickly locate the relevant rows that satisfy the conditions specified in the query, significantly reducing the number of disk I/O operations and improving query performance.

4. Benefits of Indexing

Faster Data Retrieval: Indexes allow for faster retrieval of data, especially when querying large datasets or filtering based on specific criteria.

Improved Query Performance: Queries that involve indexed columns execute more efficiently, resulting in shorter response times and improved overall system performance.

Support for Constraints: Indexes can enforce constraints such as uniqueness and primary keys, ensuring data integrity and consistency within the database.

5. Considerations and Trade-offs

Overhead: Indexes consume additional storage space and require maintenance overhead during data modifications (e.g., insertions, updates, and deletions).

Selection of Columns: Careful consideration should be given to the selection of columns to be indexed, as indexing too many columns or the wrong columns can lead to decreased performance and increased overhead.

Query Patterns: Indexing should be based on the types of queries frequently executed on the table, prioritizing columns used in WHERE clauses, JOIN conditions, and ORDER BY clauses.

Summary

Indexing is a critical aspect of relational databases that significantly improves the performance of data retrieval operations. By creating indexes on selected columns, databases can quickly locate and access specific rows or ranges of rows, leading to faster query execution and improved overall system performance. However, indexing should be used judiciously, considering factors such as query patterns, overhead, and trade-offs to achieve the desired balance between performance and resource utilization.

1. Relational Databases. Database Denormalization.

Database denormalization is a database optimization technique used to improve the performance of read-heavy database operations by reducing the number of joins needed to retrieve data. It involves intentionally introducing redundancy into the database schema by storing duplicate or derived data in multiple locations, typically in separate tables. This redundancy eliminates the need for complex joins and enables faster data retrieval. Here's an overview of database denormalization:

1. Normalization vs. Denormalization

Normalization: In database normalization, data is organized into multiple related tables, and redundant data is minimized to reduce storage space and avoid update anomalies. It follows a set of rules, usually up to the third normal form (3NF), to ensure data integrity and reduce redundancy.

Denormalization: In contrast, denormalization intentionally introduces redundancy by duplicating data across tables or adding redundant columns to improve read performance. While denormalization may violate normalization principles, it can significantly improve query performance for certain types of queries.

2. Reasons for Denormalization

Performance Optimization: Denormalization is typically used to optimize read-heavy operations, such as complex queries involving multiple joins or aggregations, by reducing the number of table lookups and joins required to retrieve data.

Reduced Query Complexity: By storing redundant data in a denormalized form, queries can be simplified, resulting in shorter query execution times and improved overall system performance.

Support for Reporting and Analytics: Denormalization is often used in data warehouses and analytical databases to support reporting and analytics, where performance is critical, and real-time updates are less of a concern.

3. Techniques for Denormalization

Flattening Hierarchical Data: In hierarchical data structures, such as parent-child relationships, denormalization involves flattening the structure by duplicating parent data in child tables or vice versa.

Adding Redundant Columns: Denormalization can involve adding redundant columns to a table to store frequently accessed data from related tables, eliminating the need for joins.

Materialized Views: Materialized views are precomputed result sets stored as tables, allowing for denormalization of complex queries and aggregations, resulting in faster query execution.

4. Trade-offs and Considerations

Data Redundancy: Denormalization increases data redundancy, which can lead to increased storage requirements and potential data inconsistencies if updates are not properly managed.

Maintenance Overhead: Maintaining denormalized data requires careful consideration of data updates, insertions, and deletions to ensure data consistency and integrity.

Query Performance vs. Update Performance: While denormalization improves read performance, it may impact update performance, as updates may need to propagate changes across multiple redundant copies of data.

5. Use Cases for Denormalization

Reporting and Analytics: Denormalization is commonly used in data warehouses and analytical databases to improve query performance for reporting and analytics purposes.

Highly Concurrent Systems: In systems with high concurrency and read-heavy workloads, denormalization can reduce contention and improve scalability by minimizing the need for joins.

Caching and Materialized Views: Denormalization can be used to create precomputed result sets or materialized views for frequently accessed queries, improving response times for users.

Summary

Database denormalization is a technique used to optimize read performance in relational databases by introducing redundancy into the database schema. While denormalization can significantly improve query performance, it requires careful consideration of trade-offs, data consistency, and maintenance overhead. It is best suited for scenarios where read performance is critical, and the benefits outweigh the drawbacks.

17. Relational Databases. Foreign keys. Relationships types - one-to-one, one-to-many etc.

In relational databases, foreign keys are a fundamental concept used to establish relationships between tables. A foreign key is a column or a set of columns in one table that references the primary key (or a unique key) in another table. Foreign keys enforce referential integrity, ensuring that relationships between tables remain consistent. Here's an overview of foreign keys and relationship types:

Foreign Keys

Definition: A foreign key is a column or a combination of columns in a table that refers to the primary key (or a unique key) in another table.

Purpose: Foreign keys establish relationships between tables, enabling data integrity and enforcing referential integrity constraints.

Enforcement: When a foreign key constraint is defined, the database ensures that the values in the foreign key column(s) match the values in the referenced primary key (or unique key) column(s) of the related table.

Actions on Update/Delete: Foreign key constraints can specify actions to be taken on updates or deletes, such as cascading updates or deletes, restricting updates or deletes, or setting null values.

Relationship Types

1. One-to-One (1:1) Relationship

Definition: A one-to-one relationship exists when each record in one table is related to exactly one record in another table, and vice versa.

Example: One-to-one relationships are relatively rare in relational databases but may be used to model entities with optional or dependent attributes that can be separated into distinct tables.

2. One-to-Many (1

) Relationship

Definition: A one-to-many relationship exists when each record in one table can be related to multiple records in another table, but each record in the second table is related to exactly one record in the first table.

Example: A common example is the relationship between a parent table (e.g., "Customers") and a child table (e.g., "Orders"), where each customer can have multiple orders.

3. Many-to-One (M:1) Relationship

Definition: A many-to-one relationship is the inverse of a one-to-many relationship, where multiple records in one table can be related to a single record in another table.

Example: In the same example of customers and orders, many orders can be associated with a single customer.

4. Many-to-Many (M

) Relationship

Definition: A many-to-many relationship exists when multiple records in one table can be related to multiple records in another table.

Example: A common example is the relationship between students and classes in a school database, where each student can enroll in multiple classes, and each class can have multiple students.

Summary

Foreign keys are essential for establishing relationships between tables in relational databases, enforcing referential integrity, and maintaining data consistency. Understanding relationship types, such as one-to-one, one-to-many, many-to-one, and many-to-many, helps in modeling data structures effectively and ensuring that database designs accurately represent real-world relationships between entities.

18. SQL basics. DDL commands.

SQL (Structured Query Language) is a standard language used to interact with relational databases. DDL (Data Definition Language) commands are SQL commands used to define, modify, and manage the structure of database objects such as tables, indexes, views, and schemas. Here are some common DDL commands:

1. CREATE

The CREATE command is used to create database objects such as tables, indexes, views, and schemas.

Create Table: Creates a new table in the database.

sql

Copy code

CREATE TABLE TableName (

Column1 datatype1 constraints,

Column2 datatype2 constraints,

...

);

Create Index: Creates an index on one or more columns of a table to improve query performance.

sql

Copy code

CREATE INDEX IndexName ON TableName (Column1, Column2, ...);

Create View: Creates a virtual table based on the result set of a SELECT query.

sql

Copy code

CREATE VIEW ViewName AS

SELECT Column1, Column2, ...

FROM TableName

WHERE condition;

2. ALTER

The ALTER command is used to modify the structure of existing database objects.

Alter Table: Adds, modifies, or drops columns, constraints, or indexes from a table.

sql

Copy code

ALTER TABLE TableName

ADD ColumnName datatype constraints;

Alter Index: Modifies the definition of an existing index.

sql

Copy code

ALTER INDEX IndexName RENAME TO NewIndexName;

3. DROP

The DROP command is used to remove database objects such as tables, indexes, views, and schemas.

Drop Table: Deletes a table from the database.

sql

Copy code

DROP TABLE TableName;

Drop Index: Deletes an index from the database.

sql

Copy code

DROP INDEX IndexName;

Drop View: Deletes a view from the database.

sql

Copy code

DROP VIEW ViewName;

4. TRUNCATE

The TRUNCATE command is used to remove all rows from a table, while maintaining its structure.

Truncate Table: Deletes all rows from a table.

sql

Copy code

TRUNCATE TABLE TableName;

5. RENAME

The RENAME command is used to rename database objects such as tables, indexes, views, and schemas.

Rename Table: Renames a table in the database.

sql

Copy code

RENAME TABLE OldTableName TO NewTableName;

Rename Index: Renames an index in the database.

sql

Copy code

RENAME INDEX OldIndexName TO NewIndexName;

Summary

DDL commands are used to define, modify, and manage the structure of database objects in SQL. These commands allow database administrators and developers to create, alter, drop, and manage tables, indexes, views, and schemas, providing the foundation for storing and organizing data within a relational database system.

1. SQL basics. DML commands.

DML (Data Manipulation Language) commands are a subset of SQL commands used to manage data within database objects such as tables. Unlike DDL commands, which focus on the structure of the database, DML commands are used to manipulate the data stored within the database objects. Here are some common DML commands:

1. SELECT

The SELECT command is used to retrieve data from one or more tables in a database.

sql

Copy code

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

2. INSERT

The INSERT command is used to add new rows of data into a table.

sql

Copy code

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

3. UPDATE

The UPDATE command is used to modify existing data in a table.

sql

Copy code

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

4. DELETE

The DELETE command is used to remove rows of data from a table.

sql

Copy code

DELETE FROM table\_name

WHERE condition;

Summary

DML commands are essential for managing the data stored within database tables. With DML commands, you can retrieve, insert, update, and delete data, allowing you to manipulate the contents of tables according to your application's requirements. These commands form the backbone of database interactions and are crucial for performing data manipulation tasks in SQL.

1. SQL basics. Aggregation functions.

Aggregation functions, also known as aggregate functions, are SQL functions used to perform calculations on sets of values to return a single aggregated value. These functions are commonly used in SQL queries to summarize data and perform calculations across multiple rows. Here are some common aggregation functions in SQL:

1. COUNT()

The COUNT() function counts the number of rows in a result set or the number of non-null values in a specific column.

sql

Copy code

SELECT COUNT(column\_name) AS count\_alias

FROM table\_name;

2. SUM()

The SUM() function calculates the sum of values in a numeric column.

sql

Copy code

SELECT SUM(numeric\_column) AS sum\_alias

FROM table\_name;

3. AVG()

The AVG() function calculates the average (mean) of values in a numeric column.

sql

Copy code

SELECT AVG(numeric\_column) AS avg\_alias

FROM table\_name;

4. MIN()

The MIN() function returns the minimum value in a column.

sql

Copy code

SELECT MIN(column\_name) AS min\_alias

FROM table\_name;

5. MAX()

The MAX() function returns the maximum value in a column.

sql

Copy code

SELECT MAX(column\_name) AS max\_alias

FROM table\_name;

6. GROUP BY

The GROUP BY clause is used in conjunction with aggregate functions to group the result set by one or more columns and perform aggregate calculations within each group.

sql

Copy code

SELECT column1, aggregate\_function(column2)

FROM table\_name

GROUP BY column1;

7. HAVING

The HAVING clause is used to filter the result set based on aggregate values calculated using aggregate functions.

sql

Copy code

SELECT column1, aggregate\_function(column2)

FROM table\_name

GROUP BY column1

HAVING aggregate\_function(column2) > value;

Example:

Let's say we have a table named sales with columns product\_id, quantity, and unit\_price. We can use aggregate functions to calculate total sales quantity, total sales amount, and average unit price per product:

sql

Copy code

SELECT

product\_id,

SUM(quantity) AS total\_quantity,

SUM(quantity \* unit\_price) AS total\_sales\_amount,

AVG(unit\_price) AS avg\_unit\_price

FROM sales

GROUP BY product\_id;

Summary

Aggregate functions are powerful tools in SQL for summarizing data and performing calculations across multiple rows. By using aggregate functions along with grouping and filtering clauses, you can generate meaningful insights and summaries from your database tables.

