



STUDY MATERIAL OF CLASS 9

1. INTRODUCTION TO PYTHON

Python is a beginner-friendly, high-level programming language widely used in data science, web development, and automation. Block-based coding platforms like **Google Colaboratory (Colab)** or **Blockly** make Python even more accessible to beginners.

1.1 Basics

Here's a concise explanation of the **Basics of Python**, covering **syntax**, **indentation**, **comments**, and the **print() function**:

1. Python Syntax

- Python syntax refers to the set of rules that defines how a Python program will be written and interpreted.
- Python uses a **clean and readable** syntax.

```
x = 5
y = 10
print(x + y)
```

2. Indentation (Critical in Python)

- **Indentation** is used to define blocks of code.
- Unlike many other languages (like C, Java, etc.) that use { } to define blocks, Python relies on **whitespace indentation**.
- **4 spaces** is the typical standard (but tabs or any consistent space count is accepted).

```
if x > y:
    print("x is greater")
else:
    print("y is greater or equal")
```

!Incorrect indentation causes errors:

```
if x > y:
print("x is greater")    # ✗ This will raise an IndentationError
```

3. Comments

- Used to explain code or make it more readable.
- Ignored during execution.
- **Single-line comments** start with #.

```
# This is a single-line comment
x = 10  # This comment is after a statement
```

- **Multi-line comments** can be written using triple quotes ''' or """ (commonly used for docstrings too):

```
'''
This is a multi-line comment
used in Python
'''
```

4. print() Function

- Used to **display output** to the screen.

```
print("Hello, World!")    # Prints text
print(3 + 7)              # Prints 10
print("Sum is:", x + y)   # Combines text and variable output
```

Example Code Putting It All Together:

```
# This program adds two numbers
x = 5
y = 10

# Check which number is larger
if x > y:
    print("x is greater")
else:
    print("y is greater or equal")

print("The sum is:", x + y)  # Output the result
```

1.2 Variables, Constants, Keywords

Here's a clear explanation of **variables** and **constants** in Python:

1. Variables in Python

✔ **Definition:** A variable is a name that refers to a value stored in memory. It can change during the program's execution.

✔ **How to Declare:** No need to specify the data type. Just assign a value using =.

```
x = 10      # Integer
name = "John" # String
pi = 3.14   # Float
```

✔ **Variable Naming Rules:**

- Must begin with a **letter** or underscore (_)
- Can contain letters, numbers, and underscores
- **Case-sensitive** (Name and name are different)
- Avoid using Python keywords (like if, while, for)

```
# Valid
_age = 25
total_amount = 100
```

```
# Invalid
# 2name = "Alice" ✗
# if = 10      ✗ (keyword)
```

2. Constants in Python

Definition: A constant is a variable whose value should not change during the execution of a program.

⚠ Python **doesn't have true constants**, but by convention, constants are **written in all UPPERCASE letters**.

✔ **How to Declare:**

```
PI = 3.14159
MAX_USERS = 100
APP_NAME = "MyApp"
```

→ These are not enforced by Python but are understood to be **unchangeable** by developers.

🔗 Summary Table

Feature	Variable	Constant
Can change?	✔ Yes	⊘ Should not change
Syntax	lowercase or camelCase	ALL_UPPERCASE
Example	user_name = "Alice"	PI = 3.14
Enforced?	Yes by Python	No (convention only)

- **Keywords** are reserved words like if, else, for, while, def, which cannot be used as variable names. List of keywords have been provided in 8th class study material (CoreDaoVip Global Curriculum)

1.3 Decision Making in Python

1. if, elif, else
2. Simulated switch (since Python doesn't have a built-in switch like C/C++)
3. Ternary/Conditional operator

1. if-elif-else Statement

Used for **conditional branching**.

✓ **Example:**

```
x = 15
```

```
if x > 20:
    print("x is greater than 20")
elif x == 15:
    print("x is equal to 15")
else:
    print("x is less than 20")
```

2. Simulated switch using dict

Python does **not have a switch-case** statement natively. Instead, we use a **dictionary** with functions or values.

✓ **Example:**

```
def switch_day(day):
    switcher = {
        1: "Monday",
        2: "Tuesday",
        3: "Wednesday",
        4: "Thursday",
        5: "Friday",
        6: "Saturday",
        7: "Sunday"
    }
    return switcher.get(day, "Invalid day")

print(switch_day(3)) # Output: Wednesday
```

3. Ternary Operator (One-line if-else)

✓ **Syntax:**

```
value_if_true if condition else value_if_false
```

✓ **Example:**

```
age = 18
```

```
status = "Adult" if age >= 18 else "Minor"
print(status) # Output: Adult
```

Table 1 Summary Table

Type	Syntax Example	Use Case
if-else	if x > 0: print("Positive")	Multi-branch logic
switch (dict)	switcher.get(choice, "Invalid")	Replace multiple if-elif chains
ternary	"Yes" if x == 1 else "No"	One-line decision

1.4 Loops in Python

This section considers 4 types of looping mechanisms:

1. **for loop**
2. **while loop**
3. **Simulated do-while loop**
4. **Nested loops**

1. for Loop

✓ Used when you know how many times to iterate — often with sequences like lists, strings, or range().

◆ **Syntax:**

```
for variable in sequence:
    # code block
```

✓ **Example:**

```
for i in range(1, 6):
    print("Number:", i)
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

2. while Loop

✓ Used when the condition is checked first, and you don't know how many iterations are needed in advance.

✓ **Example:**

```
count = 1
while count <= 5:
    print("Count is:", count)
    count += 1
```

3. Simulated do-while Loop

Python does **not** have a **built-in do-while loop**, but we can simulate it.

✔ Example:

```
count = 1
while True:
    print("Count is:", count)
    count += 1
    if count > 5:
        break
```

This runs at least **once**, just like do-while in other languages.

4. Nested Loops

✔ A loop inside another loop, useful for working with grids, matrices, patterns, etc.

✔ Example:

```
for i in range(1, 4):          # Outer loop
    for j in range(1, 4):      # Inner loop
        print(f"{i},{j}", end=" ")
    print()                   # New line after each inner loop
```

Output:

```
1,1  1,2  1,3
2,1  2,2  2,3
3,1  3,2  3,3
```

Table 2 Summary Table

Loop Type	Use Case	Condition Check	Guaranteed Run at Least Once	Python Support
for	Iterating over known ranges/lists	Before loop	✗	✔ Yes
while	Repeat while condition is true	Before loop	✗	✔ Yes
do-while	Repeat at least once, then check	After loop	✔ (simulated using while)	✗ (Simulate)
Nested Loops	Patterns, 2D data, combinations	Varies	✗	✔ Yes

1.5 Introduction to Object oriented Programming in Python

What is Object-Oriented Programming?

OOP is a programming paradigm that organizes code into **objects** that contain both **data** (attributes) and **behavior** (methods).

Table 3 Key Concepts of OOP in Python

Concept	Description
Class	A blueprint for creating objects
Object	An instance of a class
Constructor	A special method (<code>__init__</code>) that runs when an object is created
Attributes	Variables that hold data about the object
Methods	Functions defined inside a class
Self	A reference to the current object
Inheritance	One class can inherit properties from another
Encapsulation	Restricts access to certain details (private/public)
Polymorphism	Allows different classes to define methods with the same name but different behavior

1. Class and Object

✓ Example:

```
class Person:
    def __init__(self, name, age):    # Constructor
        self.name = name             # Attribute
        self.age = age

    def greet(self):                  # Method
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Create Object
p1 = Person("Alice", 25)
p1.greet()
```

Output:

Hello, my name is Alice and I am 25 years old.

2. Inheritance

Allows a class to use properties and methods from another class.

```
class Student(Person):              # Student inherits from Person
    def __init__(self, name, age, grade):
        super().__init__(name, age) # Call parent constructor
        self.grade = grade

    def show(self):
        print(f"{self.name} is in grade {self.grade}")

s1 = Student("Bob", 18, "12th")
s1.greet()    # Inherited method
s1.show()
```

3. Encapsulation

You can make variables **private** using `__` (double underscores).

```
class Account:
    def __init__(self, balance):
        self.__balance = balance    # Private attribute

    def show_balance(self):
        print("Balance:", self.__balance)

a1 = Account(1000)
a1.show_balance()
# print(a1.__balance) # ✗ This will raise an error
```

4. Polymorphism

Same method name, different behavior in different classes.

