Blockchain-Project Explaination

1. Understanding `exchange.py`

This is your **main Python application** that simulates a blockchain-style **Loyalty Token Wallet System**.

Your project has 4 major components:

🔥 A. Wallet System (Create, Login, Transfer)

Class: Wallet

exchange

This class represents each user's wallet.

Important Attributes

Attribute	Meaning
`owner_name`	Username of wallet owner
`address`	Random 160-bit hex address (like Ethereum)
`balance`	Amount of loyalty tokens (LYC)
`password_hash`	SHA-256 hashed password

Functions inside Wallet

- `to_dict()` → Convert wallet to JSON
- `from_dict()` → Read wallet from JSON file
- `set_password()` → Saves password as hash
- `verify_password()` → Checks login password

🔥 B. WalletManager (Saving, Loading, Authentication)

Class: WalletManager

exchange

Manages all wallets using `wallets.json`.

Functions

1. load_wallets()

Loads all wallets from file.

2. save_wallets()

Writes updated balance & passwords into JSON file.

3. create_wallet()

- Creates new wallet
- Generates address
- Sets 0 LYC balance
- Sets password (hashed)

4. authenticate()

Used for:

- Login
- Confirming transfer security

It checks:

1. Does wallet exist?
2. Does wallet have password?
3. Does user enter correct password (3 attempts)?

5. get_wallet_by_name() / get_wallet_by_address()

Used to find wallets during login & transfer.

🔥 C. VoucherVerifier (Swap voucher → Loyalty Tokens)

Class: VoucherVerifier

exchange

Contains predefined voucher codes like:

`"Startbucks_90_off": ("Starbucks", 50) "Amazon_Festive": ("Amazon", 100)`

Functions

verify_and_redeem(code)

- Checks if voucher is valid
- Checks if it is already redeemed
- Returns brand + token value
- Adds tokens to user wallet

🔥 D. User Session (Menu after login)

Function: user_session()

exchange

After login, user gets 4 options:

1. Swap Voucher

→ Enter voucher code → Add tokens to wallet.

2. Check Wallet Details

→ Shows address, balance.

3. Transfer Tokens

Steps:

1. Enter receiver wallet address
2. Enter amount
3. Ask password again (extra security)
4. Deduct & add balance

4. Logout

→ Exit to main menu.

🔥 E. Main Menu (Starting point)

Function: main()

exchange

Shows:

`1. Login 2. Create wallet 3. Exit`

Handles everything using `WalletManager` and `VoucherVerifier`.

🎤 FACULTY VIVA QUESTIONS + PERFECT ANSWERS

❓ 1. What does your project do?

Ans:
It is a Python-based loyalty token exchange system where users can create a wallet, login, redeem vouchers for LYC tokens, and transfer tokens securely.

❓ 2. How is security handled?

Ans:
We use SHA-256 hashing to store passwords safely.
Passwords are never stored directly.

❓ 3. How are wallets stored?

Ans:
In `wallets.json`.
The data is stored as dictionary:

```
{ "vraj": { "owner_name": "vraj", "address": "0xabc123...", "balance": 100, "password_hash": "hashed value" } }
```

❓ 4. What is voucher swapping?

Ans:
Each voucher code has a brand and value.
When user enters code:

- Verified
- Marked redeemed
- Token value added to wallet

❓ 5. What happens during token transfer?

Ans:

1. User enters receiver address
2. Enter amount
3. Password re-verification
4. Tokens deducted and credited

❓ 6. Why do we re-confirm password during transfer?

Ans:
For additional security to avoid unauthorized transfers.

❓ 7. How is wallet address generated?

Ans:
Using `random.getrandbits(160)`
Converted to hex to look like Ethereum address.

❓ 8. What is purpose of VoucherVerifier?

Ans:
To validate voucher codes and convert them into loyalty tokens.

🚀 1. LoyaltyToken.sol — (Your TOKEN Contract)

This contract is SAME as an ERC20 token.
It creates your blockchain currency **Loyalty Token (LYC)**.

✅ Purpose

- To create a custom token
- To mint tokens when vouchers are redeemed

- To transfer tokens between users
- To keep track of balances

🧩 Key Components

✓ Token Name & Symbol

```
string public name = "LoyaltyToken"; string public symbol = "LYC";
```

Faculty may ask:

> What is the name & ticker of your token?
> **Ans:** LoyaltyToken (LYC)

✓ Total Supply

Stored in:

```
uint256 public totalSupply;
```

Increases when new tokens are minted.

✓ Balance Mapping

```
mapping(address => uint256) public balanceOf;
```

Stores the balance for each user wallet.

✓ Transfer Function

```
function transfer(address to, uint256 amount)
```

- Checks sender balance
- Deducts from sender
- Adds to receiver

Faculty may ask:

> How do you prevent transfer of more tokens than user has?
> **Ans:** require(balance[msg.sender] >= amount)

✓ Mint Function

`function mint(address to, uint256 amount)`

Called by **Exchange.sol** when voucher is redeemed.