1. SQL basics. Transactions. Commit, Rollback.

In SQL, a transaction is a sequence of one or more SQL statements that are executed as a single unit of work. Transactions allow you to ensure the consistency and integrity of your database by grouping related operations together and ensuring that they either all succeed or all fail. SQL provides commands like COMMIT and ROLLBACK to manage transactions effectively:

1. BEGIN TRANSACTION / START TRANSACTION

The BEGIN TRANSACTION or START TRANSACTION command is used to mark the beginning of a transaction. It indicates the start of a series of SQL statements that should be treated as a single unit of work.

sql

Copy code

BEGIN TRANSACTION;

-- SQL statements

COMMIT;

2. COMMIT

The COMMIT command is used to save the changes made during a transaction to the database. It marks the successful completion of the transaction and makes the changes permanent.

sql

Copy code

BEGIN TRANSACTION;

-- SQL statements

COMMIT;

3. ROLLBACK

The ROLLBACK command is used to undo the changes made during a transaction and restore the database to its state before the transaction began. It is typically used to handle errors or abort a transaction that cannot be completed successfully.

sql

Copy code

BEGIN TRANSACTION;

-- SQL statements

ROLLBACK;

Example:

Suppose we have a banking database with tables for accounts and transactions. We want to transfer funds from one account to another while ensuring that both debit and credit transactions occur atomically. We can use transactions to achieve this:

sql

Copy code

BEGIN TRANSACTION;

UPDATE accounts

SET balance = balance - 100

WHERE account\_number = 'sender\_account';

UPDATE accounts

SET balance = balance + 100

WHERE account\_number = 'receiver\_account';

COMMIT;

If any error occurs during the transaction (e.g., network failure, database constraint violation), we can roll back the changes to maintain data integrity:

sql

Copy code

BEGIN TRANSACTION;

UPDATE accounts

SET balance = balance - 100

WHERE account\_number = 'sender\_account';

UPDATE accounts

SET balance = balance + 100

WHERE account\_number = 'receiver\_account';

ROLLBACK;

Summary

Transactions in SQL provide a way to ensure data consistency and integrity by grouping related operations into atomic units of work. The COMMIT command is used to save the changes made during a transaction, while the ROLLBACK command is used to undo the changes and revert the database to its previous state. By using transactions effectively, you can ensure that database operations are reliable and maintainable, even in the face of errors or failures.

1. SQL basics. Types of joins.

In SQL, joins are used to combine rows from two or more tables based on related columns between them. There are several types of joins in SQL, each serving different purposes and producing different results. Here are the most common types of joins:

1. INNER JOIN

An INNER JOIN returns rows that have matching values in both tables.

sql

Copy code

SELECT \*

FROM table1

INNER JOIN table2

ON table1.column = table2.column;

2. LEFT JOIN (or LEFT OUTER JOIN)

A LEFT JOIN returns all rows from the left table (table1), and the matched rows from the right table (table2). If there is no match, NULL values are returned for the columns from the right table.

sql

Copy code

SELECT \*

FROM table1

LEFT JOIN table2

ON table1.column = table2.column;

3. RIGHT JOIN (or RIGHT OUTER JOIN)

A RIGHT JOIN returns all rows from the right table (table2), and the matched rows from the left table (table1). If there is no match, NULL values are returned for the columns from the left table.

sql

Copy code

SELECT \*

FROM table1

RIGHT JOIN table2

ON table1.column = table2.column;

4. FULL JOIN (or FULL OUTER JOIN)

A FULL JOIN returns all rows when there is a match in either left (table1) or right (table2) table. If there is no match, NULL values are returned for the columns from the table without a match.

sql

Copy code

SELECT \*

FROM table1

FULL JOIN table2

ON table1.column = table2.column;

5. CROSS JOIN

A CROSS JOIN returns the Cartesian product of rows from the tables involved. It combines each row from the first table with every row from the second table.

sql

Copy code

SELECT \*

FROM table1

CROSS JOIN table2;

Summary

SQL joins allow you to combine data from multiple tables based on related columns. The choice of join type depends on the specific requirements of your query and the relationship between the tables. By understanding the different types of joins and how they work, you can write SQL queries to retrieve and manipulate data effectively from your database.

1. SQL basics. Primary key. Unique key. Null values.

In SQL, primary keys and unique keys are constraints used to enforce data integrity and ensure uniqueness within a table. Additionally, NULL values represent missing or unknown information in a database. Let's discuss each concept in detail:

1. Primary Key

Definition: A primary key is a column or a set of columns that uniquely identifies each row in a table. It must contain unique values and cannot contain NULL values.

Usage: Primary keys are used to enforce entity integrity and ensure that each row in a table is uniquely identifiable.

Syntax: Primary keys are defined using the PRIMARY KEY constraint when creating or altering a table.

sql

Copy code

CREATE TABLE table\_name (

column1 datatype PRIMARY KEY,

column2 datatype,

...

);

2. Unique Key

Definition: A unique key is a constraint that ensures that the values in a column or a set of columns are unique within a table. Unlike primary keys, unique keys can allow NULL values.

Usage: Unique keys are used to enforce uniqueness constraints on columns that do not serve as the primary key but should have unique values.

Syntax: Unique keys are defined using the UNIQUE constraint when creating or altering a table.

sql

Copy code

CREATE TABLE table\_name (

column1 datatype UNIQUE,

column2 datatype,

...

);

3. Null Values

Definition: NULL is a special value in SQL that represents missing or unknown information. It is different from zero or an empty string, as it signifies the absence of a value rather than a specific value.

Usage: NULL values can be used to indicate missing data or to represent unknown information in a database.

Behavior: By default, most columns in SQL tables allow NULL values unless explicitly constrained to disallow them using the NOT NULL constraint.

sql

Copy code

CREATE TABLE table\_name (

column1 datatype NOT NULL,

column2 datatype,

...

);

Summary

Primary keys ensure the uniqueness of each row in a table and are used to identify individual records. Unique keys enforce uniqueness constraints on columns, allowing NULL values. NULL values represent missing or unknown information in the database and can be used in columns unless constrained by the NOT NULL constraint. Understanding these concepts is essential for designing and maintaining a well-structured and normalized database schema.

1. Python basics. Lists, tuples, sets, dictionaries.

In Python, lists, tuples, sets, and dictionaries are built-in data structures used to store collections of elements. Each data structure has its own characteristics, usage, and methods. Let's explore each of them:

1. Lists

Definition: Lists are ordered collections of elements, which can be of any data type (e.g., integers, strings, other lists).

Characteristics:

Mutable: Elements can be added, removed, or modified.

Ordered: Elements are stored in a specific order and can be accessed by their index.

Example:

python

Copy code

my\_list = [1, 2, 3, 'a', 'b', 'c']

Usage:

Storing collections of similar or heterogeneous data.

Manipulating and iterating over sequences of elements.

2. Tuples

Definition: Tuples are ordered collections of elements, similar to lists, but they are immutable, meaning their elements cannot be changed after creation.

Characteristics:

Immutable: Once created, elements cannot be added, removed, or modified.

Ordered: Elements are stored in a specific order and can be accessed by their index.

Example:

python

Copy code

my\_tuple = (1, 2, 3, 'a', 'b', 'c')

Usage:

Representing fixed collections of elements, such as coordinates, database records, or function return values.

Ensuring data integrity by preventing accidental modifications.

3. Sets

Definition: Sets are unordered collections of unique elements, meaning they cannot contain duplicate values.

Characteristics:

Mutable: Elements can be added or removed, but individual elements cannot be modified.

Unordered: Elements are not stored in any specific order, and indexing is not supported.

Unique: Sets automatically remove duplicate elements.

Example:

python

Copy code

my\_set = {1, 2, 3, 'a', 'b', 'c'}

Usage:

Eliminating duplicate elements from a collection.

Performing set operations such as union, intersection, and difference.

4. Dictionaries

Definition: Dictionaries are unordered collections of key-value pairs, where each key is associated with a value.

Characteristics:

Mutable: Key-value pairs can be added, removed, or modified.

Unordered: Key-value pairs are not stored in any specific order, and indexing is not supported.

Keys: Must be unique and hashable (immutable), such as strings, integers, or tuples.

Example:

python

Copy code

my\_dict = {'a': 1, 'b': 2, 'c': 3}

Usage:

Storing and retrieving data using meaningful keys.

Representing structured data with key-value associations.

Summary

Lists, tuples, sets, and dictionaries are fundamental data structures in Python, each serving different purposes and offering different characteristics. Understanding their differences and choosing the appropriate data structure for your needs is essential for writing clean, efficient, and maintainable Python code.

25. Python basics. Threads vs Processes.

In Python, both threads and processes are used to achieve parallelism and concurrency, allowing programs to execute multiple tasks simultaneously. However, they operate differently and have distinct characteristics. Let's explore the differences between threads and processes:

Threads

Definition: Threads are the smallest units of execution within a process. Multiple threads can exist within a single process and share the same memory space.

Characteristics:

Lightweight: Threads are lightweight compared to processes, as they share resources such as memory and file descriptors.

Shared Memory: Threads within the same process share memory, allowing them to communicate and exchange data easily.

Global Interpreter Lock (GIL): In CPython, the global interpreter lock restricts execution to a single thread at a time, making it challenging to achieve true parallelism with threads in CPU-bound tasks.

Usage:

Suitable for I/O-bound tasks such as network requests, file I/O, and database operations, where waiting for external resources dominates the execution time.

Convenient for concurrent execution of multiple tasks that require coordination or communication.

Processes

Definition: Processes are independent units of execution with their own memory space. Each process runs in its own address space and does not share memory with other processes by default.

Characteristics:

Heavyweight: Processes are heavier than threads, as they require separate memory space and resources.

Isolated Memory: Processes have separate memory spaces, providing isolation and protection against memory corruption.

True Parallelism: Processes can achieve true parallelism on multi-core systems, as they run concurrently and independently of each other.

Usage:

Suitable for CPU-bound tasks such as complex computations, mathematical operations, and data processing, where parallelism can significantly improve performance.

Useful for running independent tasks that do not require communication or coordination between them.

Comparison

Concurrency vs. Parallelism: Threads achieve concurrency, allowing multiple tasks to run concurrently within the same process, while processes achieve true parallelism, running independently of each other.

Communication and Synchronization: Threads within the same process can communicate and synchronize easily through shared memory, while processes require inter-process communication (IPC) mechanisms such as pipes, queues, or shared memory to exchange data.

Resource Management: Threads are more lightweight and consume fewer resources than processes, making them more suitable for applications that require many concurrent tasks.

Safety and Isolation: Processes provide stronger isolation and safety guarantees, as they have separate memory spaces, reducing the risk of memory corruption and unintended side effects.

Summary

Threads and processes are both used for achieving parallelism and concurrency in Python, but they operate differently and have distinct characteristics. Threads are lightweight units of execution within a process, suitable for I/O-bound tasks and concurrent execution of multiple tasks. Processes are independent units of execution with separate memory spaces, suitable for CPU-bound tasks and achieving true parallelism on multi-core systems. Understanding the differences between threads and processes is essential for designing and implementing concurrent and parallel applications in Python.