```
class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

# Polymorphism
for animal in (Dog(), Cat()):
    animal.speak()

Output:
Woof!
Meow!
```

Table 4 Summary Table

Feature	Python Syntax Example
Class	<code>class MyClass:</code>
Object	<code>obj = MyClass()</code>
Constructor	<code>def __init__(self):</code>
Attribute	<code>self.name = "value"</code>
Method	<code>def my_method(self):</code>
Inheritance	<code>class Child(Parent):</code>
Encapsulation	<code>self.__private = "secret"</code>
Polymorphism	<code>def speak():</code> (in multiple classes)

1.6 Python Lab

Example 1: Right-Angled Triangle of Stars *

```
rows = 5
for i in range(1, rows + 1):    # Outer loop for rows
    for j in range(1, i + 1):    # Inner loop for stars
        print("*", end=" ")
    print()                     # Newline after each row
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Example 2: Number Pattern

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Example 3: Inverted Triangle Pattern

```
rows = 5
for i in range(rows, 0, -1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

Output:

```
* * * * *
* * * *
* * *
* *
*
```

2. MACHINE LEARNING

2.1 What is Machine Learning (ML)?

Machine Learning is a branch of **Artificial Intelligence (AI)** that enables systems to **learn from data** and improve their performance over time **without being explicitly programmed**.

Instead of hard-coding rules, ML systems **learn patterns** from **past data** to make **predictions or decisions** on new data.

2.2 How Do Apps Learn from Data?

Step-by-step Learning Process:

1. **Collect Data**
E.g., user behavior, sensor data, images, transactions
 2. **Preprocess Data**
Clean, normalize, and prepare it for training
 3. **Train Model**
Feed data to an **ML algorithm** to learn patterns
 4. **Evaluate Model**
Test it on unseen data (called testing or validation set)
 5. **Make Predictions**
Use the trained model to predict future outcomes
 6. **Improve**
Feedback loop helps in retraining with more or better data
-

2.3 Types of Machine Learning Models

2.3.1. Supervised Learning

Model learns from labeled data (input → output)

- **Example Algorithms:**
 - **Linear Regression** (for prediction)
 - **Logistic Regression** (for classification)
 - **Support Vector Machine (SVM)**
 - **Decision Trees**
 - **Random Forest**
 - **K-Nearest Neighbors (KNN)**
- **Use Cases:**
Email spam detection, credit score prediction, fraud detection

2.3.2. Unsupervised Learning

Model learns patterns from **unlabeled data**

- **Example Algorithms:**
 - **K-Means Clustering**
 - **Hierarchical Clustering**
 - **Principal Component Analysis (PCA)**
 - **Use Cases:**
Customer segmentation, market basket analysis, anomaly detection
-

2.3.3. Reinforcement Learning

Agent learns to make decisions through **trial and error** with **rewards and penalties**

- **Example Algorithms:**
 - Q-Learning
 - Deep Q Networks (DQN)
 - Policy Gradient
 - **Use Cases:**
Robotics, game playing (e.g., AlphaGo), self-driving cars
-

2.3.4. Semi-Supervised Learning

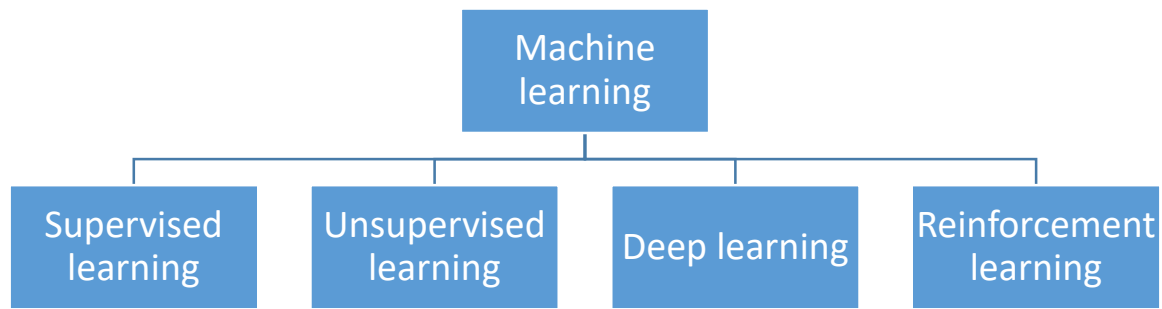
Combination of **small labeled + large unlabeled data**

- Useful when labeled data is expensive/time-consuming to obtain
-

2.3.5. Deep Learning (a subset of ML)

Uses **neural networks** with multiple layers (deep networks)

- **Example Models:**
 - Convolutional Neural Networks (CNN) – for images
 - Recurrent Neural Networks (RNN) – for sequences/time
 - LSTM – long-term memory for text or IoT data
 - Transformers – for language models (e.g., ChatGPT)

**Fig 1** Different type of ML Models**Table 5** Comparison of ML Models

Model	Type	Best For	Pros	Cons
Linear Regression	Supervised	Predicting continuous values	Simple, fast	Not good with complex data
Logistic Regression	Supervised	Binary classification	Interpretable	Works only for linear cases
Decision Tree	Supervised	Classification/regression	Easy to understand	Overfitting on noisy data
Random Forest	Supervised	Classification/regression	High accuracy	Slower, less interpretable
K-Means	Unsupervised	Clustering similar items	Fast, scalable	Need to define number of clusters
SVM	Supervised	Classification of clear margin	Accurate for small datasets	Slow on large datasets
Neural Networks	Deep Learning	Complex tasks like vision/NLP	Powerful, automatic features	Needs lots of data and power
Reinforcement Learning	Reinforcement	Learning in dynamic environments	Adapts over time	Hard to train, needs simulation

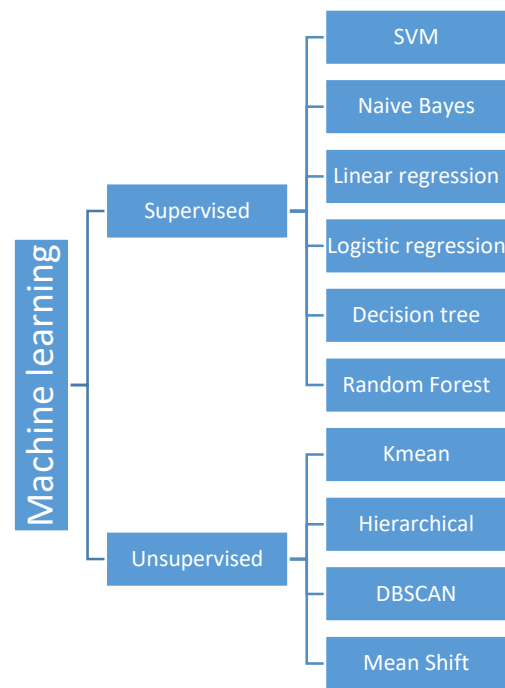


Fig 2 Hierarchical Diagram of Supervised and Unsupervised ML

Categorization of Machine Learning Models with their use case

1. Supervised Learning Models

→ Learn from **labeled data** (input → output)

Table 6 Supervised Learning Models

Model	Use Case	Type
Linear Regression	Predict numeric values	Regression
Logistic Regression	Predict binary classes	Classification
Decision Tree	Interpretable branching model	Both
Random Forest	Ensemble of decision trees	Both
Support Vector Machine (SVM)	Clear class boundaries	Classification
K-Nearest Neighbors (KNN)	Based on distance	Classification
Naive Bayes	Based on probability	Classification
Gradient Boosting (XGBoost, LightGBM)	Ensemble learning	Both

2. Unsupervised Learning Models

→ Learn from **unlabeled data** (no output)

Table 7 Unsupervised Learning Models

Model	Use Case	Type
K-Means Clustering	Group similar items	Clustering
Hierarchical Clustering	Tree-like groupings	Clustering
DBSCAN	Density-based clustering	Clustering
Principal Component Analysis (PCA)	Reduce dimensions	Dim. Reduction
Autoencoders	Neural network for compression	Dim. Reduction
Apriori / FP-Growth	Association rules	Pattern Mining
Isolation Forest	Detect outliers	Anomaly Detection

3. Reinforcement Learning Models

→ Learn by **trial and error** with rewards and penalties

Table 8 Reinforcement Learning Models

Model/Algorithm	Use Case
Q-Learning	Tabular state-action problems
Deep Q Networks (DQN)	Complex game AI
Policy Gradient Methods	Direct action optimization
Actor-Critic Models	Balance learning & action

Used in: Game AI (e.g., AlphaGo), Robotics, Self-driving cars

4. Deep Learning Models (*Subset of ML using Neural Networks*)

Table 9 Deep Learning Models

Model	Use Case
Artificial Neural Network (ANN)	General purpose predictions
Convolutional Neural Network (CNN)	Image, video, vision tasks
Recurrent Neural Network (RNN)	Sequence, time series
LSTM (Long Short-Term Memory)	Memory-based sequence tasks
Transformer (e.g., BERT, GPT)	NLP, translation, chatbots
GANs (Generative Adversarial Networks)	Data generation

Table 10 Summary Table

Learning Type	Model Examples	Best For
Supervised	Linear Regression, SVM, Random Forest	Prediction, classification
Unsupervised	K-Means, PCA, Autoencoders	Grouping, dimensionality reduction
Reinforcement	Q-Learning, DQN	Decision-making over time
Deep Learning	CNN, RNN, Transformer	Images, text, complex data

💡 Final Notes:

- All deep learning is machine learning, but not all machine learning is deep learning.
- Models like **Random Forest** and **XGBoost** are popular for structured data (e.g., CSV files).
- **CNNs**, **RNNs**, and **Transformers** are used when dealing with unstructured data like images, text, and audio.

3.BLOCKCHAIN

3.1 What is Blockchain Structure?

A **blockchain** is a **decentralized, distributed ledger** that stores data (usually transactions) in **linked blocks**. Once added, data is nearly impossible to alter — making it **secure, transparent, and tamper-proof**.

Structure of a Blockchain

Each **block** in a blockchain contains the following:

```

+-----+
| Block Header |
+-----+
| 1. Block Number |
| 2. Timestamp |
| 3. Nonce (random number) |
| 4. Previous Block Hash |
| 5. Current Block Hash |
+-----+
| Transactions (Data) |
+-----+

```

Table 11 Key Components

Part	Purpose
Block Number	Position of the block in the chain
Timestamp	When the block was created
Nonce	Random number used for mining (proof-of-work)
Previous Hash	Links this block to the previous one
Data	Transaction records
Hash	Unique ID of this block (generated by hashing)

Each block is **linked to the previous one** through the **previous hash**, forming a **chain**.

3.2 What is Hashing in Blockchain?

✓ Hashing is a process of converting input (data) into a fixed-length alphanumeric string (called hash or digest) using a hash function.

Most commonly used: **SHA-256** (Secure Hash Algorithm 256-bit)

Properties of Hashing:

1. **Deterministic** – Same input always gives same output
 2. **Irreversible** – Can't derive the input from output
 3. **Fast** – Quickly computes hash for any data
 4. **Collision-resistant** – No two inputs produce same output
 5. **Avalanche effect** – Small change in input → huge change in hash
-

Example of SHA-256 Hash:

```
import hashlib

data = "Hello, Blockchain!"
hash_result = hashlib.sha256(data.encode()).hexdigest()
print("SHA-256 Hash:", hash_result)
```

Output:

SHA-256 Hash: 9a0c... (a 64-character hexadecimal string)

How Blockchain Uses Hashing

► Every block contains:

- Hash of previous block
- Hash of its own data

This chaining ensures **immutability**:

🔒 If someone tries to modify data in any block, its hash changes, breaking the chain, making tampering obvious.

Table 12 Example Chain (simplified):

Block	Data	Prev Hash	Hash (current)
1	Genesis Block	0000	A1B2C3...
2	50 coins	A1B2C3...	F4E5D6...
3	20 coins	F4E5D6...	1A2B3C...

If Block 2 is altered, Block 3's previous hash won't match → blockchain is broken unless recalculated (which is computationally infeasible in real networks).

Table 13 Summary

Concept	Description
Blockchain	A chain of blocks storing data securely
Block	Contains data, previous hash, and current hash
Hashing	Converts data to fixed-size encrypted strings
Hash Function	Ensures data integrity and uniqueness
Immutability	Changes to one block break the whole chain

3.3 Proof of Work/ Proof of Stack

Proof of Work (PoW) and **Proof of Stake (PoS)** are two key **consensus mechanisms** used in blockchain to validate transactions and add new blocks securely:

3.3.1. Proof of Work (PoW)

✓ What it is:

Proof of Work is a **consensus algorithm** that requires miners to **solve complex mathematical puzzles** using computing power. The first one to solve it gets to **add a new block** to the blockchain and receive a **reward**.

🔧 How it works:

1. Miners compete to solve a **cryptographic puzzle** (e.g., find a hash with certain number of leading zeros).
2. The first miner to solve it **broadcasts the solution** to the network.
3. The solution is **verified** by other nodes.
4. A **new block** is added to the chain.
5. The miner gets a **reward** (e.g., Bitcoin).