🎤 Summary

LoyaltyToken = Creates, stores, transfers, and mints loyalty tokens.

🚀 2. Exchange.sol — (Your SWAP/REDEEM Contract)

This contract handles the **voucher → token exchange**.

✅ Purpose

- To check if voucher is valid
- To call LoyaltyToken.sol and mint tokens
- To prevent same voucher from being used again

🧩 Key Components

✓ VoucherVerifier Contract Instance

`VoucherVerifier verifier;`

Used to check if voucher exists.

✓ Token Contract Instance

`LoyaltyToken token;`

Used to mint tokens to user wallet.

✓ Redeemed Codes Record

`mapping(string => bool) redeemed;`

Prevents:

- Double redeem
- Fraud

✓ Redeem Function

`function redeemVoucher(string memory code)`

Steps:

1. Check if `redeemed[code] == false`
2. Call verifier to get voucher value
3. Mint that value of tokens
4. Mark voucher as redeemed
5. Send tokens to user wallet

Faculty may ask:

> How do you ensure a voucher cannot be reused?
> **Ans:** By storing voucher codes in a `mapping` and marking them true after use.

🎤 Summary

Exchange.sol = Middleman between voucher and token.
It verifies voucher → mints tokens → transfers tokens.

🚀 3. VoucherVerifier.sol — (Your COUPON Contract)

This contract stores all your voucher codes and their values.

✅ Purpose

- Stores predefined voucher codes
- Verifies whether a voucher is valid
- Returns **brand name + token value**

🧩 Key Components

✓ Voucher Storage Structure

Usually:

`struct Voucher { string brand; uint256 value; }`

✓ Voucher Mapping

`mapping(string => Voucher) public vouchers;`

Example entries:

`"Startbucks_90_off" → ("Starbucks", 50) "Amazon_Festive" → ("Amazon", 100)`

✓ Validation Function

`function verify(string memory code)`

Returns:

- brand
- value

Errors:

- invalid voucher
- deleted/expired voucher

🎤 Summary

VoucherVerifier.sol = Stores the voucher database and returns value for Exchange.sol.

🧠 Faculty Viva Questions & Perfect Answers

❓ 1. What is the relationship between the 3 contracts?

Ans:

- VoucherVerifier validates the code
- Exchange uses that validation and mints tokens

- LoyaltyToken actually stores and manages the tokens

❓ 2. Why did you separate contracts?

Ans:
To follow modular smart contract design:

- One for tokens
- One for vouchers
- One for token swapping

❓ 3. Where are tokens minted?

Ans: In **LoyaltyToken.sol** → **mint()**

❓ 4. How do you prevent using the same voucher again?

Ans:
In Exchange.sol we use:

```
mapping(string => bool) redeemed;
```

❓ 5. Is your token centralized or decentralized?

Ans:
It is semi-decentralized because the Exchange contract controls minting.

❓ 6. Why is VoucherVerifier a separate contract?

Ans:
So we can update vouchers independently without changing token logic.

Viva Questions:

🎯 SECTION–1: PYTHON SIDE (exchange.py) VIVA

✅ Basic Concepts

❓ What is the purpose of your project?

It is a blockchain-inspired wallet platform where users:

- create a wallet
- redeem vouchers to earn loyalty tokens
- store tokens securely
- transfer tokens between wallets

✅ Wallet & Security

❓ How are passwords stored?

Using SHA-256 hashing (`hashlib.sha256`).

❓ Why do you not store raw passwords?

To prevent unauthorized access even if file is leaked.

❓ What happens if someone enters wrong password?

You allow 3 attempts → deny access.

❓ How is wallet address generated?

Using:

`random.getrandbits(160)`

→ Converted into 40-digit hex (like Ethereum).

✅ JSON Storage

❓ Where are wallets stored?

Inside **wallets.json** as a dictionary.

❓ What values do you store?

- Owner name
- Address
- Balance
- Password hash

❓ Why JSON?

Lightweight, easy to load/save, human-readable.

✅ Voucher Swapping

❓ How do vouchers work?

Voucher codes stored in a dictionary:

`"Amazon_Festive": ("Amazon", 100)`

When user enters a valid code:

- It gets marked redeemed
- Tokens added to wallet

❓ How do you prevent redeeming twice?

Using:

`self.redeemed = set()`

✅ Token Transfer

❓ How do you transfer tokens?

1. Enter receiver address
2. Verify amount
3. Re-enter password
4. Update both balances
5. Save to JSON

❓ What if sender has insufficient balance?

The transfer is rejected.

❓ Why re-authenticate user during transfer?

Extra security so that any stolen session cannot send tokens.

🎯 SECTION–2: SOLIDITY SIDE VIVA (Smart Contracts)

🚀 LOYALTYTOKEN.SOL (Token Contract)

✓ Basic Questions

❓ What type of contract is LoyaltyToken?

