1 D1W2: List

1.1 Introduction to List in Python

In Python, a list is an ordered collection of mutable items with following characteristics:

- Ordered: The elements in a list have a specific order, and this order is maintained. You can access elements using their index.
- Mutable: After creating a list, you can add, remove, modify, or reorder elements within it.
- Collection: A list can hold any type of data (integers, floats, strings, booleans, even other lists or objects) and can contain elements of different data types.

1.2 How to create a List

Using square brackets []: This is the most common way. You can refer to the following basic example about the list.

```
empty_list = []
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed_data = [1, "Python", False, [1, 2, 3]]
```

1.3 Basic List Operations

The ability to be changed (mutability) is a key strength of Lists.

1.3.1 Change Element

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "orange"
print(fruits) # Output: ['apple', 'orange', 'cherry']
```

1.3.2 Add Element

- append(): Add a single element to the end of the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("grape")
print(fruits) # Output: ['apple', 'banana', 'cherry', 'grape']
```

- insert(index, element): Insert an element into a specified position.

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "kiwi")
print(fruits) # Output: ['apple', 'kiwi', 'banana', 'cherry']
```

- extend(iterable): Adds all elements from another iterable to the end of the list.

```
fruits = ["apple", "banana", "cherry"]
more_fruits = ["mango", "pear"]
fruits.extend(more_fruits)
print(fruits)
# Output: ['apple', 'banana', 'cherry', 'mango', 'pear']
```

1.3.3 Remove Element

- remove(value): Removes the first occurrence of a specified value.

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("apple")
print(fruits) # Output: ['banana', 'cherry']
```

- pop(index): Removes and returns the element at a specified position (defaults to the last element).

```
fruits = ["apple", "banana", "cherry"]
removed_fruits = fruits.pop(0)
print(removed_fruits) #Output: ['apple']
print(fruits) # Output: ['banana', 'cherry']
```

- clear(): Removes all elements, making the list empty.

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits) # Output: []
```

1.3.4 Other Operations

- index(value): Returns the index of the first occurrence of a value.

```
my_list = ["a", "b", "c"]
print(my_list.index("b")) # Output: 1
```

- len(list): Returns the number of elements in the list.

```
my_list = [2, 1, 3]
print(len(my_list)) # Output: 3
```

- count(value): Counts the number of occurrences of a specific value.

```
my_list = [2, 1, 3]
print(my_list.count(2)) # Output: 1
```

- sort(): Sorts the elements in the list in ascending order (modifies the list in-place).

```
my_list = [3, 1, 4, 1, 5, 9]
my_list.sort()
print(my_list) # Output: [1, 1, 3, 4, 5, 9]
```

- copy(): Returns a shallow copy of the list