🔒 Security:

- Very **secure** but **energy-intensive**.
- Makes it **very hard** for anyone to tamper with the blockchain because they would need to redo all the work.

⚙️ Used by:

- **Bitcoin**
- **Litecoin**

✗ Disadvantages:

- Requires huge **computational power**
- **Slow** and **energy-consuming**

3.3.2. Proof of Stake (PoS)

✓ What it is:

Proof of Stake is a **greener alternative** to PoW. Instead of solving puzzles, validators are **randomly chosen** to create a new block based on how much **cryptocurrency they “stake”** (lock up as collateral).

🔑 How it works:

1. Users **stake** their coins (lock them in the network).
2. The network **selects a validator**, often based on a mix of:
 - Amount staked
 - Randomness
 - Coin age
3. The selected validator **validates the block** and adds it to the chain.
4. Validator receives **transaction fees or rewards**.

🔒 Security:

- Encourages honesty: **bad behavior = loss of stake**.
- **Less power-hungry** than PoW.

⚙️ Used by:

- **Ethereum 2.0**
- **Cardano**
- **Polkadot**

✓ Advantages:

- **Energy-efficient**
- **Faster** than PoW
- Environmentally friendly

Table 14 Comparison Table: PoW vs PoS

Feature	Proof of Work (PoW)	Proof of Stake (PoS)
Who validates?	Miners (solve puzzles)	Validators (stake coins)
Energy use	🔌 High (computing power needed)	🌱 Low (eco-friendly)
Speed	🐢 Slower	⚡ Faster
Security	🔒 Very secure	🔒 Secure, but depends on design
Rewards	Block reward + fees	Staking reward or fees
Used by	Bitcoin, Litecoin	Ethereum 2.0, Cardano, Polkadot
Risk of attack	Needs 51% hash power	Needs 51% of staked coins

Table 15 Summary

Concept	Description
PoW	Solve puzzles using CPU/GPU to validate blocks — secure but slow & power-hungry
PoS	Validators are chosen based on staked coins — faster, energy-efficient

4.Exploring CoredaoVIP and 9nftmania?

However it is discussed in study material of class 6, class 7 and class 8. But for further details students should visit official website of 9nftmania and CoredaoVIP.

9nftmania

Visit official site of <https://9nftmania.com>

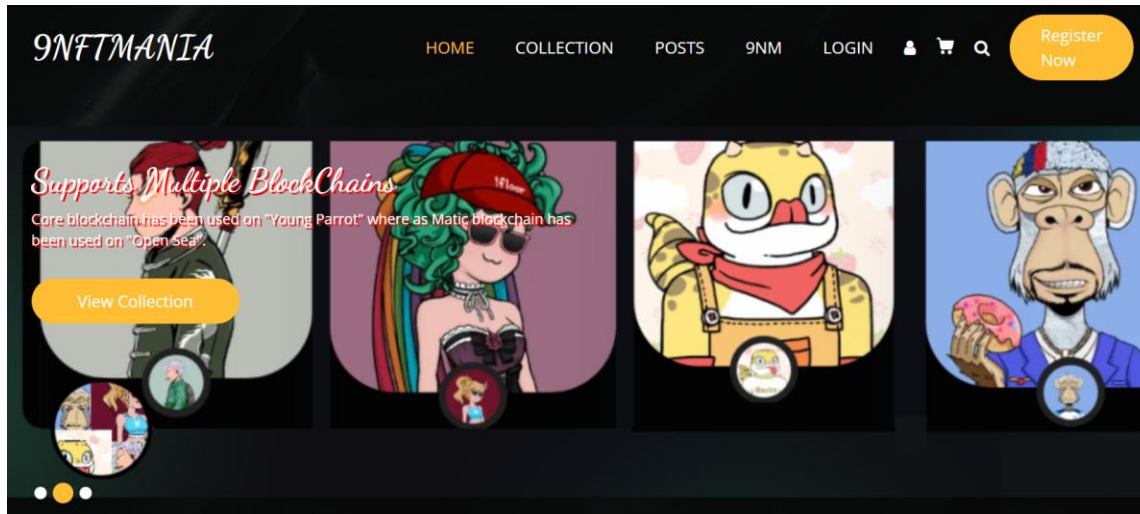


Fig 3 Official website of 9nftmania

CoredaoVIP

Visit official site of <https://coredao.vip>

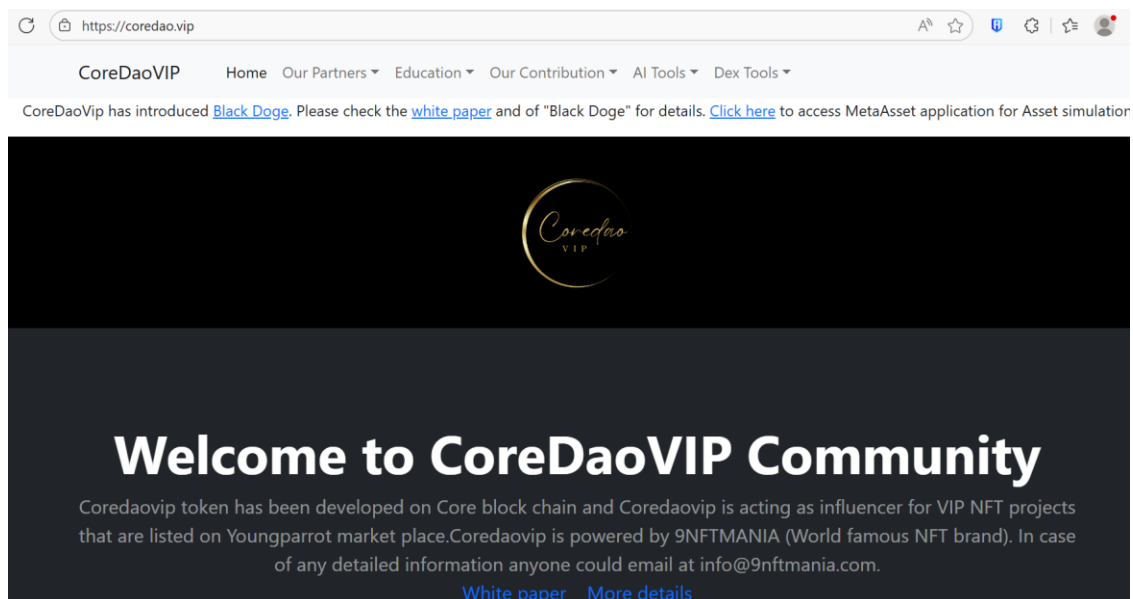


Fig 4 Official website of CoredaoVIP

5.DEX vs CEX

Already discussed in study material of Class 8

6.DEFI vs CEFI

Already discussed in study material of Class 8

7.EXPLORING WEB 3.0**7.1 WEB 3.0**

Web 3.0, also known as the **Semantic Web** or **decentralized web**, represents the next phase in the evolution of the internet, following Web 1.0 (static web) and Web 2.0 (interactive/social web).

7.2 What is Web 3.0?

Web 3.0 aims to create a **smarter, more connected, and decentralized internet**, where users have **greater control over their data and digital identities**, and where machines can understand and process content more meaningfully.

Table 16 Key Characteristics of Web 3.0

Feature	Description
Decentralization	Data is stored across blockchain or distributed networks, reducing reliance on centralized servers.