Object Oriented Design

1.Object Oriented Design - Products and multi-level categories.

class Product:

def \_\_init\_\_(self, product\_id, name, description, price):

self.product\_id = product\_id

self.name = name

self.description = description

self.price = price

def display\_info(self):

print(f"ID: {self.product\_id}, Name: {self.name}, Description: {self.description}, Price: ${self.price}")

class Category:

def \_\_init\_\_(self, category\_id, name, parent=None):

self.category\_id = category\_id

self.name = name

self.parent = parent

self.products = []

self.subcategories = []

def add\_product(self, product):

self.products.append(product)

def add\_subcategory(self, category):

self.subcategories.append(category)

def display\_info(self):

print(f"ID: {self.category\_id}, Name: {self.name}")

if self.parent:

print(f"Parent Category: {self.parent.name}")

if self.products:

print("Products:")

for product in self.products:

print(f"- {product.name}")

if self.subcategories:

print("Subcategories:")

for subcategory in self.subcategories:

print(f"- {subcategory.name}")

2.Object Oriented Design - Chats, group chats. Members.

class User:

def \_\_init\_\_(self, user\_id, username, email, password):

self.user\_id = user\_id

self.username = username

self.email = email

self.password = password

self.profile\_picture = None

self.bio = None

def authenticate(self, password):

return self.password == password

def update\_profile(self, profile\_picture=None, bio=None):

if profile\_picture:

self.profile\_picture = profile\_picture

if bio:

self.bio = bio

class Chat:

def \_\_init\_\_(self, chat\_id):

self.chat\_id = chat\_id

self.members = []

self.messages = []

def add\_member(self, user\_id):

self.members.append(user\_id)

def remove\_member(self, user\_id):

if user\_id in self.members:

self.members.remove(user\_id)

def send\_message(self, message):

self.messages.append(message)

def display\_chat\_history(self):

for message in self.messages:

print(message)

class GroupChat(Chat):

def \_\_init\_\_(self, chat\_id, group\_name):

super().\_\_init\_\_(chat\_id)

self.group\_name = group\_name

def update\_group\_name(self, new\_name):

self.group\_name = new\_name

def invite\_member(self, user\_id):

self.members.append(user\_id)

def remove\_member(self, user\_id):

if user\_id in self.members:

self.members.remove(user\_id)

3.Object Oriented Design - Bank accounts. Transactions.  
from datetime import datetime

class Account:

def \_\_init\_\_(self, account\_number, account\_type, balance, owner):

self.account\_number = account\_number

self.account\_type = account\_type

self.balance = balance

self.owner = owner

def deposit(self, amount):

self.balance += amount

return amount

def withdrawal(self, amount):

if self.balance >= amount:

self.balance -= amount

return amount

else:

return "Insufficient funds"

def transfer(self, amount, recipient\_account):

withdrawal\_amount = self.withdrawal(amount)

if isinstance(withdrawal\_amount, str):

return withdrawal\_amount

else:

recipient\_account.deposit(withdrawal\_amount)

return withdrawal\_amount

def display\_account\_info(self):

print(f"Account Number: {self.account\_number}")

print(f"Account Type: {self.account\_type}")

print(f"Balance: ${self.balance}")

print(f"Owner: {self.owner.username}")

class Transaction:

def \_\_init\_\_(self, transaction\_id, transaction\_type, amount, sender\_account, receiver\_account=None):

self.transaction\_id = transaction\_id

self.transaction\_type = transaction\_type

self.amount = amount

self.timestamp = datetime.now()

self.sender\_account = sender\_account

self.receiver\_account = receiver\_account

def display\_transaction\_info(self):

print(f"Transaction ID: {self.transaction\_id}")

print(f"Transaction Type: {self.transaction\_type}")

print(f"Amount: ${self.amount}")

print(f"Timestamp: {self.timestamp}")

print(f"Sender Account: {self.sender\_account.account\_number}")

if self.receiver\_account:

print(f"Receiver Account: {self.receiver\_account.account\_number}")

class User:

def \_\_init\_\_(self, user\_id, username, email, password):

self.user\_id = user\_id

self.username = username

self.email = email

self.password = password

self.accounts = []

def authenticate(self, password):

return self.password == password

def add\_account(self, account):

self.accounts.append(account)

def remove\_account(self, account):

if account in self.accounts:

self.accounts.remove(account)

def display\_user\_info(self):

print(f"User ID: {self.user\_id}")

print(f"Username: {self.username}")

print(f"Email: {self.email}")

4.Object Oriented Design - Quizzes, questions.

class Question:

def \_\_init\_\_(self, question\_text, options, correct\_option, question\_type):

self.question\_text = question\_text

self.options = options

self.correct\_option = correct\_option

self.question\_type = question\_type

def display\_question(self):

print(self.question\_text)

if self.question\_type == "Multiple Choice":

for i, option in enumerate(self.options):

print(f"{i + 1}. {option}")

elif self.question\_type == "True/False":

print("1. True")

print("2. False")

def check\_answer(self, user\_answer):

return user\_answer == self.correct\_option

class Quiz:

def \_\_init\_\_(self):

self.questions = []

self.total\_score = 0

def add\_question(self, question):

self.questions.append(question)

def remove\_question(self, question):

if question in self.questions:

self.questions.remove(question)

def take\_quiz(self):

for question in self.questions:

question.display\_question()

user\_answer = input("Your answer: ")

if question.check\_answer(user\_answer):

self.total\_score += 1

def display\_quiz\_results(self):

print(f"Total Score: {self.total\_score}/{len(self.questions)}")

1. Object Oriented Design - Books, genres and authors. Multiple authors.

class Author:

def \_\_init\_\_(self, author\_id, name):

self.author\_id = author\_id

self.name = name

self.books = []

def add\_book(self, book):

self.books.append(book)

def remove\_book(self, book):

if book in self.books:

self.books.remove(book)

def display\_info(self):

print(f"Author ID: {self.author\_id}")

print(f"Name: {self.name}")

if self.books:

print("Books:")

for book in self.books:

print(f"- {book.title}")

class Genre:

def \_\_init\_\_(self, genre\_id, name):

self.genre\_id = genre\_id

self.name = name

self.books = []

def add\_book(self, book):

self.books.append(book)

def remove\_book(self, book):

if book in self.books:

self.books.remove(book)

def display\_info(self):

print(f"Genre ID: {self.genre\_id}")

print(f"Name: {self.name}")

if self.books:

print("Books:")

for book in self.books:

print(f"- {book.title}")

class Book:

def \_\_init\_\_(self, book\_id, title, genre):

self.book\_id = book\_id

self.title = title

self.authors = []

self.genre = genre

def add\_author(self, author):

self.authors.append(author)

def remove\_author(self, author):

if author in self.authors:

self.authors.remove(author)

def display\_info(self):

print(f"Book ID: {self.book\_id}")

print(f"Title: {self.title}")

if self.authors:

print("Authors:")

for author in self.authors:

print(f"- {author.name}")

print(f"Genre: {self.genre.name}")

6. Object Oriented Design - Calendar and events.

User

Represents a user who can have one or more calendars and can create, update, or delete events.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name, email):

self.user\_id = user\_id

self.name = name

self.email = email

self.calendars = []

def create\_calendar(self, name):

calendar = Calendar(name, self)

self.calendars.append(calendar)

return calendar

def delete\_calendar(self, calendar):

self.calendars.remove(calendar)

def get\_calendars(self):

return self.calendars

Calendar

Represents a collection of events and belongs to a user.

python

Copy code

class Calendar:

def \_\_init\_\_(self, name, owner):

self.name = name

self.owner = owner

self.events = []

def add\_event(self, event):

self.events.append(event)

def remove\_event(self, event):

self.events.remove(event)

def get\_events(self):

return self.events

Event

Represents an event with a title, description, start and end times, location, and reminders.

python

Copy code

class Event:

def \_\_init\_\_(self, title, description, start\_time, end\_time, location=None):

self.title = title

self.description = description

self.start\_time = start\_time

self.end\_time = end\_time

self.location = location

self.reminders = []

def add\_reminder(self, reminder):

self.reminders.append(reminder)

def remove\_reminder(self, reminder):

self.reminders.remove(reminder)

def get\_reminders(self):

return self.reminders

Reminder

Represents a reminder for an event.

python

Copy code

class Reminder:

def \_\_init\_\_(self, time\_before\_event, message):

self.time\_before\_event = time\_before\_event

self.message = message

Location

Represents the location of an event.

python

Copy code

class Location:

def \_\_init\_\_(self, name, address):

self.name = name

self.address = address

7. Object Oriented Design - Departments. Employees. Managers.

Department

Represents a department within a company, which can have multiple employees and a manager.

python

Copy code

class Department:

def \_\_init\_\_(self, department\_id, name):

self.department\_id = department\_id

self.name = name

self.employees = []

self.manager = None

def add\_employee(self, employee):

self.employees.append(employee)

employee.department = self

def remove\_employee(self, employee):

self.employees.remove(employee)

employee.department = None

def set\_manager(self, manager):

self.manager = manager

manager.department = self

def get\_employees(self):

return self.employees

def get\_manager(self):

return self.manager

Employee

Represents an employee working in a department. An employee can have a manager.

python

Copy code

class Employee:

def \_\_init\_\_(self, employee\_id, name, position, salary):

self.employee\_id = employee\_id

self.name = name

self.position = position

self.salary = salary

self.department = None

self.manager = None

def assign\_manager(self, manager):

self.manager = manager

def get\_details(self):

return {

'ID': self.employee\_id,

'Name': self.name,

'Position': self.position,

'Salary': self.salary,

'Department': self.department.name if self.department else None,

'Manager': self.manager.name if self.manager else None

}

Manager

Represents a manager who is also an employee. A manager can manage a department and other employees.

python

Copy code

class Manager(Employee):

def \_\_init\_\_(self, employee\_id, name, position, salary):

super().\_\_init\_\_(employee\_id, name, position, salary)

self.subordinates = []

def add\_subordinate(self, employee):

self.subordinates.append(employee)

employee.assign\_manager(self)

1. Object Oriented Design - Posts. Reactions. Comments.

User

Represents a user who can create posts, reactions, and comments.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name):

self.user\_id = user\_id

self.name = name

self.posts = []

self.reactions = []

self.comments = []

def create\_post(self, content):

post = Post(content, self)

self.posts.append(post)

return post

def react\_to\_post(self, post, reaction\_type):

reaction = Reaction(self, post, reaction\_type)

self.reactions.append(reaction)

post.add\_reaction(reaction)

return reaction

def comment\_on\_post(self, post, content):

comment = Comment(content, self, post)

self.comments.append(comment)

post.add\_comment(comment)

return comment

def get\_posts(self):

return self.posts

def get\_reactions(self):

return self.reactions

def get\_comments(self):

return self.comments

Post

Represents a post created by a user. A post can have multiple reactions and comments.

python

Copy code

class Post:

def \_\_init\_\_(self, content, author):

self.post\_id = id(self) # Using `id` for unique identifier

self.content = content

self.author = author

self.reactions = []

self.comments = []

def add\_reaction(self, reaction):

self.reactions.append(reaction)

def add\_comment(self, comment):

self.comments.append(comment)

def get\_reactions(self):

return self.reactions

def get\_comments(self):

return self.comments

Reaction

Represents a reaction to a post. A reaction is made by a user and can be of various types (like, love, etc.).

python

Copy code

class Reaction:

def \_\_init\_\_(self, user, post, reaction\_type):

self.reaction\_id = id(self) # Using `id` for unique identifier

self.user = user

self.post = post

self.reaction\_type = reaction\_type

Comment

Represents a comment on a post. A comment is made by a user.

python

Copy code

class Comment:

def \_\_init\_\_(self, content, author, post):

self.comment\_id = id(self) # Using `id` for unique identifier

self.content = content

self.author = author

self.post = post

1. Object Oriented Design - Polls. Questions. Answers.

User

Represents a user who can create polls, respond to questions, and provide answers.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name):

self.user\_id = user\_id

self.name = name

self.polls = []

self.responses = []

def create\_poll(self, title):

poll = Poll(title, self)

self.polls.append(poll)

return poll

def respond\_to\_question(self, question, selected\_option):

response = Response(self, question, selected\_option)

self.responses.append(response)

question.add\_response(response)

return response

def get\_polls(self):

return self.polls

def get\_responses(self):

return self.responses

Poll

Represents a poll which can have multiple questions.

python

Copy code

class Poll:

def \_\_init\_\_(self, title, creator):

self.poll\_id = id(self) # Using `id` for unique identifier

self.title = title

self.creator = creator

self.questions = []

def add\_question(self, question):

self.questions.append(question)

def get\_questions(self):

return self.questions

Question

Represents a question within a poll. A question can have multiple possible answers.

python

Copy code

class Question:

def \_\_init\_\_(self, text):

self.question\_id = id(self) # Using `id` for unique identifier

self.text = text

self.answers = []

self.responses = []

def add\_answer(self, answer):

self.answers.append(answer)

def add\_response(self, response):

self.responses.append(response)

def get\_answers(self):

return self.answers

def get\_responses(self):

return self.responses

Answer

Represents a possible answer to a question.

python

Copy code

class Answer:

def \_\_init\_\_(self, text):

self.answer\_id = id(self) # Using `id` for unique identifier

self.text = text

Response

Represents a user's response to a question, selecting one of the possible answers.