It is an ERC20-like token contract used to store, mint, and transfer loyalty tokens.

❓ What does mint() do?

Creates new tokens and adds them to a user's balance.

❓ Where is total supply stored?

```
uint256 public totalSupply;
```

❓ Who can mint tokens?

Only the **Exchange** contract (usually onlyOwner or authorized entity).

✓ Medium-Level Questions

❓ What data structure stores token balances?

```
mapping(address => uint256) public balanceOf;
```

❓ How do you prevent overflow or invalid transfers?

Using:

```
require(balanceOf[msg.sender] >= amount)
```

❓ Why didn't you use OpenZeppelin ERC20?

To keep contract simple for academic project, custom implementation is enough.

✓ Advanced

❓ Is your token centralized?

Yes, minting is controlled by a contract — not fully decentralized.

❓ Why do we need an ERC20-like structure?

To allow:

- standardized transfers
- compatibility with dApps
- wallets to read balance

🚀 EXCHANGE.SOL (Voucher Redemption Contract)

✓ Basic Questions

❓ What is the purpose of Exchange.sol?

It acts as the **bridge** between voucher codes and token minting.

❓ How does Exchange.sol prevent double redemption?

```
mapping(string => bool) redeemed;
```

❓ What happens in redeemVoucher()?

1. Check voucher is valid
2. Check not redeemed
3. Ask VoucherVerifier for value
4. Mint tokens
5. Mark redeemed

✓ Medium-Level

❓ How does Exchange access LoyaltyToken contract?

Via:

```
LoyaltyToken token;
```

Passed in constructor.

❓ Why separate Exchange and Token contract?

To separate responsibilities:

- Token handles ownership & balances
- Exchange handles business logic

✓ Advanced

❓ What if VoucherVerifier fails?

Transaction reverts.

❓ What security issues are possible?

- Someone could bypass logic if not properly restricted
- Redeeming same code via string manipulation if not normalized

---

🚀 VOUCHERVERIFIER.SOL (Voucher Database)

✓ Basic Questions

❓ What is VoucherVerifier.sol for?

It stores valid voucher codes and returns their value.

❓ What does verify() return?

Brand + Token value.

❓ Can a voucher be changed after deployment?

Not unless contract allows it.

✓ Medium-Level

❓ How are vouchers stored?

Usually in:

`mapping(string => Voucher)`

❓ Why choose struct for vouchers?

To store multiple fields:

- brand
- value

✓ Advanced

❓ Is VoucherVerifier trustless?

No, it is centralized because owner decides vouchers.

❓ Why use string as key?

Because voucher codes are typically alphanumeric.

🎯 SECTION–3: CROSS-CONTRACT VIVA

❓ Explain how all 3 contracts communicate.

1. User enters code → backend calls Exchange
2. Exchange → calls VoucherVerifier.verify()
3. Exchange → calls LoyaltyToken.mint()
4. Tokens are credited to user wallet

❓ Why not merge all 3 into one big contract?

For modularity:

- Token logic
- Voucher checking logic
- Redemption logic
  are independent and replaceable.

❓ What will happen if VoucherVerifier returns invalid code?

Exchange will revert and minting will not happen.

❓ Why do we need a separate VoucherVerifier?

To avoid mixing:

- token logic
- voucher logic

and to update vouchers easily.

🎯 SECTION–4: BLOCKCHAIN CONCEPTS VIVA

❓ What is a smart contract?

A program stored on blockchain that runs automatically when triggered.

❓ What is gas?

The fee required to execute smart contract operations.

❓ What is require() in solidity?

A statement used to enforce rules. If false → transaction reverts.

❓ What is a mapping?

A key-value data structure in Solidity.

❓ What is a constructor?

A special function that runs only once during deployment.

🎯 SECTION–5: SECURITY VIVA

❓ How do you prevent reentrancy?

Your contract does not involve external payable calls, so reentrancy is not possible.

❓ How do you prevent double redemption?

`mapping(string => bool) redeemed` ensures a voucher can only be redeemed once.

❓ How does hashing improve security?

Even if file/database is leaked, attacker cannot reverse hash.

❓ Does your Python wallet store private keys?

No. It is not a blockchain wallet; it’s a simulated wallet system.

🎯 SECTION–6: HIGH-LEVEL PROJECT QUESTIONS

❓ Explain your project in 1 minute.

Your final viva explanation:

**“My project is a Blockchain-based Loyalty Token Exchange System.
We designed a custom token (LYC) using Solidity, created a voucher verification smart contract, and developed an exchange contract to convert voucher codes into tokens.
Using Python, we built a wallet infrastructure where users can create wallets, redeem vouchers, and transfer tokens securely. Passwords are hashed using SHA-256, and all wallet data is stored in JSON. The system integrates both blockchain logic (Solidity) and application logic (Python).”**