Semantic Understanding	Machines can interpret the meaning/context of data, enabling smarter search and interaction.
Artificial Intelligence	AI and machine learning are embedded into applications to provide better personalization and automation.
Ubiquity	Web 3.0 applications can be accessed from any device or platform, anywhere.
Trustless and Permissionless	Users can interact without needing a central authority or permissions, thanks to blockchain.
Ownership and Control	Users own their data and digital assets via cryptographic keys and decentralized identifiers (DIDs).

⚙️ Core Technologies Behind Web 3.0

- **Blockchain** (e.g., Ethereum, Polkadot)
- **Cryptocurrencies & Tokens** (e.g., ETH, NFTs)
- **Decentralized Apps (dApps)**
- **Smart Contracts**
- **Decentralized Storage** (e.g., IPFS, Filecoin)
- **Artificial Intelligence & Machine Learning**
- **Semantic Web technologies** (e.g., RDF, OWL)

Table 17 Differences: Web 1.0 vs 2.0 vs 3.0

Feature	Web 1.0	Web 2.0	Web 3.0
Timeframe	1990s – early 2000s	2000s – present	Emerging (2020s onward)
Nature	Static content	Interactive & social	Intelligent, decentralized
Users	Consumers	Content creators	Owners & stakeholders
Control	Centralized	Centralized platforms	Decentralized, peer-to-peer
Monetization	Banner ads	Ad revenue, user data	Tokens, smart contracts

□ Applications of Web 3.0

- **DeFi (Decentralized Finance)** – lending, staking, trading without intermediaries
- **NFTs (Non-Fungible Tokens)** – ownership of digital assets like art or music
- **DAOs (Decentralized Autonomous Organizations)** – community-run governance
- **Decentralized Identity** – self-sovereign identity management
- **Metaverse** – immersive, persistent virtual worlds (integrated with crypto)

△ Challenges

- **Scalability** and transaction speed
- **Regulatory uncertainty**
- **User experience** (still complex for non-tech users)
- **Security and scams** in decentralized ecosystems

★ Summary

Web 3.0 is a vision for a decentralized and intelligent internet where users regain control of their data and digital interactions. It merges technologies like blockchain, AI, and the semantic web to create trustless, permissionless, and user-centric digital ecosystems.

8.Exploring OpenSea

OpenSea is the world's largest decentralized marketplace for NFTs (Non-Fungible Tokens), built primarily on the **Ethereum blockchain** and also supporting **Polygon, Klaytn, and Arbitrum**.

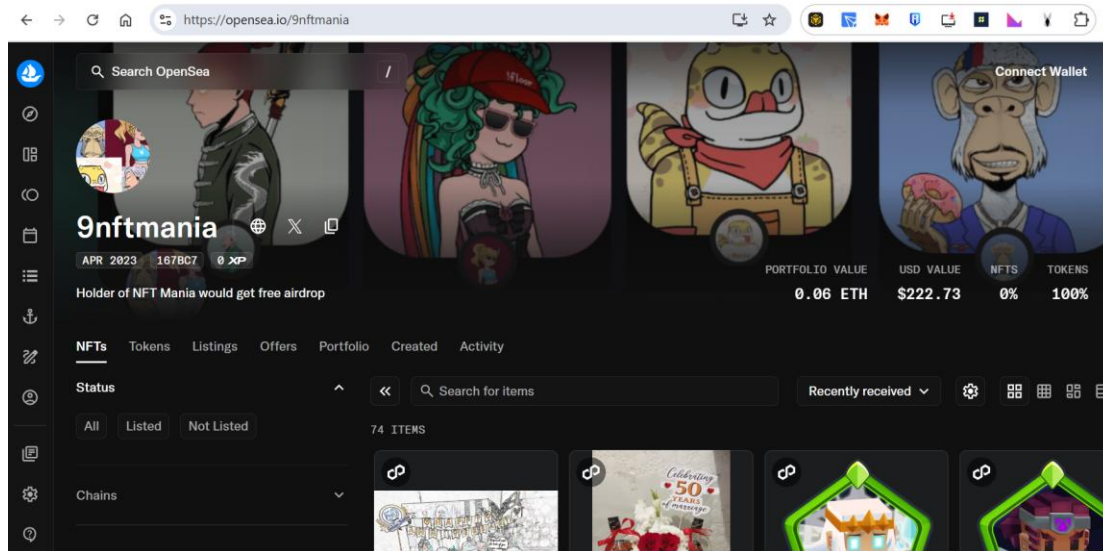


Fig 5 OpenSea NFT Market place

Key Features:

- **Buy, sell, and mint NFTs**
- Supports **ERC-721** and **ERC-1155** token standards
- Wallet integration (e.g., MetaMask)
- Creator royalties and collection management
- On-chain and off-chain metadata indexing

Use Cases:

- Digital art trading
- Collectibles
- Gaming assets
- Virtual real estate

9. 🧑🔬 Researcher Economy (Web3 for Research)

The **Researcher Economy** refers to a decentralized, blockchain-powered ecosystem where:

- Researchers **own their data and IP**
- Contributions are **tokenized and rewarded**
- Peer review is **transparent and traceable**