python

Copy code

class Response:

def \_\_init\_\_(self, user, question, selected\_option):

self.response\_id = id(self) # Using `id` for unique identifier

self.user = user

self.question = question

self.selected\_option = selected\_option

1. Object Oriented Design - Students. Exams. Results.

Student

Represents a student who can take exams and have results.

python

Copy code

class Student:

def \_\_init\_\_(self, student\_id, name):

self.student\_id = student\_id

self.name = name

self.results = []

def add\_result(self, result):

self.results.append(result)

def get\_results(self):

return self.results

Exam

Represents an exam which students can take.

python

Copy code

class Exam:

def \_\_init\_\_(self, exam\_id, subject, date):

self.exam\_id = exam\_id

self.subject = subject

self.date = date

self.results = []

def add\_result(self, result):

self.results.append(result)

def get\_results(self):

return self.results

Result

Represents a result of a student in a particular exam.

python

Copy code

class Result:

def \_\_init\_\_(self, student, exam, score):

self.result\_id = id(self) # Using `id` for unique identifier

self.student = student

self.exam = exam

self.score = score

student.add\_result(self)

exam.add\_result(self)

def get\_details(self):

return {

'Student': self.student.name,

'Exam': self.exam.subject,

'Score': self.score

}

1. Object Oriented Design - Software. Licences. Users.

User

Represents a user who can have multiple licenses for different software.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name, email):

self.user\_id = user\_id

self.name = name

self.email = email

self.licenses = []

def add\_license(self, license):

self.licenses.append(license)

def get\_licenses(self):

return self.licenses

Software

Represents a software application which can have multiple licenses issued to users.

python

Copy code

class Software:

def \_\_init\_\_(self, software\_id, name, version):

self.software\_id = software\_id

self.name = name

self.version = version

self.licenses = []

def add\_license(self, license):

self.licenses.append(license)

def get\_licenses(self):

return self.licenses

License

Represents a license issued to a user for a specific software.

python

Copy code

class License:

def \_\_init\_\_(self, license\_id, user, software, license\_key, expiration\_date):

self.license\_id = license\_id

self.user = user

self.software = software

self.license\_key = license\_key

self.expiration\_date = expiration\_date

user.add\_license(self)

software.add\_license(self)

def get\_details(self):

return {

'User': self.user.name,

'Software': self.software.name,

'License Key': self.license\_key,

'Expiration Date': self.expiration\_date

}

1. Object Oriented Design - Teachers. Students. Classes.

Teacher

Represents a teacher who can teach multiple classes.

python

Copy code

class Teacher:

def \_\_init\_\_(self, teacher\_id, name):

self.teacher\_id = teacher\_id

self.name = name

self.classes = []

def add\_class(self, class\_):

self.classes.append(class\_)

def get\_classes(self):

return self.classes

Student

Represents a student who can enroll in multiple classes.

python

Copy code

class Student:

def \_\_init\_\_(self, student\_id, name):

self.student\_id = student\_id

self.name = name

self.classes = []

def enroll\_in\_class(self, class\_):

self.classes.append(class\_)

def get\_classes(self):

return self.classes

Class

Represents a class which is taught by a teacher and can have multiple students enrolled.

python

Copy code

class Class:

def \_\_init\_\_(self, class\_id, subject, teacher):

self.class\_id = class\_id

self.subject = subject

self.teacher = teacher

self.students = []

teacher.add\_class(self)

def add\_student(self, student):

self.students.append(student)

student.enroll\_in\_class(self)

def get\_students(self):

return self.students

def get\_teacher(self):

return self.teacher

1. Object Oriented Design - Hotel. Rooms. Bookings.

Hotel

Represents a hotel which can have multiple rooms.

python

Copy code

class Hotel:

def \_\_init\_\_(self, hotel\_id, name, location):

self.hotel\_id = hotel\_id

self.name = name

self.location = location

self.rooms = []

def add\_room(self, room):

self.rooms.append(room)

def get\_rooms(self):

return self.rooms

Room

Represents a room in the hotel which can be booked by guests.

python

Copy code

class Room:

def \_\_init\_\_(self, room\_id, number, room\_type, price\_per\_night):

self.room\_id = room\_id

self.number = number

self.room\_type = room\_type

self.price\_per\_night = price\_per\_night

self.bookings = []

def add\_booking(self, booking):

self.bookings.append(booking)

def get\_bookings(self):

return self.bookings

Booking

Represents a booking for a room by a guest.

python

Copy code

class Booking:

def \_\_init\_\_(self, booking\_id, guest, room, check\_in\_date, check\_out\_date):

self.booking\_id = booking\_id

self.guest = guest

self.room = room

self.check\_in\_date = check\_in\_date

self.check\_out\_date = check\_out\_date

room.add\_booking(self)

def get\_details(self):

return {

'Guest': self.guest.name,

'Room Number': self.room.number,

'Check-in Date': self.check\_in\_date,

'Check-out Date': self.check\_out\_date

}

Guest

Represents a guest who can make bookings.

python

Copy code

class Guest:

def \_\_init\_\_(self, guest\_id, name, email):

self.guest\_id = guest\_id

self.name = name

self.email = email

self.bookings = []

def add\_booking(self, booking):

self.bookings.append(booking)

def get\_bookings(self):

return self.bookings

1. Object Oriented Design - Events. Tickets. Customers.

Event

Represents an event which can have multiple tickets available for sale.

python

Copy code

class Event:

def \_\_init\_\_(self, event\_id, name, date, location):

self.event\_id = event\_id

self.name = name

self.date = date

self.location = location

self.tickets = []

def add\_ticket(self, ticket):

self.tickets.append(ticket)

def get\_tickets(self):

return self.tickets

Ticket

Represents a ticket for an event which can be purchased by customers.

python

Copy code

class Ticket:

def \_\_init\_\_(self, ticket\_id, event, price, quantity):

self.ticket\_id = ticket\_id

self.event = event

self.price = price

self.quantity = quantity

def buy\_ticket(self):

if self.quantity > 0:

self.quantity -= 1

return True

else:

return False

Customer

Represents a customer who can purchase tickets for events.

python

Copy code

class Customer:

def \_\_init\_\_(self, customer\_id, name, email):

self.customer\_id = customer\_id

self.name = name

self.email = email

self.tickets = []

def buy\_ticket(self, ticket):

if ticket.buy\_ticket():

self.tickets.append(ticket)

return True

else:

return False

def get\_tickets(self):

return self.tickets

15. Object Oriented Design - Restaurant menu. Customers. Orders.

MenuItem

Represents an item on the restaurant's menu.

python

Copy code

class MenuItem:

def \_\_init\_\_(self, item\_id, name, description, price):

self.item\_id = item\_id

self.name = name

self.description = description

self.price = price

Restaurant

Represents the restaurant, which contains a menu.

python

Copy code

class Restaurant:

def \_\_init\_\_(self, name):

self.name = name

self.menu = []

def add\_menu\_item(self, menu\_item):

self.menu.append(menu\_item)

def get\_menu(self):

return self.menu

Customer

Represents a customer who can place orders.

python

Copy code

class Customer:

def \_\_init\_\_(self, name):

self.name = name

self.orders = []

def place\_order(self, order):

self.orders.append(order)

def get\_orders(self):

return self.orders

Order

Represents an order placed by a customer.

python

Copy code

class Order:

def \_\_init\_\_(self, order\_id, customer):

self.order\_id = order\_id

self.customer = customer

self.items = []

def add\_item(self, menu\_item, quantity=1):

self.items.append((menu\_item, quantity))

def get\_total\_price(self):

total\_price = sum(item[0].price \* item[1] for item in self.items)

return total\_price

16. Object Oriented Design - Patients. Doctors. Appointments.

Patient

Represents a patient who can schedule appointments with doctors.

python

Copy code

class Patient:

def \_\_init\_\_(self, patient\_id, name):

self.patient\_id = patient\_id

self.name = name

self.appointments = []

def schedule\_appointment(self, appointment):

self.appointments.append(appointment)

def get\_appointments(self):

return self.appointments

Doctor

Represents a doctor who can have appointments with patients.

python

Copy code

class Doctor:

def \_\_init\_\_(self, doctor\_id, name, specialty):

self.doctor\_id = doctor\_id

self.name = name

self.specialty = specialty

self.appointments = []

def schedule\_appointment(self, appointment):

self.appointments.append(appointment)

def get\_appointments(self):

return self.appointments

Appointment

Represents an appointment scheduled between a patient and a doctor.

python

Copy code

class Appointment:

def \_\_init\_\_(self, appointment\_id, patient, doctor, date, time):

self.appointment\_id = appointment\_id

self.patient = patient

self.doctor = doctor

self.date = date

self.time = time

patient.schedule\_appointment(self)

doctor.schedule\_appointment(self)

1. Object Oriented Design - Job Listings. Applicants. Interviews.

JobListing

Represents a job listing posted by a company.

python

Copy code

class JobListing:

def \_\_init\_\_(self, job\_id, title, description, requirements):

self.job\_id = job\_id

self.title = title

self.description = description

self.requirements = requirements

self.applicants = []

def add\_applicant(self, applicant):

self.applicants.append(applicant)

def get\_applicants(self):

return self.applicants

Applicant

Represents a person who applies for a job listing.

python

Copy code

class Applicant:

def \_\_init\_\_(self, applicant\_id, name, resume):

self.applicant\_id = applicant\_id

self.name = name

self.resume = resume

self.interviews = []

def apply(self, job\_listing):

job\_listing.add\_applicant(self)

def schedule\_interview(self, interview):

self.interviews.append(interview)

def get\_interviews(self):

return self.interviews

Interview

Represents an interview scheduled for an applicant for a specific job listing.

python

Copy code

class Interview:

def \_\_init\_\_(self, interview\_id, applicant, job\_listing, date, time):

self.interview\_id = interview\_id

self.applicant = applicant

self.job\_listing = job\_listing

self.date = date

self.time = time

applicant.schedule\_interview(self)

def get\_details(self):

return {

'Applicant': self.applicant.name,

'Job Title': self.job\_listing.title,

'Date': self.date,

'Time': self.time

}

1. Object Oriented Design - Articles. Users. Views.

Article

Represents an article published on a website.

python

Copy code

class Article:

def \_\_init\_\_(self, article\_id, title, content, author):

self.article\_id = article\_id

self.title = title

self.content = content

self.author = author

self.views = []

def add\_view(self, view):

self.views.append(view)

def get\_views(self):

return self.views

User

Represents a user who can view articles.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name):

self.user\_id = user\_id

self.name = name

def view\_article(self, article):

view = View(article=article, user=self)

article.add\_view(view)

View

Represents a view of an article by a user.

python

Copy code

class View:

def \_\_init\_\_(self, article, user):

self.article = article

self.user = user

1. Object Oriented Design - Users. Friends. Friend requests.

User

Represents a user of the system who can send and receive friend requests.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name):

self.user\_id = user\_id

self.name = name

self.friends = []

self.received\_requests = []

def send\_friend\_request(self, user):

request = FriendRequest(sender=self, receiver=user)

user.received\_requests.append(request)

def accept\_friend\_request(self, request):

if request in self.received\_requests:

sender = request.sender

self.friends.append(sender)

sender.friends.append(self)

self.received\_requests.remove(request)

return True

return False

def get\_friend\_requests(self):

return self.received\_requests

FriendRequest

Represents a friend request sent from one user to another.

python

Copy code

class FriendRequest:

def \_\_init\_\_(self, sender, receiver):

self.sender = sender

self.receiver = receiver

1. Object Oriented Design - Invoices. Products/Services. Suppliers.

Invoice

Represents an invoice for products/services provided by suppliers.

python

Copy code

class Invoice:

def \_\_init\_\_(self, invoice\_id, date, supplier):

self.invoice\_id = invoice\_id

self.date = date

self.supplier = supplier

self.items = []

def add\_item(self, item):

self.items.append(item)

def get\_total\_amount(self):

total\_amount = sum(item.get\_total\_price() for item in self.items)

return total\_amount

ProductService

Represents a product or service offered by suppliers.

python

Copy code

class ProductService:

def \_\_init\_\_(self, product\_id, name, price):

self.product\_id = product\_id

self.name = name

self.price = price

def get\_total\_price(self, quantity=1):

return self.price \* quantity

Supplier

Represents a supplier providing products/services.