```
original = [1, 2, 3]
copied = original.copy()
print(copied) # Output: [1, 2, 3]
```

1.4 Practice List

You can refer to this Colab link for the list coding with Machine Learning/Deep-learning. Colab url

1.5 Summary

In this article, we covered the following.

- Undertanding about List in Python
- How to create a List and List's characteristics
- Basic List Operations

Lists are one of four built-in data types in Python used to store collections of data, the other is Tuple, Set, Dictionary. We will learn others the next day. With List's characteristics, we can use it with data which need an ordered collection of elements or modify elements (add, remove, change) after creating the collection or store elements of different data types.

2 D2W2: Delving into List

2.1 Overview of Data Structures in Python

- Data structures in Python help in organizing and storing data efficiently.
- They allow data to be accessed and updated efficiently.
- Python has built-in data structures like lists, tuples, sets, and dictionaries.
- There are also user-defined data structures like stack, queue, tree, and graph.

2.2 Types of Lists

2.2.1 1D List

- A 1D list is a simple linear collection of elements.
- Example: `data = [4, 5, 6, 7, 8, 9]`.
- Indexing can be done forward and backward: `data[0] = 4`, `data[-1] = 9`.

2.2.2 2D List

- A 2D list represents a matrix (list of lists).
- Example: `m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.
- Accessing elements: `m[0][0] = 1`, `m[2][1] = 8`.

2.3 Common List Operations

- Concatenation: `data1 + data2`
- Repetition: `data * 3`
- Appending elements: `data.append(4)`
- Inserting elements: `data.insert(0, 4)`
- Extending lists: `data.extend([9, 2])`
- Sorting: `data.sort()`
- Reversing: `data.reverse()`

2.4 Indexing and Slicing

- Forward Indexing: `data[0]` accesses the first element.
- Backward Indexing: `data[-1]` accesses the last element.
- Slicing: `data[start:end]` to access sublists.

2.5 Adding and Deleting Elements

- Add an element: `data.append(4)`
- Insert an element: `data.insert(0, 4)`
- Remove an element: `data.remove(5)`
- Pop an element: `data.pop(2)`
- Clear the list: `data.clear()`

2.6 Sorting and Built-in Functions

- `sort()` - Sort the list in ascending order.
- `sort(reverse=True)` - Sort the list in descending order.
- Built-in Functions:
  - `len(data)` - Returns the number of elements.
  - `min(data)` - Returns the smallest element.
  - `max(data)` - Returns the largest element.
  - `count(value)` - Returns the frequency of an element.
  - `copy()` - Creates a copy of the list.
  - `sum()` - Returns the sum of the elements in the list.

2.7 Built-in Functions

2.7.1 sum()

- `sum(data)` computes the sum of the elements in the list.
- Example: `data = [6, 5, 7, 1, 9, 2]`, `sum(data) = 30`.
- Custom summation can also be implemented with a loop or list comprehension.

2.7.2 zip()

- Combines two lists element-wise.
- Example: `zip([1, 2, 3], [5, 6, 7])` results in pairs: `(1, 5), (2, 6), (3, 7)`.

2.7.3 reversed()

- Returns a reversed iterator of the list.
- Example: `data = [6, 1, 7]`, `reversed(data)` results in `[7, 1, 6]`.

2.7.4 enumerate()

- Returns both the index and value of each element in a list.
- Example: `data = [6, 1, 7]`, `enumerate(data)` gives `(0, 6), (1, 1), (2, 7)`.

2.8 Examples

2.8.1 Sum of Even Numbers

- Define a function to compute the sum of even numbers in a list.
- Example: `data = [6, 5, 7, 1, 9, 2]` computes `sum1(data) = 8`.

2.8.2 Sum of Elements with Even Indices

- Define a function to compute the sum of elements at even indices.
- Example: `data = [6, 5, 7, 1, 9, 2]` computes `sum2(data) = 22`.

2.9 List Comprehension

- A concise way to create lists using a single line of code.
- Example: `[x * x for x in data]` to square each element in the list.
- Can also be used for conditional operations: `[x for x in data if x > 0]`.

2.10 2D List Operations

2.10.1 Creating a 2D List

- Use a list of lists to represent a 2D array.
- Example: `m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.

2.10.2 Hadamard Product (Element-wise Multiplication)

- Perform element-wise multiplication between two 2D matrices.
- Example: `G = [[3, 5], [4, 9]]`, `H = [[1, 6], [3, 2]]` results in `N = [[3, 30], [12, 18]]`.

3 D3W2: Database - SQL

1. Entity Relationship Diagram (ERD)

ERD is a visual tool used to model the logical structure of a relational database.

- **Entities**: Represented as rectangles (e.g., `Customer`, `Product`).
- **Attributes**: Represented as ovals. Key attributes are underlined.
- **Relationships**: Represented as diamonds. Types: 1:1, 1:N, M:N.
- **Weak Entities**: Depend on another entity.
- **Associative Entities**: Represent many-to-many relationships.
- **Crow's Foot Notation**: Visualizes cardinality (e.g., "0 or many", "1 and only 1").

Example ERD Tables:

```
Customer(CustomerID, Name, Birthday, Phone, Address, City, State)
Product(ProductID, Name, Quantity, UnitPrice)
Order(OrderID, CustomerID, OrderDate, Status, Comment, ShippedDate)
OrderDetail(OrderID, ProductID, Quantity)
```

Table 1: Customer Table

CustomerID	**Name**	**Birthday**	**City**
001	Vinh	2005-06-12	Hanoi
002	An	2006-08-19	Da Nang
003	Loc	2007-11-05	Ho Chi Minh

Table 2: Product Table

ProductID	**Name**	**Quantity**	**UnitPrice**
P01	Smartphone	20	200000
P02	TV	10	1000000
P03	Watch	15	500000

Table 3: Order Table

OrderID	**CustomerID**	**OrderDate**	**Status**
O01	001	2025-06-12	Paid
O02	002	2025-06-13	Pending

Table 4: OrderDetail Table

OrderID	**ProductID**	**Quantity**
O01	P01	1
O01	P03	2
O02	P02	1

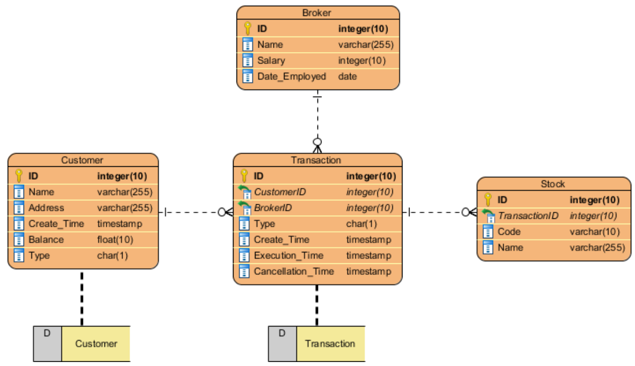

Figure 1: An example of an ERD diagram

2. Database Normalization

Normalization reduces data redundancy and improves data integrity.

First Normal Form (1NF)

- All attributes must contain only atomic values.
- No repeating groups or arrays.

Example Before:

```
StudentID | Name | PhoneNumbers
---------------------------------
101       | Vinh | 19001080, 19001081
```

After 1NF:

```
StudentID | Name | PhoneNumber
------------------------------
101       | Vinh | 19001080
101       | Vinh | 19001081
```

Second Normal Form (2NF)

- Be in 1NF.
- Remove partial dependencies (non-key attribute depends on part of composite key).

Example:

```
StudentProject(StudentID, StudentName, ProjectID, ProjectName)
-- Becomes:
Student(StudentID, StudentName)
Project(ProjectID, ProjectName)
Student_Project(StudentID, ProjectID)
```

Third Normal Form (3NF)

- Be in 2NF.
- Remove transitive dependencies (non-key depends on another non-key).

Example:

```
Student(StudentID, StudentName, Zipcode, City)
-- Becomes:
Student(StudentID, StudentName, Zipcode)
Zipcode(Zipcode, City)
```

Boyce-Codd Normal Form (BCNF)

- Every determinant must be a super key.

Example:

```
Instructor(InstructorID, InstructorName, CourseID)
-- InstructorID -> CourseID is a problem if InstructorID is not a super key
-- Decompose to:
Instructor(InstructorID, InstructorName)
Course(CourseID, InstructorID)
```

Fourth Normal Form (4NF)

- Be in BCNF.
- Remove multi-valued dependencies.

Example:

```
Model(ModelID, Color, Factory)
-- Decompose to:
Model_Color(ModelID, Color)
Model_Factory(ModelID, Factory)
```

Fifth Normal Form (5NF)

- Be in 4NF.
- No join dependency anomalies.

Example:

```
Dealer_Product_Supplier(Dealer, Product, Supplier)
-- Decompose to:
Dealer_Product(Dealer, Product)
Product_Supplier(Product, Supplier)
Dealer_Supplier(Dealer, Supplier)
```

3. SQL Implementation Example

```sql
CREATE TABLE Customer (
  CustomerID INT PRIMARY KEY,
  Name VARCHAR(100),
  Birthday DATE,
  Phone VARCHAR(15),
  Address VARCHAR(100),
  City VARCHAR(50),
  State VARCHAR(50)
);
```

```sql
CREATE TABLE Product (
   ProductID INT PRIMARY KEY,
   Name VARCHAR(100),
   Quantity INT,
   UnitPrice DECIMAL(10,2)
);

CREATE TABLE "Order" (
   OrderID INT PRIMARY KEY,
   CustomerID INT,
   OrderDate DATE,
   Status VARCHAR(50),
  FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);