- **NFTs and DAO-based governance** manage ownership and access

Use of Blockchain:

- Immutable research records
- Funding via tokenized incentives or smart contracts
- Open-access data sharing with incentives
- Authorship validation through token ownership

10. WORKING ON DEX (DECENTRALIZED EXCHANGE)

A **DEX** enables peer-to-peer cryptocurrency trading without intermediaries. Unlike centralized exchanges (CEX), DEXs use **smart contracts** to automate trades and asset custody.

Table 18 List of Different DEXs

DEX Name	Blockchain(s)	AMM Type	Native Token	Key Features
Uniswap	Ethereum, Arbitrum, Polygon	Constant Product ($x*y=k$)	UNI	Pioneering DEX, concentrated liquidity in V3
PancakeSwap	BNB Chain	Constant Product	CAKE	Yield farming, lottery, NFTs
SushiSwap	Multi-chain	Constant Product	SUSHI	Community-driven, supports lending
Curve Finance	Ethereum, Fantom, Arbitrum	StableSwap	CRV	Optimized for stablecoin trading with low slippage
IceCreamSwap	zkSync, Mantle, CoreDAO, etc.	Constant Product	ICE	Multi-chain support, low gas DEX, new-chain integration
LFGSwap	CoreDAO	Constant Product	LFG	CoreDAO-based DEX, liquidity mining and staking
ArcherSwap	CoreDAO	Constant Product	BOW	NFT farming, launchpad features
ShadowSwap	CoreDAO	Constant Product	SHDW	CoreDAO ecosystem, privacy focus, fast UI
Balancer	Ethereum, Polygon	Weighted Pool AMM	BAL	Custom token weight pools
1inch	Multi-chain	Aggregator	1INCH	Finds best prices by aggregating multiple DEXs
QuickSwap	Polygon	Constant Product	QUICK	Fast and affordable Polygon-native DEX
Raydium	Solana	AMM + Order Book	RAY	Integrates Serum's on-chain order book
Trader Joe	Avalanche	Constant Product	JOE	Lending, farming, and NFT marketplace
DODO	Ethereum, BNB Chain	Proactive Market Maker	DODO	Capital-efficient liquidity model
SpookySwap	Fantom	Constant Product	BOO	Fantom-native DEX with cross-chain bridges
SundaeSwap	Cardano	Constant Product	SUNDAE	First DEX on Cardano, decentralized governance
ThorSwap	ThorChain (Cross-chain)	Cross-chain AMM	RUNE	Native cross-chain swaps (no wrapped assets)

Notes on New Additions:

- **IceCreamSwap:** Notable for supporting multiple emerging chains like CoreDAO and zkSync. Offers easy deployment and multi-chain interoperability.
- **LFGSwap:** Focused on the CoreDAO ecosystem. Offers yield farming and token swapping features.
- **ArcherSwap:** Features like NFT farming, launchpad, and staking pools, CoreDAO-based.
- **ShadowSwap:** Lightweight DEX on CoreDAO with a focus on user interface speed and privacy.

DEX LABORATORY

1. Swap

- Enables instant exchange between two tokens
- Uses **Automated Market Maker (AMM)** model
- Example: ETH \rightleftharpoons USDT on Uniswap, Icecreamswap, PancakeSwap, archerswap, lfgswap

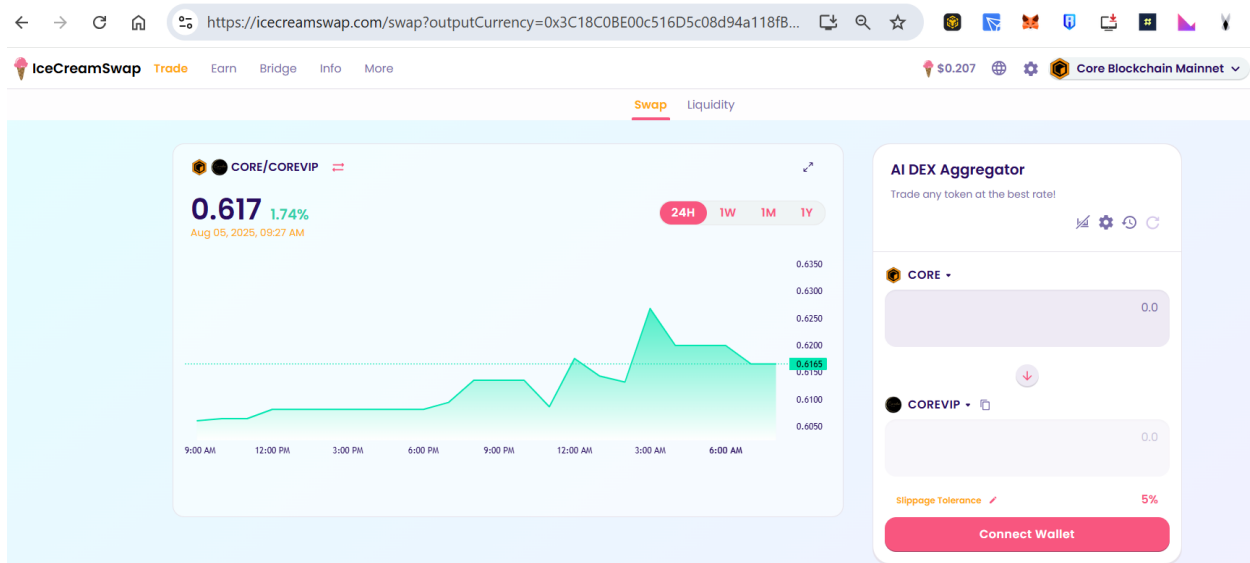


Fig 6 Swapping on icecreamswap

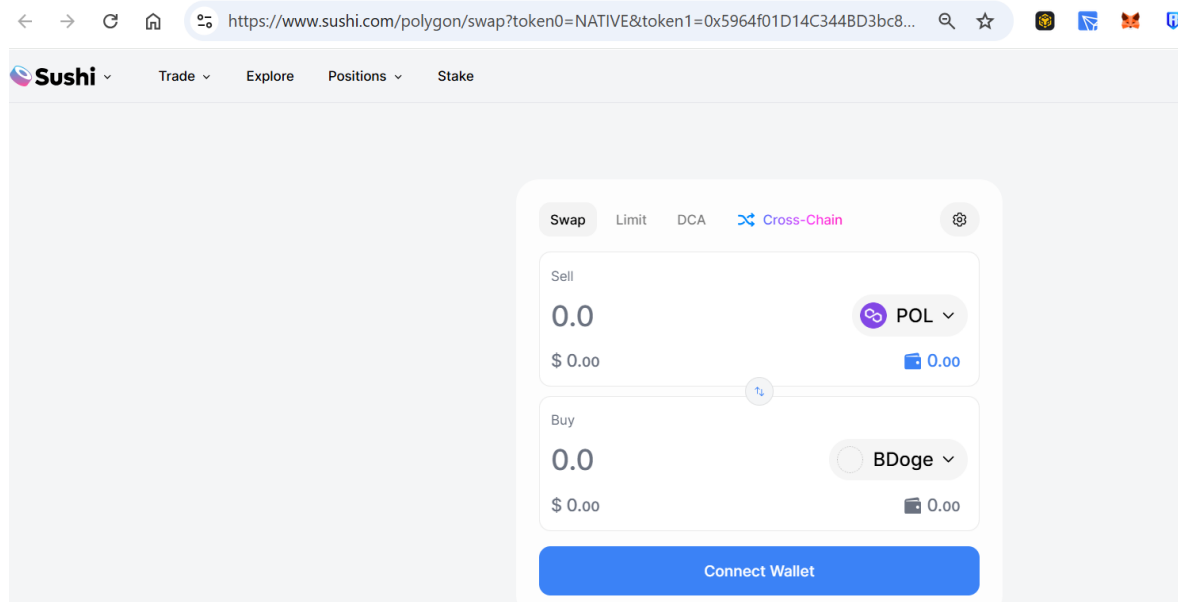


Fig 7 Swapping on sushiswap

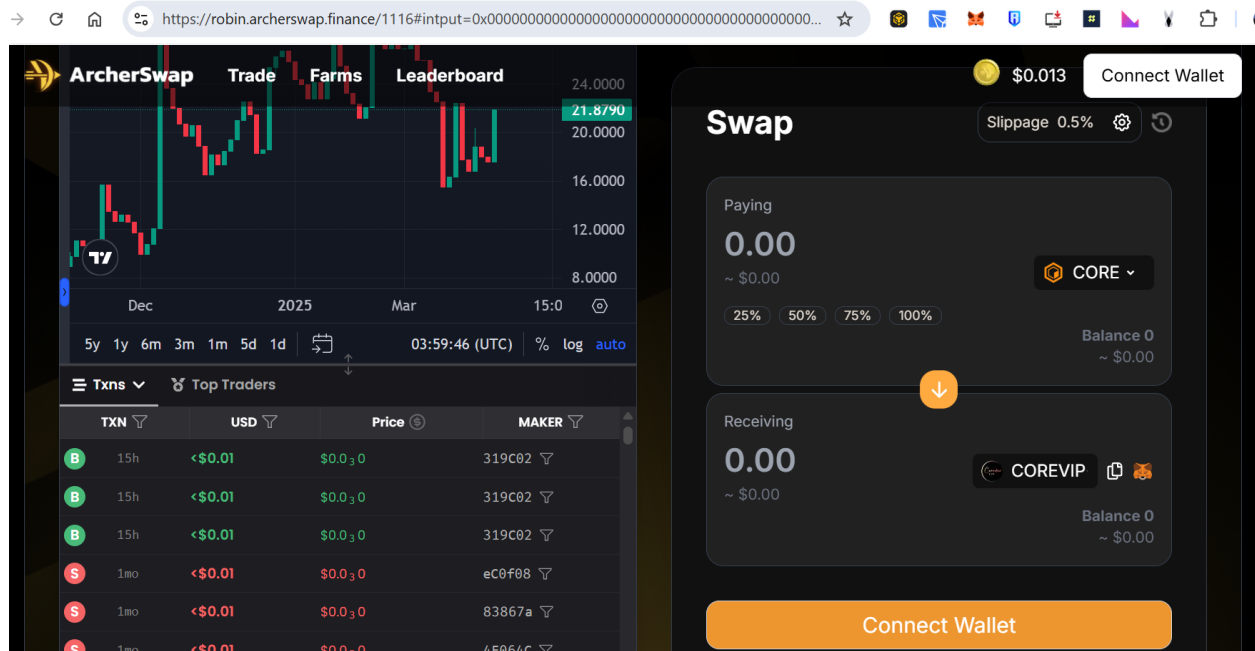


Fig 8 Swapping on Archerswap

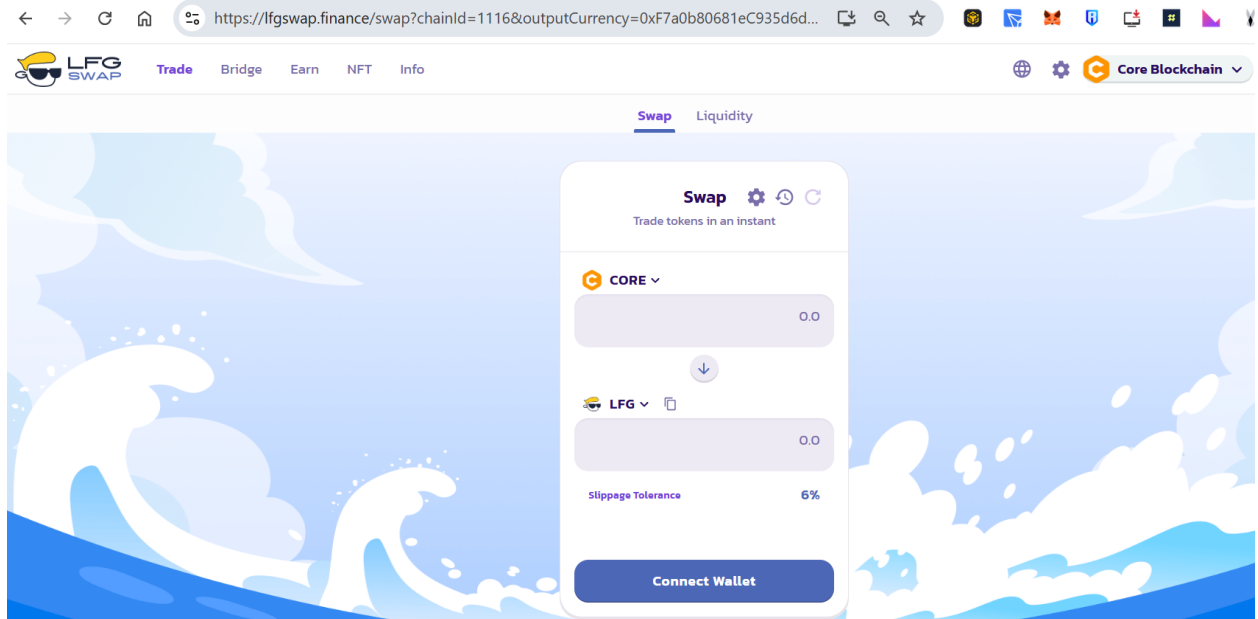


Fig 9 Swapping on lfgswap

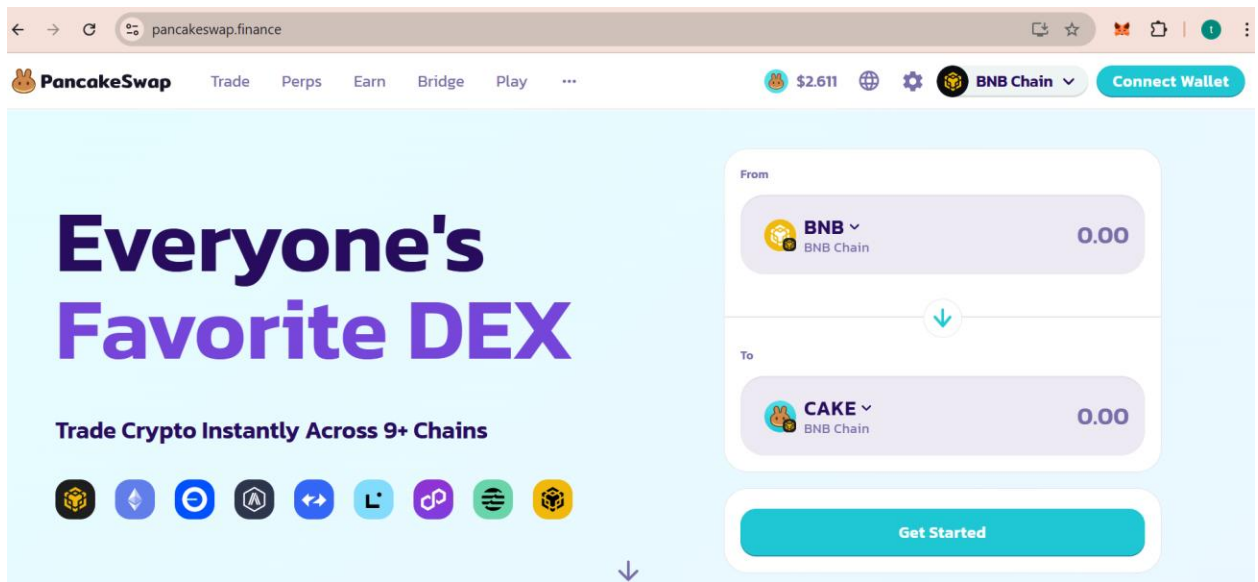


Fig 10 Swapping on pancakeswap