python

Copy code

class Supplier:

def \_\_init\_\_(self, supplier\_id, name):

self.supplier\_id = supplier\_id

self.name = name

self.invoices = []

def generate\_invoice(self, date):

invoice = Invoice(invoice\_id=len(self.invoices) + 1, date=date, supplier=self)

self.invoices.append(invoice)

return invoice

def get\_invoices(self):

return self.invoices

1. Object Oriented Design - Channels. Subscribers.

Channel

Represents a communication channel that delivers content to subscribers.

python

Copy code

class Channel:

def \_\_init\_\_(self, channel\_id, name):

self.channel\_id = channel\_id

self.name = name

self.subscribers = []

def subscribe(self, subscriber):

if subscriber not in self.subscribers:

self.subscribers.append(subscriber)

subscriber.subscribe\_to\_channel(self)

def unsubscribe(self, subscriber):

if subscriber in self.subscribers:

self.subscribers.remove(subscriber)

subscriber.unsubscribe\_from\_channel(self)

def notify\_subscribers(self, message):

for subscriber in self.subscribers:

subscriber.receive\_notification(self, message)

Subscriber

Represents a user who subscribes to channels to receive notifications.

python

Copy code

class Subscriber:

def \_\_init\_\_(self, subscriber\_id, name):

self.subscriber\_id = subscriber\_id

self.name = name

self.channels = []

def subscribe\_to\_channel(self, channel):

self.channels.append(channel)

def unsubscribe\_from\_channel(self, channel):

if channel in self.channels:

self.channels.remove(channel)

def receive\_notification(self, channel, message):

print(f"Notification for {self.name} from {channel.name}: {message}")

1. Object Oriented Design - Packages. Tracking.

Package

Represents a package that is being shipped or delivered.

python

Copy code

class Package:

def \_\_init\_\_(self, package\_id, sender, recipient, status="Pending"):

self.package\_id = package\_id

self.sender = sender

self.recipient = recipient

self.status = status

def update\_status(self, status):

self.status = status

TrackingSystem

Represents a tracking system that tracks the status of packages.

python

Copy code

class TrackingSystem:

def \_\_init\_\_(self):

self.packages = {}

def add\_package(self, package):

self.packages[package.package\_id] = package

def update\_package\_status(self, package\_id, status):

if package\_id in self.packages:

package = self.packages[package\_id]

package.update\_status(status)

def track\_package\_status(self, package\_id):

if package\_id in self.packages:

return self.packages[package\_id].status

else:

return "Package not found"

1. Object Oriented Design - Events. Halls. Tickets. Sales.

Event

Represents an event that takes place in a specific hall.

python

Copy code

class Event:

def \_\_init\_\_(self, event\_id, name, date, hall):

self.event\_id = event\_id

self.name = name

self.date = date

self.hall = hall

self.tickets = []

def add\_ticket(self, ticket):

self.tickets.append(ticket)

def get\_available\_tickets(self):

return [ticket for ticket in self.tickets if not ticket.is\_sold()]

def get\_total\_tickets\_sold(self):

return sum(ticket.quantity\_sold for ticket in self.tickets)

Hall

Represents a hall where events take place.

python

Copy code

class Hall:

def \_\_init\_\_(self, hall\_id, name, capacity):

self.hall\_id = hall\_id

self.name = name

self.capacity = capacity

Ticket

Represents a ticket for an event.

python

Copy code

class Ticket:

def \_\_init\_\_(self, ticket\_id, event, price, quantity):

self.ticket\_id = ticket\_id

self.event = event

self.price = price

self.quantity = quantity

self.quantity\_sold = 0

def is\_sold(self):

return self.quantity\_sold >= self.quantity

def sell\_ticket(self, quantity=1):

if self.quantity\_sold + quantity <= self.quantity:

self.quantity\_sold += quantity

return True

else:

return False

SalesSystem

Represents a sales system for managing ticket sales.

python

Copy code

class SalesSystem:

def \_\_init\_\_(self):

self.events = []

def add\_event(self, event):

self.events.append(event)

def sell\_ticket(self, event\_id, ticket\_id, quantity=1):

for event in self.events:

if event.event\_id == event\_id:

for ticket in event.tickets:

if ticket.ticket\_id == ticket\_id:

return ticket.sell\_ticket(quantity)

return False

1. Object Oriented Design - LinkedIn skills. Users. Endorsements.

Skill

Represents a skill that users can add to their LinkedIn profiles.

python

Copy code

class Skill:

def \_\_init\_\_(self, skill\_name):

self.skill\_name = skill\_name

self.endorsements = []

def add\_endorsement(self, endorsement):

self.endorsements.append(endorsement)

def get\_endorsement\_count(self):

return len(self.endorsements)

User

Represents a user profile on LinkedIn.

python

Copy code

class User:

def \_\_init\_\_(self, user\_id, name):

self.user\_id = user\_id

self.name = name

self.skills = {}

def add\_skill(self, skill):

if skill.skill\_name not in self.skills:

self.skills[skill.skill\_name] = skill

def endorse\_skill(self, skill):

if skill.skill\_name in self.skills:

endorsement = Endorsement(endorser=self, skill=skill)

skill.add\_endorsement(endorsement)

Endorsement

Represents an endorsement given by a user for a skill.

python

Copy code

class Endorsement:

def \_\_init\_\_(self, endorser, skill):

self.endorser = endorser

self.skill = skill

1. Object Oriented Design - Employees. Checkins. Checkouts.

Employee

Represents an employee in the organization.

python

Copy code

class Employee:

def \_\_init\_\_(self, employee\_id, name):

self.employee\_id = employee\_id

self.name = name

self.checkins = []

self.checkouts = []

def check\_in(self, checkin):

self.checkins.append(checkin)

def check\_out(self, checkout):

self.checkouts.append(checkout)

def get\_last\_checkin(self):

if self.checkins:

return self.checkins[-1]

else:

return None

def get\_last\_checkout(self):

if self.checkouts:

return self.checkouts[-1]

else:

return None

Checkin

Represents an employee's check-in time.

python

Copy code

class Checkin:

def \_\_init\_\_(self, checkin\_time):

self.checkin\_time = checkin\_time

Checkout

Represents an employee's check-out time.

python

Copy code

class Checkout:

def \_\_init\_\_(self, checkout\_time):

self.checkout\_time = checkout\_time

CheckinCheckoutSystem

Represents a system for managing employee check-ins and check-outs.

python

Copy code

class CheckinCheckoutSystem:

def \_\_init\_\_(self):

self.employees = {}

def add\_employee(self, employee):

self.employees[employee.employee\_id] = employee

def check\_in\_employee(self, employee\_id, checkin\_time):

if employee\_id in self.employees:

employee = self.employees[employee\_id]

checkin = Checkin(checkin\_time)

employee.check\_in(checkin)

def check\_out\_employee(self, employee\_id, checkout\_time):

if employee\_id in self.employees:

employee = self.employees[employee\_id]

checkout = Checkout(checkout\_time)

employee.check\_out(checkout)

def get\_employee\_last\_checkin(self, employee\_id):

if employee\_id in self.employees:

employee = self.employees[employee\_id]

return employee.get\_last\_checkin()

else:

return None

def get\_employee\_last\_checkout(self, employee\_id):

if employee\_id in self.employees:

employee = self.employees[employee\_id]

return employee.get\_last\_checkout()

else:

return None

Database Design

1. Database Design - Products and multi-level categories:

sql

Copy code

CREATE TABLE Category (

CategoryID INT PRIMARY KEY,

Name NVARCHAR(100),

ParentCategoryID INT NULL REFERENCES Category(CategoryID)

);

CREATE TABLE Product (

ProductID INT PRIMARY KEY,

Name NVARCHAR(100),

Description NVARCHAR(MAX),

Price DECIMAL(10, 2),

CategoryID INT REFERENCES Category(CategoryID)

);

2. Database Design - Chats, group chats. Members:

sql

Copy code

CREATE TABLE Chat (

ChatID INT PRIMARY KEY,

Type NVARCHAR(50)

);

CREATE TABLE Member (

MemberID INT PRIMARY KEY,

UserID INT REFERENCES [User](UserID),

ChatID INT REFERENCES Chat(ChatID)

);

3. Database Design - Bank accounts. Transactions:

sql

Copy code

CREATE TABLE Account (

AccountID INT PRIMARY KEY,

UserID INT REFERENCES [User](UserID),

AccountType NVARCHAR(50),

Balance DECIMAL(18, 2)

);

CREATE TABLE Transaction (

TransactionID INT PRIMARY KEY,

AccountID INT REFERENCES Account(AccountID),

Type NVARCHAR(50),

Amount DECIMAL(18, 2),

Timestamp DATETIME

);

4. Database Design - Quizzes, questions:

sql

Copy code

CREATE TABLE Question (

QuestionID INT PRIMARY KEY,

Text NVARCHAR(MAX),

Type NVARCHAR(50),

CorrectOption NVARCHAR(10)

);

CREATE TABLE Quiz (

QuizID INT PRIMARY KEY,

Name NVARCHAR(100)

);

CREATE TABLE QuizQuestion (

QuizID INT REFERENCES Quiz(QuizID),

QuestionID INT REFERENCES Question(QuestionID),

PRIMARY KEY (QuizID, QuestionID)

);

5. Database Design - Books, genres and authors. Multiple authors:

sql

Copy code

CREATE TABLE Author (

AuthorID INT PRIMARY KEY,

Name NVARCHAR(100)

);

CREATE TABLE Genre (

GenreID INT PRIMARY KEY,

Name NVARCHAR(100)

);

CREATE TABLE Book (

BookID INT PRIMARY KEY,

Title NVARCHAR(100),

GenreID INT REFERENCES Genre(GenreID)

);

CREATE TABLE BookAuthor (

BookID INT REFERENCES Book(BookID),

AuthorID INT REFERENCES Author(AuthorID),

PRIMARY KEY (BookID, AuthorID)

);

6. Database Design - Calendar and events:

sql

Copy code

CREATE TABLE Event (

EventID INT PRIMARY KEY,

Title NVARCHAR(100),

Description NVARCHAR(MAX),

StartDateTime DATETIME,

EndDateTime DATETIME,

UserID INT REFERENCES [User](UserID)

);

7. Database Design - Departments, Employees, Managers

sql

Copy code

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(255) NOT NULL

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(255) NOT NULL,

DepartmentID INT,

ManagerID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID),

FOREIGN KEY (ManagerID) REFERENCES Employees(EmployeeID)

);

8. Database Design - Posts, Reactions, Comments

sql

Copy code

CREATE TABLE Posts (

PostID INT PRIMARY KEY,

Content TEXT NOT NULL,

CreatedAt DATETIME NOT NULL

);

CREATE TABLE Reactions (

ReactionID INT PRIMARY KEY,

PostID INT,

ReactionType VARCHAR(50),

CreatedAt DATETIME NOT NULL,

FOREIGN KEY (PostID) REFERENCES Posts(PostID)

);

CREATE TABLE Comments (

CommentID INT PRIMARY KEY,

PostID INT,

Content TEXT NOT NULL,

CreatedAt DATETIME NOT NULL,

FOREIGN KEY (PostID) REFERENCES Posts(PostID)

);

9. Database Design - Polls, Questions, Answers

sql

Copy code

CREATE TABLE Polls (

PollID INT PRIMARY KEY,

PollName VARCHAR(255) NOT NULL

);

CREATE TABLE Questions (

QuestionID INT PRIMARY KEY,

PollID INT,

QuestionText TEXT NOT NULL,

FOREIGN KEY (PollID) REFERENCES Polls(PollID)

);

CREATE TABLE Answers (

AnswerID INT PRIMARY KEY,

QuestionID INT,

AnswerText TEXT NOT NULL,

FOREIGN KEY (QuestionID) REFERENCES Questions(QuestionID)

);

10. Database Design - Students, Exams, Results

sql

Copy code

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

StudentName VARCHAR(255) NOT NULL

);

CREATE TABLE Exams (

ExamID INT PRIMARY KEY,

ExamName VARCHAR(255) NOT NULL

);

CREATE TABLE Results (

ResultID INT PRIMARY KEY,

StudentID INT,

ExamID INT,

Score DECIMAL(5,2),

FOREIGN KEY (StudentID) REFERENCES Students(StudentID),

FOREIGN KEY (ExamID) REFERENCES Exams(ExamID)