CREATE TABLE OrderDetail (
   OrderID INT,
   ProductID INT,
   Quantity INT,
  PRIMARY KEY (OrderID, ProductID),
  FOREIGN KEY (OrderID) REFERENCES "Order"(OrderID),
  FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
```

4. Conclusion

- Entity Relationship Diagrams (ERDs) are essential tools in database design, allowing for clear visualization of entities, attributes, and relationships within a system.
- Proper use of ERD notation, such as Chen's or Crow's Foot, ensures logical structure and reduces ambiguity during the design phase.
- Normalization is critical for reducing data redundancy and avoiding anomalies. Through forms like 1NF, 2NF, 3NF, and beyond, it ensures each table has a clear and consistent structure.
- Normal forms help in organizing data dependencies and in decomposing complex tables into smaller, more manageable ones without information loss.
- Implementing a normalized ERD using SQL helps maintain data integrity, enforces relational constraints, and supports efficient query execution.
- Together, ERD modeling, normalization theory, and SQL implementation form the core practices of building robust, scalable, and maintainable relational databases.

4 D4W2: Advanced Data Structure (IoU, NMS, and Histogram)

Data Structure: a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently

Immutable: Cannot change the value of a variable (String, Tuples)

Mutable : Can change the value of a variable (List, Dictionaries, Set)

4.1 Tuple

Tuple are another kind of sequence that functions much like a list, but immutable

For example: x = ('Glen', 'Sally', 'Joseph'), y = (1 2 3)

4.1.1 Structure

tuple-name = (element-1, , element-n)

4.1.2 Method

count() : count the time which value appear
index() : find the position of value
len() : find length of a tuple
sorted() : sort min -¿ max
Other: swap, sys.getsizeof, type, list2tuple, tuple2list, use to protect data

4.2 Set

Set : cannot have multiple occurences of the same element and store unordered values

4.2.1 Structure

Using curly bracket, can contains different types of data Example: "cat", "dog", "anime, 1, "cat", True, 40.0

4.2.2 Method

Access: for ... in set-name: print()
Copy: .copy()
Join: .union()
Insert: .update()
Other: Bitwise, remove, create

4.3 Dictionary

Like lists but use keys instead of numbers to look up value

4.3.1 Structure

Example: $'learning-rate':0.1,'metric':'accuracy$

4.3.2 Method

Create, Update, Copy
Note: Copy: shallow copy or use deepcopy()
Other method:
popitem() - return last value in dictionary
clear() - remove all iemts dictionary
Use del keyword to delete an item

5 D5W2: Git and Github for Version Control

VCS: Version Control System: The system tracks and records changes to files over time. A VCS (Version Control System) helps log who made the changes, what was changed, and when it happened
Type: Centralized VCS (CVCS) and Distributed VCS (DVCS)

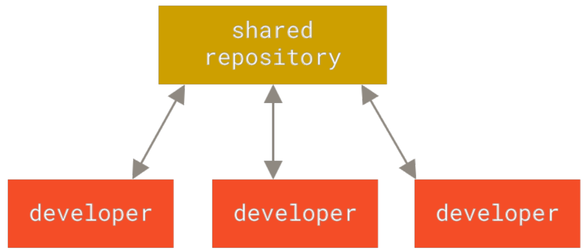

Figure 2: Centralized

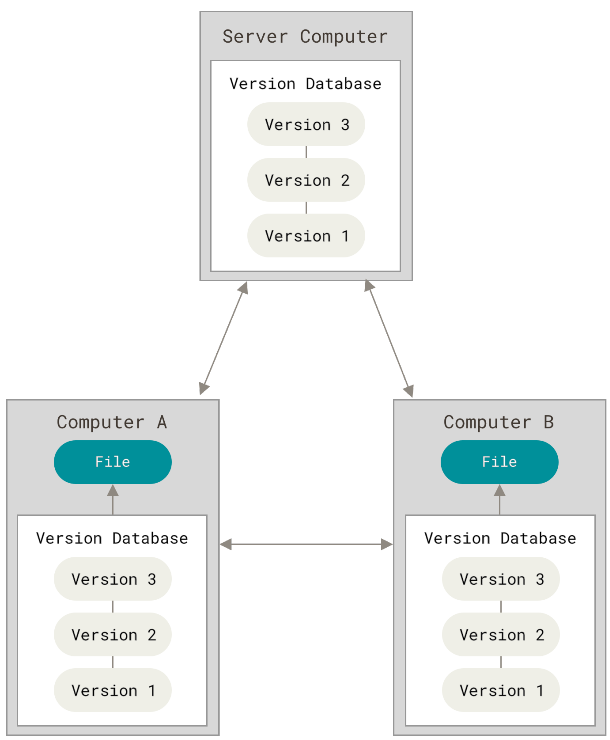

Figure 3: Distributed

5.1 The Working Principles of Git

- Snapshot
  - SHA-1
  - Offline-first

5.2 Method

git clone: copy repository from Github to computer
git status: check status
4 main status: untracked, modified, staged, commited
git log: display commit history
git stash: save change
git stash: apply changes which be saved
git branch ... : work with branch
Rebase: Create clean commit history