```
// Simplified swap pseudo-logic  
tokenA.transferFrom(user, pool);  
tokenB.transferTo(user);
```

2. Liquidity Pool

- Users deposit token pairs (e.g., ETH/USDT) into smart contracts
- Pool provides liquidity for swaps
- Users earn **LP (Liquidity Provider) tokens** and a share of fees

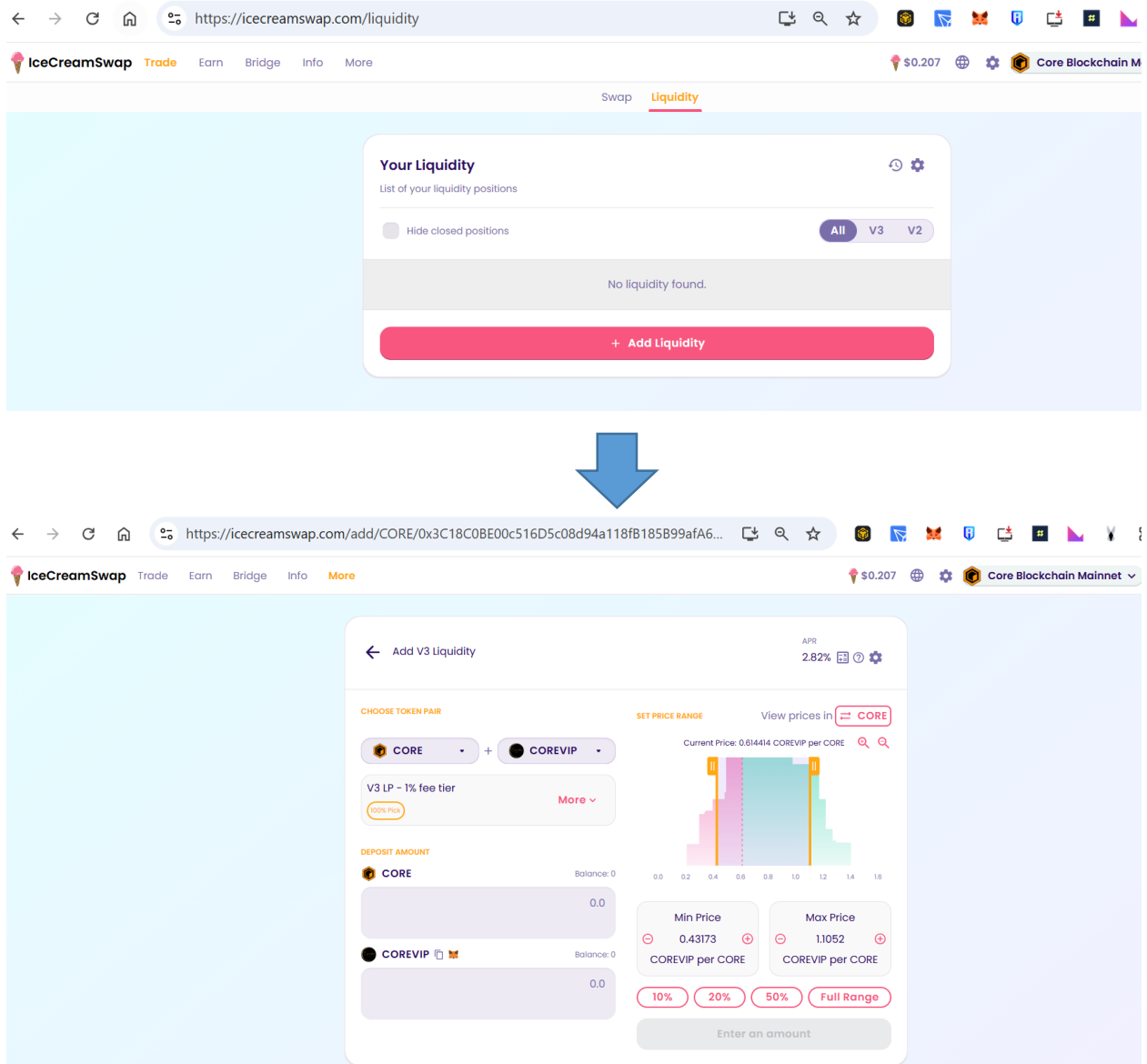
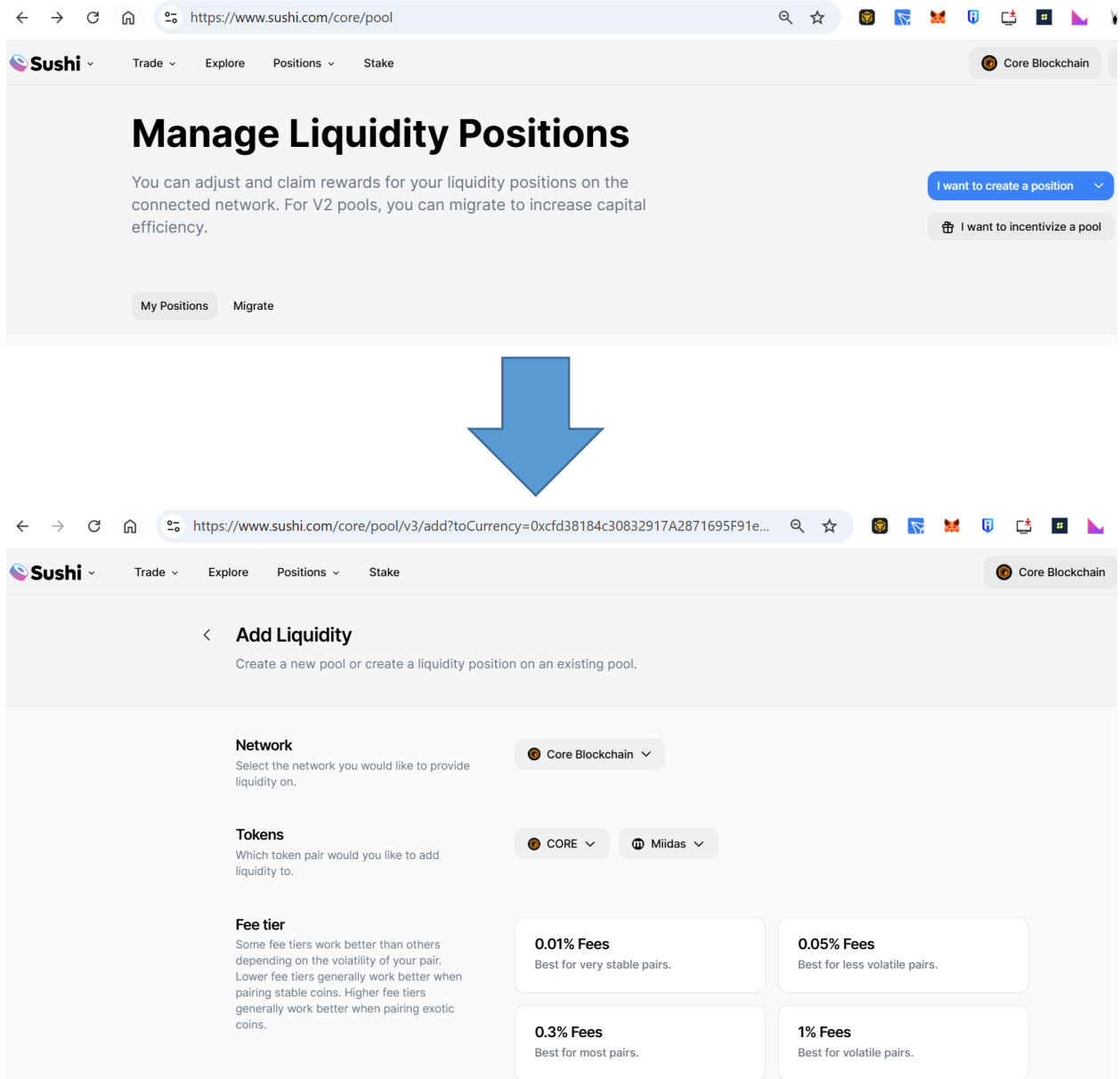


Fig 11 Liquidity pooling in icecreamswap



Manage Liquidity Positions

You can adjust and claim rewards for your liquidity positions on the connected network. For V2 pools, you can migrate to increase capital efficiency.

[I want to create a position](#)

[I want to incentivize a pool](#)

[My Positions](#) [Migrate](#)

Add Liquidity

Create a new pool or create a liquidity position on an existing pool.

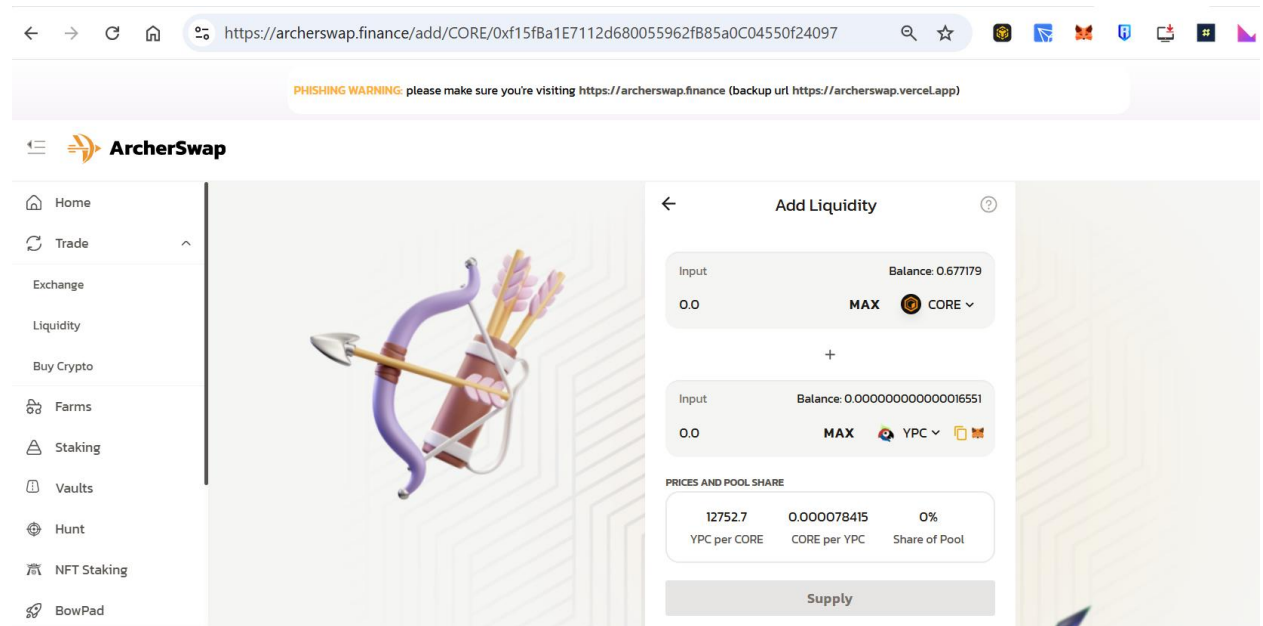
Network
Select the network you would like to provide liquidity on.

Tokens
Which token pair would you like to add liquidity to.

Fee tier
Some fee tiers work better than others depending on the volatility of your pair. Lower fee tiers generally work better when pairing stable coins. Higher fee tiers generally work better when pairing exotic coins.

0.01% Fees Best for very stable pairs.	0.05% Fees Best for less volatile pairs.
0.3% Fees Best for most pairs.	1% Fees Best for volatile pairs.

Fig 12 Liquidity pooling in icecreamswap



29

Example:

- Alice deposits \$500 worth of ETH and \$500 worth of USDT into a pool.
- She earns LP tokens representing her share.
- She receives 0.3% from every swap fee based on her pool share.

3. Locking

- Used to lock LP tokens or native tokens for a fixed period
- Purpose: build **trust**, **stability**, and **prevent rug pulls**
- Locking is usually done using smart contract vaults

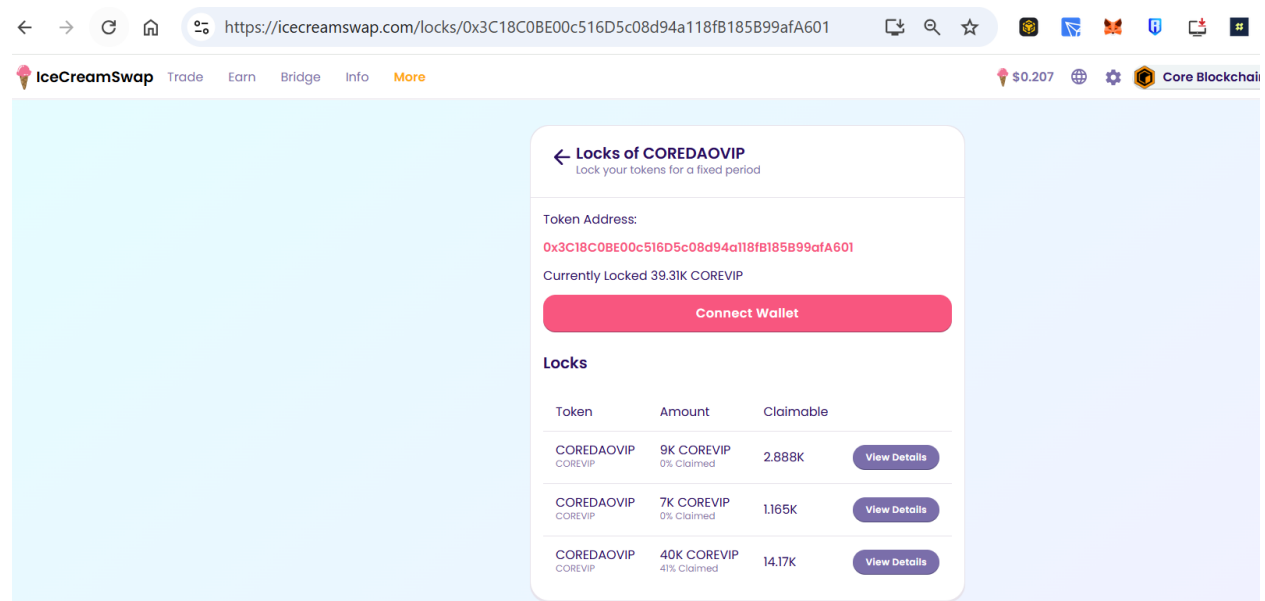


Fig 14 Locking on icecreamswap

Use Cases:

- Team token lockups
- Farming lock-ins
- DAO governance locks

4. Gas Fees / Slippage

Gas fees and **network fees** are **related but not always the same** — their meanings can vary slightly depending on the **blockchain** or **context**.

❑ Gas Fees

- Cost of executing operations (like swaps) on the blockchain
- Paid in native tokens (e.g., ETH for Ethereum)
- Varies based on network congestion and transaction complexity

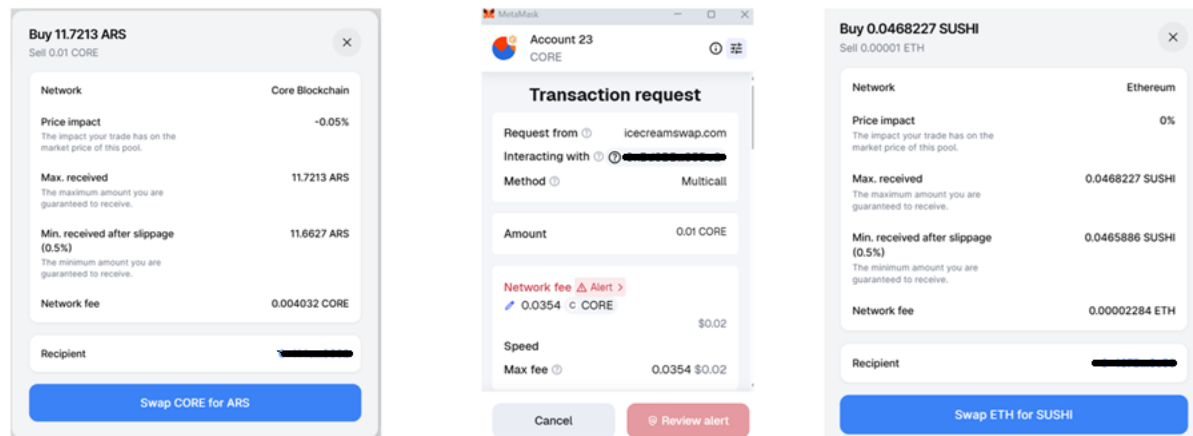