);

11. Database Design - Software, Licenses, Users

sql

Copy code

CREATE TABLE Software (

SoftwareID INT PRIMARY KEY,

SoftwareName VARCHAR(255) NOT NULL

);

CREATE TABLE Licenses (

LicenseID INT PRIMARY KEY,

SoftwareID INT,

LicenseKey VARCHAR(255) NOT NULL,

ExpirationDate DATE,

FOREIGN KEY (SoftwareID) REFERENCES Software(SoftwareID)

);

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(255) NOT NULL,

LicenseID INT,

FOREIGN KEY (LicenseID) REFERENCES Licenses(LicenseID)

);

12. Database Design - Teachers, Students, Classes

sql

Copy code

CREATE TABLE Teachers (

TeacherID INT PRIMARY KEY,

TeacherName VARCHAR(255) NOT NULL

);

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

StudentName VARCHAR(255) NOT NULL

);

CREATE TABLE Classes (

ClassID INT PRIMARY KEY,

ClassName VARCHAR(255) NOT NULL,

TeacherID INT,

FOREIGN KEY (TeacherID) REFERENCES Teachers(TeacherID)

);

CREATE TABLE Enrollments (

EnrollmentID INT PRIMARY KEY,

ClassID INT,

StudentID INT,

FOREIGN KEY (ClassID) REFERENCES Classes(ClassID),

FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

);

13. Database Design - Hotel, Rooms, Bookings

sql

Copy code

CREATE TABLE Hotels (

HotelID INT PRIMARY KEY,

HotelName VARCHAR(255) NOT NULL

);

CREATE TABLE Rooms (

RoomID INT PRIMARY KEY,

HotelID INT,

RoomNumber VARCHAR(10) NOT NULL,

RoomType VARCHAR(50),

FOREIGN KEY (HotelID) REFERENCES Hotels(HotelID)

);

CREATE TABLE Bookings (

BookingID INT PRIMARY KEY,

RoomID INT,

CustomerName VARCHAR(255) NOT NULL,

CheckInDate DATE NOT NULL,

CheckOutDate DATE NOT NULL,

FOREIGN KEY (RoomID) REFERENCES Rooms(RoomID)

);

14. Database Design - Events, Tickets, Customers

sql

Copy code

CREATE TABLE Events (

EventID INT PRIMARY KEY,

EventName VARCHAR(255) NOT NULL,

EventDate DATETIME NOT NULL

);

CREATE TABLE Tickets (

TicketID INT PRIMARY KEY,

EventID INT,

TicketType VARCHAR(50),

Price DECIMAL(10,2),

FOREIGN KEY (EventID) REFERENCES Events(EventID)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(255) NOT NULL

);

CREATE TABLE TicketSales (

SaleID INT PRIMARY KEY,

TicketID INT,

CustomerID INT,

SaleDate DATETIME NOT NULL,

FOREIGN KEY (TicketID) REFERENCES Tickets(TicketID),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

15. Database Design - Restaurant Menu, Customers, Orders

sql

Copy code

CREATE TABLE MenuItems (

MenuItemID INT PRIMARY KEY,

ItemName VARCHAR(255) NOT NULL,

Price DECIMAL(10,2)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(255) NOT NULL

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATETIME NOT NULL,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE OrderItems (

OrderItemID INT PRIMARY KEY,

OrderID INT,

MenuItemID INT,

Quantity INT NOT NULL,

FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),

FOREIGN KEY (MenuItemID) REFERENCES MenuItems(MenuItemID)

);

16. Database Design - Patients, Doctors, Appointments

sql

Copy code

CREATE TABLE Patients (

PatientID INT PRIMARY KEY,

PatientName VARCHAR(255) NOT NULL,

DateOfBirth DATE

);

CREATE TABLE Doctors (

DoctorID INT PRIMARY KEY,

DoctorName VARCHAR(255) NOT NULL,

Specialty VARCHAR(255)

);

CREATE TABLE Appointments (

AppointmentID INT PRIMARY KEY,

PatientID INT,

DoctorID INT,

AppointmentDate DATETIME NOT NULL,

Reason TEXT,

FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),

FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)

);

17. Database Design - Job Listings, Applicants, Interviews

sql

Copy code

CREATE TABLE JobListings (

JobID INT PRIMARY KEY,

JobTitle VARCHAR(255) NOT NULL,

JobDescription TEXT NOT NULL,

PostedDate DATE NOT NULL

);

CREATE TABLE Applicants (

ApplicantID INT PRIMARY KEY,

ApplicantName VARCHAR(255) NOT NULL,

Resume TEXT

);

CREATE TABLE Interviews (

InterviewID INT PRIMARY KEY,

JobID INT,

ApplicantID INT,

InterviewDate DATETIME NOT NULL,

InterviewOutcome VARCHAR(255),

FOREIGN KEY (JobID) REFERENCES JobListings(JobID),

FOREIGN KEY (ApplicantID) REFERENCES Applicants(ApplicantID)

);

18. Database Design - Articles, Users, Views

sql

Copy code

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(255) NOT NULL,

Email VARCHAR(255) NOT NULL

);

CREATE TABLE Articles (

ArticleID INT PRIMARY KEY,

AuthorID INT,

Title VARCHAR(255) NOT NULL,

Content TEXT NOT NULL,

PublishedDate DATE NOT NULL,

FOREIGN KEY (AuthorID) REFERENCES Users(UserID)

);

CREATE TABLE Views (

ViewID INT PRIMARY KEY,

ArticleID INT,

UserID INT,

ViewDate DATETIME NOT NULL,

FOREIGN KEY (ArticleID) REFERENCES Articles(ArticleID),

FOREIGN KEY (UserID) REFERENCES Users(UserID)

);

19. Database Design - Users, Friends, Friend Requests

sql

Copy code

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(255) NOT NULL,

Email VARCHAR(255) NOT NULL

);

CREATE TABLE Friends (

UserID1 INT,

UserID2 INT,

PRIMARY KEY (UserID1, UserID2),

FOREIGN KEY (UserID1) REFERENCES Users(UserID),

FOREIGN KEY (UserID2) REFERENCES Users(UserID)

);

CREATE TABLE FriendRequests (

RequestID INT PRIMARY KEY,

FromUserID INT,

ToUserID INT,

RequestDate DATETIME NOT NULL,

Status VARCHAR(50),

FOREIGN KEY (FromUserID) REFERENCES Users(UserID),

FOREIGN KEY (ToUserID) REFERENCES Users(UserID)

);

20. Database Design - Invoices, Products/Services, Suppliers

sql

Copy code

CREATE TABLE Suppliers (

SupplierID INT PRIMARY KEY,

SupplierName VARCHAR(255) NOT NULL,

ContactInfo TEXT

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(255) NOT NULL,

SupplierID INT,

Price DECIMAL(10,2),

FOREIGN KEY (SupplierID) REFERENCES Suppliers(SupplierID)

);

CREATE TABLE Invoices (

InvoiceID INT PRIMARY KEY,

InvoiceDate DATE NOT NULL,

CustomerName VARCHAR(255) NOT NULL

);

CREATE TABLE InvoiceItems (

InvoiceItemID INT PRIMARY KEY,

InvoiceID INT,

ProductID INT,

Quantity INT NOT NULL,

FOREIGN KEY (InvoiceID) REFERENCES Invoices(InvoiceID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

21. Database Design - Channels, Subscribers

sql

Copy code

CREATE TABLE Channels (

ChannelID INT PRIMARY KEY,

ChannelName VARCHAR(255) NOT NULL,

Description TEXT

);

CREATE TABLE Subscribers (

SubscriberID INT PRIMARY KEY,

ChannelID INT,

UserID INT,

SubscriptionDate DATE NOT NULL,

FOREIGN KEY (ChannelID) REFERENCES Channels(ChannelID),

FOREIGN KEY (UserID) REFERENCES Users(UserID)

);

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(255) NOT NULL,

Email VARCHAR(255) NOT NULL

);

22. Database Design - Packages, Tracking

sql

Copy code

CREATE TABLE Packages (

PackageID INT PRIMARY KEY,

SenderName VARCHAR(255) NOT NULL,

ReceiverName VARCHAR(255) NOT NULL,

Weight DECIMAL(10,2),

Dimensions VARCHAR(255)

);

CREATE TABLE Tracking (

TrackingID INT PRIMARY KEY,

PackageID INT,

Location VARCHAR(255) NOT NULL,

Status VARCHAR(255) NOT NULL,

UpdateDate DATETIME NOT NULL,

FOREIGN KEY (PackageID) REFERENCES Packages(PackageID)

);

23. Database Design - Events, Halls, Tickets, Sales

sql

Copy code

CREATE TABLE Events (

EventID INT PRIMARY KEY,

EventName VARCHAR(255) NOT NULL,

EventDate DATETIME NOT NULL

);

CREATE TABLE Halls (

HallID INT PRIMARY KEY,

HallName VARCHAR(255) NOT NULL,

Capacity INT NOT NULL

);

CREATE TABLE Tickets (

TicketID INT PRIMARY KEY,

EventID INT,

HallID INT,

TicketType VARCHAR(50),

Price DECIMAL(10,2),

FOREIGN KEY (EventID) REFERENCES Events(EventID),

FOREIGN KEY (HallID) REFERENCES Halls(HallID)

);

CREATE TABLE Sales (

SaleID INT PRIMARY KEY,

TicketID INT,

CustomerName VARCHAR(255) NOT NULL,

SaleDate DATETIME NOT NULL,

FOREIGN KEY (TicketID) REFERENCES Tickets(TicketID)

);

24. Database Design - LinkedIn Skills, Users, Endorsements

sql

Copy code

CREATE TABLE Users (

UserID INT PRIMARY KEY,

UserName VARCHAR(255) NOT NULL,

Email VARCHAR(255) NOT NULL

);

CREATE TABLE Skills (

SkillID INT PRIMARY KEY,

SkillName VARCHAR(255) NOT NULL

);

CREATE TABLE UserSkills (

UserID INT,

SkillID INT,

PRIMARY KEY (UserID, SkillID),

FOREIGN KEY (UserID) REFERENCES Users(UserID),

FOREIGN KEY (SkillID) REFERENCES Skills(SkillID)

);

CREATE TABLE Endorsements (

EndorsementID INT PRIMARY KEY,

UserID INT,

SkillID INT,

EndorserID INT,

EndorsementDate DATETIME NOT NULL,

FOREIGN KEY (UserID) REFERENCES Users(UserID),

FOREIGN KEY (SkillID) REFERENCES Skills(SkillID),

FOREIGN KEY (EndorserID) REFERENCES Users(UserID)

);

25. Database Design - Employees, Checkins, Checkouts

sql

Copy code

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(255) NOT NULL

);

CREATE TABLE Checkins (

CheckinID INT PRIMARY KEY,

EmployeeID INT,

CheckinTime DATETIME NOT NULL,

FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)

);

CREATE TABLE Checkouts (

CheckoutID INT PRIMARY KEY,

EmployeeID INT,

CheckoutTime DATETIME NOT NULL,

FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)

);

Leetcode

1. PathSum II

Iterative way

def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:

if not root:

return []

stack = [(root, [root.val], root.val)]

result = []

while stack:

node, path, current\_sum = stack.pop()

if not node.left and not node.right:

if current\_sum == targetSum:

result.append(path)

if node.right:

stack.append((node.right, path + [node.right.val], current\_sum + node.right.val))

if node.left:

stack.append((node.left, path + [node.left.val], current\_sum + node.left.val))

return result

Recursive way

def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:

stack=[]

def pathsum(node,cur):

if not node:

return

curr=cur+[node.val]

if not (node.left or node.right):

if sum(curr)==targetSum:

stack.append(curr)

return

pathsum(node.left,curr)

pathsum(node.right,curr)

pathsum(root,[])

return stack

1. Reverse Odd Levels of Binary Tree.

Iterative way

def reverseOddLevels(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

        if not root:

            return

        queue = deque([root])

        level = 0

        while queue:

            level\_size = len(queue)

            nodes = list(queue)

            if level % 2 == 1:

                for i in range(level\_size // 2):

                    nodes[i].val, nodes[-i-1].val = nodes[-i-1].val, nodes[i].val

            for \_ in range(level\_size):

                node = queue.popleft()

                if node.left:

                    queue.append(node.left)

                if node.right:

                    queue.append(node.right)

            level += 1

        return root

Recursive way

def reverseOddLevels(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

def reverse(r1,r2,check=False):

if r1 is None or r2 is None:

return

if check:

rr1=r1.val

r1.val=r2.val

r2.val=rr1

reverse(r1.left,r2.right, not check)

reverse(r2.left,r1.right, not check)

reverse(root.left,root.right,True)

return root

1. Interleaving String.

Iterative way

def isInterleave(self, s1: str, s2: str, s3: str) -> bool:

        if len(s3)!=len(s1)+len(s2):

            return False

        dp=[[False]\*(len(s2)+1) for \_ in range(len(s1)+1)]

        dp[0][0]=True

        for j in range(1,len(s2)+1):

            dp[0][j]=dp[0][j-1] and s2[j-1]==s3[j-1]

        for i in range(1,len(s1)+1):

            dp[i][0]=dp[i-1][0] and s1[i-1]==s3[i-1]

        for i in range(1,len(s1)+1):

            for j in range(1,len(s2)+1):

                dp[i][j]=(dp[i-1][j] and s1[i-1]==s3[i+j-1]) or  (dp[i][j-1] and s2[j-1]==s3[i+j-1])

        return dp[len(s1)][len(s2)]

Recursive way

def isInterleave(self, s1: str, s2: str, s3: str) -> bool:

@cache

def dfs(i,j,k):

if k==len(s3):

return i==len(s1) and j==len(s2)

if i<len(s1) and s1[i]==s3[k]:

if dfs(i+1,j,k+1):

return True

if j<len(s2) and s2[j]==s3[k]:

if dfs(i,j+1,k+1):

return True

if len(s1)+len(s2)!=len(s3):

return False

return dfs(0,0,0)

1. Longest Common Subsequence.  
   Iterative way

def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        t1,t2=len(text1),len(text2)

        dp=[[0]\*(t2+1) for \_ in range(t1+1)]

        for i in range(1,t1+1):

            for j in range(1,t2+1):

                if text1[i-1]==text2[j-1]:

                    dp[i][j]=1+dp[i-1][j-1]

                else:

                    dp[i][j]=max(dp[i-1][j],dp[i][j-1])

        return dp[t1][t2]

Recursive way

 def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        @cache

        def re(i,j):

            if i >= len(text1) or j >= len(text2):

                return 0

            if text1[i]==text2[j]:

                return 1+re(i+1,j+1)

            return max(re(i+1,j),re(i,j+1))

        return re(0,0)

1. . Reverse Linked List II

Iterative way

 def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:

        if not head or left==right:

            return head

        dummy=ListNode(0,head)

        prev=dummy

        for \_ in range(left-1):

            prev=prev.next

        curr=prev.next

        for \_ in range (right-left):

            next\_node=curr.next

            curr.next,next\_node.next,prev.next=next\_node.next,prev.next,next\_node

        return dummy.next

Recursive way

def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:

        if not (head and left<right):

            return head

        def help(node,n):

            l,r=left,right

            if n==l:

                prev=None

                curr=node

                while n<=r:

                    curr.next,prev,curr=prev,curr,curr.next

                    n+=1

                node.next=curr

                return prev

            elif n<=l:

                node.next=help(node.next,n+1)

            return node

        return help(head,1)

66.3Sum

Iterative way

def threeSum(self, nums: List[int]) -> List[List[int]]:

        nums.sort()

        s = set()

        for i in range(len(nums)):

            j = i + 1

            k = len(nums) - 1

            while j < k:

                sum = nums[i] + nums[j] + nums[k]

                if sum == 0:

                    s.add((nums[i], nums[j], nums[k]))

                    j += 1

                    k -= 1

                elif sum < 0:

                    j += 1

                else:

                    k -= 1

        return list(s)

Recursive way

def threeSum(self, nums: List[int]) -> List[List[int]]:

        nums.sort()

        result = set()

        def helper(index, target, path):

            if len(path) == 3:

                if sum(path) == target:

                    result.add(tuple(path))

                return

            for i in range(index, len(nums)):

                if i > index and nums[i] == nums[i - 1]:

                    continue

                helper(i + 1, target, path + [nums[i]])

        helper(0, 0, [])

        return [list(triplet) for triplet in result]

7.3Sum Closets

Iterative way

def threeSumClosest(self, nums: List[int], target: int) -> int:

nums.sort()

closest\_sum = float('inf')

for i in range(len(nums)):

left = i + 1

right = len(nums) - 1

while left < right:

current\_sum = nums[i] + nums[left] + nums[right]

if abs(current\_sum - target) < abs(closest\_sum - target):

closest\_sum = current\_sum

if current\_sum < target:

left += 1

elif current\_sum > target:

right -= 1

else:

return target

return closest\_sum

Recursive way

def threeSumClosest(self, nums: List[int], target: int) -> int:

        nums.sort()

        closest\_sum = float('inf')

        def helper(index, left, right):

            nonlocal closest\_sum

            if index >= len(nums):

                return

            while left < right:

                current\_sum = nums[index] + nums[left] + nums[right]

                if abs(current\_sum - target) < abs(closest\_sum - target):

                    closest\_sum = current\_sum

                if current\_sum < target:

                    helper(index, left + 1, right)

                elif current\_sum > target:

                    helper(index, left, right - 1)

                else:

                    return

                break

        for i in range(len(nums)):

            helper(i, i + 1, len(nums) - 1)

        return closest\_sum

8.4Sum

Recursive way

   def fourSum(self, nums: List[int], target: int) -> List[List[int]]:

        nums.sort()

        result = set()

        def helper(index, target, path):

            if len(path) == 4:

                if sum(path) == target:

                    result.add(tuple(path))

                return

            for i in range(index, len(nums)):

                if i > index and nums[i] == nums[i - 1]:

                    continue

                helper(i + 1, target, path + [nums[i]])

        helper(0, target, [])

        return [list(triplet) for triplet in result]

Iterative way

def fourSum(self, nums: List[int], target: int) -> List[List[int]]:

        nums.sort()

        s = set()

        for i in range(len(nums)):

            for j in range (i+1,len(nums)):

                k = j + 1

                l = len(nums) - 1

                while k< l:

                    sum = nums[i] + nums[j] + nums[k]+nums[l]

                    if sum == target:

                        s.add((nums[i], nums[j], nums[k],nums[l]))

                        k += 1

                        l -= 1

                    elif sum < target:

                        k += 1

                    else:

                        l -= 1

        return list(s)

1. Binary Tree Right Side View.

Recursive way

def rightSideView(self, root: Optional[TreeNode]) -> List[int]:

        def dfs(node, level):

            if not node:

                return

            # if this is the first node we're visiting at this level

            if level == len(right\_view):

                right\_view.append(node.val)

            # visit the right node first, then the left node

            dfs(node.right, level + 1)

            dfs(node.left, level + 1)

        right\_view = []

        dfs(root, 0)

        return right\_view

Iterative way

 def rightSideView(self, root: Optional[TreeNode]) -> List[int]:

        if not root:

            return []

        right\_view = []

        queue = deque([root])

        while queue:

            level\_length = len(queue)

            for i in range(level\_length):

                node = queue.popleft()

                # if it's the rightmost element at the level

                if i == level\_length - 1:

                    right\_view.append(node.val)

                # add child nodes in the queue

                if node.left:

                    queue.append(node.left)

                if node.right:

                    queue.append(node.right)

        return right\_view

1. Container With Most Water.

Recursive way

def maxArea(self, height: List[int]) -> int:

        def compute\_area(left, right):

            if left >= right:

                return 0

            width = right - left

            current\_height = min(height[left], height[right])

            current\_area = width \* current\_height

            if height[left] < height[right]:

                return max(current\_area, compute\_area(left + 1, right))

            else:

                return max(current\_area, compute\_area(left, right - 1))

        return compute\_area(0, len(height) - 1)

Iterative way

 def maxArea(self, height: List[int]) -> int:

        max\_area = 0

        left, right = 0, len(height) - 1

        while left < right:

            width = right - left

            current\_height = min(height[left], height[right])

            current\_area = width \* current\_height

            max\_area = max(max\_area, current\_area)

            # Move the pointer pointing to the shorter line

            if height[left] < height[right]:

                left += 1

            else:

                right -= 1

        return max\_area

1. Fair Distribution of Cookies.

Iterative way

def distributeCookies(self, cookies: List[int], k: int) -> int:

        n = len(cookies)

        min\_unfairness = float('inf')

        stack = [(0, [0] \* k)]

        while stack:

            index, children = stack.pop()

            if index == n:

                min\_unfairness = min(min\_unfairness, max(children))

                continue

            for i in range(k):

                new\_children = children[:]

                new\_children[i] += cookies[index]

                stack.append((index + 1, new\_children))

                if children[i] == 0:

                    break

        return min\_unfairness

Recursive way

def distributeCookies(self, cookies: List[int], k: int) -> int:

        def backtrack(index, children):

            if index == len(cookies):

                return max(children)

            min\_unfairness = float('inf')

            for i in range(k):

                children[i] += cookies[index]

                min\_unfairness = min(min\_unfairness, backtrack(index + 1, children))

                children[i] -= cookies[index]

                if children[i] == 0:

                    break

            return min\_unfairness

        return backtrack(0, [0] \* k)

1. Pascal Triangle II.

Iterative way

def getRow(self, rowIndex: int) -> List[int]:

        res = [1]

        prev = 1

        for k in range(1, rowIndex + 1):

            next\_val = prev \* (rowIndex - k + 1) // k

            res.append(next\_val)

            prev = next\_val

        return res

Recursive way

def getRow(self, rowIndex: int) -> List[int]:

        @cache

        def row(index):

            if index == 0:

                return [1]

            previous\_row = row(index - 1)

            current\_row = [1] \* (index + 1)

            for i in range(1, index):

                current\_row[i] = previous\_row[i - 1] + previous\_row[i]

            return current\_row

        return row(rowIndex)

1. Sum of Left Leaves.

Recursive way

 def sumOfLeftLeaves(self, root: Optional[TreeNode]) -> int:

        if not root: return 0

        sum=0

        if root.left:

            if not root.left.left and not root.left.right:

                sum+=root.left.val

            else:

                sum+=self.sumOfLeftLeaves(root.left)

            sum+=self.sumOfLeftLeaves(root.right)

        return sum

Iterative way

def sumOfLeftLeaves(self, root: Optional[TreeNode]) -> int:

        if not root: return 0

        stack = [(root, False)]

        sum\_of\_left\_leaves = 0

        while stack:

            node, is\_left = stack.pop()

            if node:

                if not node.left and not node.right and is\_left:

                    sum\_of\_left\_leaves += node.val

                if node.right:

                    stack.append((node.right, False))

                if node.left:

                    stack.append((node.left, True))

        return sum\_of\_left\_leaves

14. Max Area of Island

dfs

def maxAreaOfIsland(self, grid: List[List[int]]) -> int:

        visit = set()

        def area(r, c):

            if not (0 <= r < len(grid) and 0 <= c < len(grid[0])

                    and (r, c) not in visit and grid[r][c]):

                return 0

            visit.add((r, c))

            return (1 + area(r+1, c) + area(r-1, c) +

                    area(r, c-1) + area(r, c+1))

        maxa=0

        for r in range(len(grid)):

            for c in range(len(grid[0])):

                maxa=max(maxa,area(r,c))

        return maxa

Bfs

def area\_bfs(r, c):

            queue = deque([(r, c)])

            area = 0

            while queue:

                row, col = queue.popleft()

                if (0 <= row < len(grid) and 0 <= col < len(grid[0]) and

                        (row, col) not in visit and grid[row][col]):

                    visit.add((row, col))

                    area += 1

                    # Add neighboring cells to the queue

                    queue.append((row + 1, col))

                    queue.append((row - 1, col))

                    queue.append((row, col + 1))

                    queue.append((row, col - 1))

            return area

        visit = set()

        max\_area = 0

        for r in range(len(grid)):

            for c in range(len(grid[0])):

                if grid[r][c] == 1 and (r, c) not in visit:

                    max\_area = max(max\_area, area\_bfs(r, c))

        return max\_area

15. Jump Game III.

Iterative way

def canReach(self, arr: List[int], start: int) -> bool:

        queue = deque([start])

        visited = set()

        while queue:

            index = queue.popleft()

            if index < 0 or index >= len(arr) or index in visited:

                continue

            if arr[index] == 0:

                return True

            visited.add(index)

            queue.append(index + arr[index])

            queue.append(index - arr[index])

        return False

Recursive way

def canReach(self, arr: List[int], start: int) -> bool:

        def dfs(index):

            if index < 0 or index >= len(arr) or index in visited:

                return False

            if arr[index] == 0:

                return True

            visited.add(index)

            return dfs(index + arr[index]) or dfs(index - arr[index])

        visited = set()

        return dfs(start)

16.Merge Intervals.