Fig 15 Network Fee Prompts during swapping

Table 19 Detailed Explanation

Term	Definition	Used in	Purpose
Gas Fee	The amount paid to execute a specific operation (like smart contracts, token swaps) on blockchains like Ethereum, BNB, Polygon, etc.	Ethereum, BNB Chain, etc.	Compensates validators for computing power
Network Fee	A broadier term that usually refers to any fee paid for using the blockchain network — includes gas fees but may also include extra costs or flat transaction fees.	Bitcoin, exchanges, wallets	Covers transaction propagation and inclusion in blocks

💡 Examples:

- **On Ethereum:**
 - Sending ETH → Network fee = **gas fee**
 - Executing a DEX swap → Network fee = **gas * gas price**
- **On Bitcoin/Core:**
 - There's no concept of "gas" → You pay a **network fee per byte** of the transaction.
- **On MetaMask or wallets:**
 - *They usually show only one “network fee” to simplify, but it is technically the gas fee.*

Table 20 Key Differences

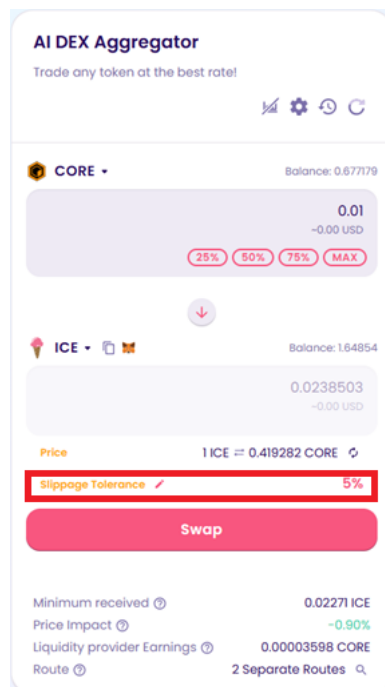
Feature	Gas Fee	Network Fee
Specific to smart contract operations	Yes	Not always
Applies to all blockchains	No (mostly Ethereum-like)	Yes (Bitcoin, Ethereum, etc.)
Can be customized by user	Yes (gas price)	Sometimes (limited in wallets)
Fixed or dynamic	Dynamic (based on demand)	Can be fixed or dynamic

✓ **Conclusion:**

- **Gas Fee** = A type of **Network Fee** used in **smart contract-enabled blockchains**.
- In many contexts, the terms are used interchangeably, but **technically they are not always the same**.

📉 **Slippage**

- Difference between expected price and actual price of a trade
- High slippage can lead to **losses** during volatile markets or low liquidity

**Fig 16** Slippage Prompts during swapping

Example:

- Slippage: 5%

Table 21 Summary Table

Component	Function	Benefit
OpenSea	NFT marketplace	Trade and create NFTs securely
Researcher Economy	Tokenized research & knowledge sharing	Decentralized academic economy
Swap	Token exchange	Fast, peer-to-peer trading
Liquidity Pool	Supply of tokens for swaps	Earn passive income from fees
Locking	Time-lock of tokens or LP	Builds trust and ensures long-term value
Gas Fees	Transaction cost	Needed for blockchain operations
Slippage	Price deviation during trade	Impacts trade accuracy and efficiency