Iterative way

def merge(self, intervals: List[List[int]]) -> List[List[int]]:

        if not intervals:

            return []

        # Sort intervals by the start time

        intervals.sort(key=lambda x: x[0])

        merged\_intervals = []

        for interval in intervals:

            if not merged\_intervals or merged\_intervals[-1][1] < interval[0]:

                merged\_intervals.append(interval)

            else:

                merged\_intervals[-1][1] = max(merged\_intervals[-1][1], interval[1])

        return merged\_intervals

Recursive way

def merge(self, intervals: List[List[int]]) -> List[List[int]]:

        if not intervals:

            return []

        # Sort intervals by the start time

        intervals.sort(key=lambda x: x[0])

         def merge\_intervals(intervals):

            if len(intervals) <= 1:

                return intervals

            mid = len(intervals) // 2

            left = merge\_intervals(intervals[:mid])

            right = merge\_intervals(intervals[mid:])

            merged = []

            i = j = 0

            while i < len(left) and j < len(right):

                if left[i][0] <= right[j][0]:

                    interval = left[i]

                    i += 1

                else:

                    interval = right[j]

                    j += 1

                if not merged or merged[-1][1] < interval[0]:

                    merged.append(interval)

                else:

                    merged[-1][1] = max(merged[-1][1], interval[1])

            while i < len(left):

                interval = left[i]

                if not merged or merged[-1][1] < interval[0]:

                    merged.append(interval)

                else:

                    merged[-1][1] = max(merged[-1][1], interval[1])

                i += 1

            while j < len(right):

                interval = right[j]

                if not merged or merged[-1][1] < interval[0]:

                    merged.append(interval)

                else:

                    merged[-1][1] = max(merged[-1][1], interval[1])

                j += 1

            return merged

        return merge\_intervals(intervals)

1. Longest Palindromic Subsequence

Recursive way

def longestPalindromeSubseq(self, s: str) -> int:

        def lps(start, end):

            # Base case: if start and end indices cross each other

            if start > end:

                return 0

            # Base case: if start and end indices point to the same character

            if start == end:

                return 1

            # If characters at start and end indices match, they are part of the LPS

            if s[start] == s[end]:

                return 2 + lps(start + 1, end - 1)

            # If characters don't match, we try both skipping either start or end character

            return max(lps(start + 1, end), lps(start, end - 1))

        return lps(0, len(s) - 1)

Iterative way

def longestPalindromeSubseq(self, s: str) -> int:

        n = len(s)

        # Create a 2D DP table to store the lengths of LPS for substrings

        dp = [[0] \* n for \_ in range(n)]

        # Base case: single characters are palindrome of length 1

        for i in range(n):

            dp[i][i] = 1

        # Iterate over the string in reverse order to fill the DP table

        for start in range(n - 1, -1, -1):

            for end in range(start + 1, n):

                if s[start] == s[end]:

                    # If characters at start and end indices match, add 2 to previous LPS length

                    dp[start][end] = 2 + dp[start + 1][end - 1]

                else:

                    # If characters don't match, take the maximum of skipping either start or end character

                    dp[start][end] = max(dp[start + 1][end], dp[start][end - 1])

        # Return the length of the longest palindromic subsequence

        return dp[0][n - 1]

18-14

19.Surrounded Regions

Dfs  recursive way

def solve(self, board: List[List[str]]) -> None:

        """

        Do not return anything, modify board in-place instead.

        """

        if not board:

            return

        # Get the dimensions of the board

        rows, cols = len(board), len(board[0])

        # Define a helper function to perform DFS

        def dfs(row, col):

            # Base case: If the cell is out of bounds or already visited, return

            if row < 0 or row >= rows or col < 0 or col >= cols or board[row][col] != 'O':

                return

            # Mark the current cell as visited

            board[row][col] = '#'

            # Recursively visit neighboring cells

            dfs(row + 1, col)

            dfs(row - 1, col)

            dfs(row, col + 1)

            dfs(row, col - 1)

        # Start DFS from the edges

        for i in range(rows):

            dfs(i, 0)

            dfs(i, cols - 1)

        for j in range(cols):

            dfs(0, j)

            dfs(rows - 1, j)

        # Replace '#' with 'O' and 'O' with 'X'

        for i in range(rows):

            for j in range(cols):

                if board[i][j] == '#':

                    board[i][j] = 'O'

                else:

                    board[i][j] = 'X'

Iterative way

def solve(self, board: List[List[str]]) -> None:

        """

        Do not return anything, modify board in-place instead.

        """

        if not board:

            return

        # Get the dimensions of the board

        rows, cols = len(board), len(board[0])

        queue = deque()

        # Start BFS from the edges and mark all 'O's and their neighboring 'O's as visited

        for i in range(rows):

            for j in range(cols):

                if (i in {0, rows - 1} or j in {0, cols - 1}) and board[i][j] == 'O':

                    queue.append((i, j))

        while queue:

            row, col = queue.popleft()

            if 0 <= row < rows and 0 <= col < cols and board[row][col] == 'O':

                board[row][col] = '#'

                queue.extend([(row + 1, col), (row - 1, col), (row, col + 1), (row, col - 1)])

        # Replace '#' with 'O' and 'O' with 'X'

        for i in range(rows):

            for j in range(cols):

                if board[i][j] == '#':

                    board[i][j] = 'O'

                else:

                    board[i][j] = 'X'

1. Course Schedule II.

Iterative way

def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:

        # Create adjacency list representation of the graph

        graph = [[] for \_ in range(numCourses)]

        indegree = [0] \* numCourses

        for course, prereq in prerequisites:

            graph[prereq].append(course)

            indegree[course] += 1

        # Initialize a queue for topological ordering

        queue = []

        result = []

        # Add courses with zero indegree to the queue

        for course in range(numCourses):

            if indegree[course] == 0:

                queue.append(course)

        # Perform topological sort using BFS

        while queue:

            course = queue.pop(0)

            result.append(course)

            for neighbor in graph[course]:

                indegree[neighbor] -= 1

                if indegree[neighbor] == 0:

                    queue.append(neighbor)

        # If any courses remain with nonzero indegree, return empty list

        if len(result) != numCourses:

            return []

        return result

Recursive way

 def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:

        # Create adjacency list representation of the graph

        graph = [[] for \_ in range(numCourses)]

        indegree = [0] \* numCourses

        for course, prereq in prerequisites:

            graph[prereq].append(course)

             # Initialize visited and result lists

        visited = [0] \* numCourses

        result = []

        def dfs(course):

            # Mark the course as visited

            visited[course] = 1

            # Visit all prerequisites of the current course

            for prereq in graph[course]:

                if visited[prereq] == 0:  # Not visited

                    if not dfs(prereq):

                        return False

                elif visited[prereq] == 1:  # Visiting (cycle detected)

                    return False

            # Mark the course as completed

            visited[course] = 2

            result.append(course)

            return True

        # Perform DFS on each course

        for course in range(numCourses):

            if visited[course] == 0:  # Not visited

                if not dfs(course):

                    return []

21-11

22. Unique Paths II

Iterative way

def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:

        m, n = len(obstacleGrid), len(obstacleGrid[0])

        # Create a DP table to store the number of unique paths to each cell

        dp = [[0] \* n for \_ in range(m)]

        # Base case: set the number of unique paths for the bottom-right cell

        dp[m - 1][n - 1] = 1 if obstacleGrid[m - 1][n - 1] == 0 else 0

        # Fill the DP table iteratively from bottom-right to top-left

        for i in range(m - 1, -1, -1):

            for j in range(n - 1, -1, -1):

                if obstacleGrid[i][j] == 1:

                    dp[i][j] = 0  # Skip obstacles

                else:

                    if i + 1 < m:

                        dp[i][j] += dp[i + 1][j]  # Count paths from below

                    if j + 1 < n:

                        dp[i][j] += dp[i][j + 1]  # Count paths from right

        return dp[0][0]

Recursive way

 def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:

        m, n = len(obstacleGrid), len(obstacleGrid[0])

        # Create a DP table to store the number of unique paths to each cell

        memo = [[-1] \* n for \_ in range(m)]

        def dfs(x, y):

            # Base case: if the cell is an obstacle, return 0

            if obstacleGrid[x][y] == 1:

                return 0

            # Base case: if the cell is the destination, return 1

            if x == m - 1 and y == n - 1:

                return 1

            # If the number of unique paths to this cell is already calculated, return it

            if memo[x][y] != -1:

                return memo[x][y]

            # Recursive case: sum the number of unique paths from the right and bottom cells

            right\_paths = dfs(x, y + 1) if y + 1 < n else 0

            down\_paths = dfs(x + 1, y) if x + 1 < m else 0

            # Update the memoization table

            memo[x][y] = right\_paths + down\_paths

            return memo[x][y]

        return dfs(0, 0)

1. House Robber II.

Recursive way

  def rob(self, nums: List[int]) -> int:

        if not nums:

            return 0

        n = len(nums)

        if n==1:

            return nums[0]

         # Helper function to calculate the maximum amount of money from a range of houses

        def rob\_range(start, end):

            dp = [0] \* (n + 1)

            dp[start] = nums[start]

            if start + 1 < n:

                dp[start + 1] = max(nums[start], nums[start + 1])

            for i in range(start + 2, end + 1):

                dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])

            return dp[end]

        # Either rob the first house and not the last house, or not rob the first house and rob the last house

        if n == 1:

            return nums[0]

        else:

            return max(rob\_range(0, n - 2), rob\_range(1, n - 1))

Iterative way

def rob(self, nums: List[int]) -> int:

        if not nums:

            return 0

        n = len(nums)

        if n == 1:

            return nums[0]

         # Initialize two variables to store the maximum amount of money from two scenarios

        prev\_max = 0

        curr\_max = 0

        # Calculate the maximum amount of money if robbing the first house and not robbing the last house

        for i in range(n - 1):

            temp = curr\_max

            curr\_max = max(curr\_max, prev\_max + nums[i])

            prev\_max = temp

        max\_first = curr\_max

        # Reset the variables for the next scenario

        prev\_max = 0

        curr\_max = 0

        # Calculate the maximum amount of money if not robbing the first house and robbing the last house

        for i in range(1, n):

            temp = curr\_max

            curr\_max = max(curr\_max, prev\_max + nums[i])

            prev\_max = temp

        max\_last = curr\_max

        # Return the maximum amount of money from the two scenarios

        return max(max\_first, max\_last)

1. Stone Game II
2. Best Time to Buy and Sell Stock with Cooldown.

Iterative way

def maxProfit(self, prices: List[int]) -> int:

        n = len(prices)

        if n <= 1:

            return 0

        hold = [0] \* (n + 1)

        empty = [0] \* (n + 1)

        hold[0] = float('-inf')

        for i in range(1, n + 1):

            hold[i] = max(hold[i - 1], empty[i - 2] - prices[i - 1])

            empty[i] = max(empty[i - 1], hold[i - 1] + prices[i - 1])

        return empty[n]

Recursive way

def maxProfit(self, prices: List[int]) -> int:

        n = len(prices)

        memo = {}

        def dfs(day, hold):

            if day >= n:

                return 0

            if (day, hold) in memo:

                return memo[(day, hold)]

            if hold:

                sell\_today = prices[day] + dfs(day + 2, False)

                not\_sell\_today = dfs(day + 1, True)

                memo[(day, hold)] = max(sell\_today, not\_sell\_today)

            else:

                buy\_today = -prices[day] + dfs(day + 1, True)

                not\_buy\_today = dfs(day + 1, False)

                memo[(day, hold)] = max(buy\_today, not\_buy\_today)

            return memo[(day, hold)]

        return dfs(0, False)