Python: (programming language)

- **Python is a high-level, interpreted programming language known for its simplicity and readability.**
- **Widely used for web development, data analysis, artificial intelligence, and more.**
- **Known for its extensive standard library and a large ecosystem of third-party packages.**
- **Supports multiple programming paradigms, including procedural, object-oriented, and functional programming.**
- **Has a strong and active community of developers contributing to its growth and evolution.**

Fundamental concepts of Python:

• Syntax :

Python syntax is clean, simple, and easy to read, emphasizing readability and straightforward code structure.Syntax refers to the set of rules ,In Python syntax governs the arrangement of elements such as keywords, identifiers, operators, and punctuation marks to create valid code.

Here are some key aspects of Python syntax:

1. Indentation : Python uses indentation to **define blocks of code**, such as loops, conditional statements, and function definitions. typically using four spaces per indentation level.

```
Ex:  if x < 5:
        print("x is less than 5")
     else:
        print("x is greater than or equal to 5")
```

2. Statements and Expressions : Python code consists of statements and expressions. **Statements are instructions that perform actions**, such as variable assignment or control flow statements. Expressions produce values and can be **combined with operators to form more complex expressions.**

```
Ex:  x = 5 // Statement
     y = x + 3// Expression
```

3. Keywords and Identifiers : Keywords **are reserved words** in Python that have special meanings and cannot be used as identifiers (variable names, function names, etc.). Identifiers are user-defined names used to identify variables, functions, classes, and other objects in Python code.

Ex : if, else, def, while, for, in, True, False, None // Keywords

variable_name, function_name, class_name// Identifiers

4. Comments : Comments in Python start with the `#` symbol and are **used to add explanatory notes** or documentation within the code. Comments are ignored by the Python interpreter and are for human readers.

Ex: # This is a comment

x = 10 # This is also a comment

5. Line Continuation : Python allows the use of the backslash (`\`) character to continue a statement across multiple lines for better readability. Alternatively, parentheses `()` or brackets `[]` can also be used for line continuation inside parentheses or brackets.

Ex: # Line continuation using backslash

**total = 1 + **

**2 + **

3

- **Data Types** : Python represent the kind of values that **can be stored** and **manipulated by the program, .** Each data type defines a set of values **,**Python supports various built-in data types such **as int**egers, **float**s, **str**ings, **list**s, **tuple**s, **dict**ionaries, and **set**s.

1. Integer (int) : Represents **whole numbers**, both positive and negative, without

any decimal point**. Ex :** `5`, `-10`, `1000`

2. Float (float): Represents **decimal numbers**, also known as **floating-point numbers**.

Ex : `3.14`, `-0.001`, `2.718`

3. String (str): Represents a **sequence of characters** enclosed within single quotes (`'`)

or double quotes (`"`)**.Ex**: `'Hello'`, `"Python"`, `'123'`

4. Boolean (bool) : Represents a **binary value** indicating either **true or false**. Used in

conditional expressions and logical operations .**Ex**: `True`, `False`

5. List : Represents **an ordered collection** of elements enclosed within **square brackets** (`[]`). Lists can contain elements of different **data types. Ex:** `[1, 2, 3, 'hello', True]`

6. Tuple: Similar to lists, but tuples are **immutable,** meaning **their elements cannot** be

changed after creation.Tuples are defined **using parentheses** (`()`).**Ex**: `(1, 2, 3, 'hello')`

7. Dictionary (dict) : Represents a collection **of key-value pairs** enclosed within **curly**

braces (`{}`). Each key-value pair maps **a unique key** to its corresponding value.

Ex: `{**'name': 'John'**, *'age': 30*, **'city': 'New York'**}`

8. Set : Represents an **unordered collection** of unique elements. Sets are defined using

curly braces (`{}`) or the **`set()`** constructor. Ex: `{1, 2, 3, 4}`, `{'apple', 'banana', 'orange'}`

- **Control Structures :** Control structures like loops (for, while) and conditional statements (if, elif, else) govern the flow of execution in Python programs.Control structures in Python are mechanisms that **control the flow of execution within a program.** They allow you to dictate which parts of the code should be executed based on certain conditions or how many times a certain block of code should be repeated.

1. Conditional Statements: Conditional statements, such as `if`, `elif`, and `else`, enable you to execute specific blocks of code based on whether certain conditions are true or false.

- The **`if`** statement checks a condition and executes the block of code indented beneath it if the condition evaluates to true.
- The `**elif**` (else if) statement allows you to check additional conditions if the preceding `if` condition(s) are false.
- The **`else`** statement provides a fallback option if none of the previous conditions are met.

2. Loops: Loops, including `for` and `while` loops, allow you to execute a block of code repeatedly.

- A **`for`** loop iterates over a sequence of items (e.g., a list, tuple, or string) and executes the block of code once for each item in the sequence.
- A `**while**` loop repeatedly executes a block of code as long as a specified condition remains true. The loop continues until the condition evaluates to false.

- **Functions :** Functions are blocks of reusable code that perform specific tasks. They help in organizing and modularizing code for better maintainability.

Functions in programming are like mini-programs within a larger program. They are blocks of code that perform a specific task, and they can be reused multiple times throughout the program. Here's an explanation of how functions work and why they are essential:

1. Modularity : Functions help in **organizing code into smaller**, manageable chunks. Instead of writing all the code for a particular task in one place, you **can break it down into smaller functions**, each responsible for a **specific aspect of the task.**

- This modularity makes the code **easier to understand**, **maintain, and debug**. It also promotes code reusability since you can call the same function from different parts of the program whenever you need to perform the task it's designed for.

2. Abstraction : - Functions allow you to abstract away implementation details and focus on **the high-level logic of your program**. When you call a function, you don't need to know how it's implemented; you only need to know what it does and how to use it.

-This abstraction makes the code **more readable** and **less error-prone** since you can hide complex implementation details behind simple, descriptive function names.

3. Encapsulation: - Functions encapsulate a piece of functionality, meaning they **bundle together related code into a single unit**. This encapsulation helps in managing complexity by breaking down **a large program into smaller,** more manageable parts.

- Each function can have its own **local variables and parameters**, which are independent of variables in other functions. This prevents unintended side effects and makes it easier to reason about the behavior of **each function in isolation**.

4. Reusability: - Functions promote code reusability by allowing you to **write code once and reuse it multiple times throughout your program**. Once you define a function, you can call it from any part of your program, passing different arguments to customize its behavior as needed.

- This reusability not only saves time and effort but also helps in maintaining consistency and avoiding duplication of code.

- **Modules and Packages :** Python modules are files containing Python code, while **packages are directories containing multiple modules**. **In Python, modules and packages are fundamental concepts** for **organizing, reusing, and distributing code.**

1. Modules: Modules in Python are individual files containing Python code. Each **module can define variables, functions, and classes** that can be used in other Python files. Modules allow you to organize related code into **separate files, making your codebase more modular and easier to manage.**

To use code from a module in your program, you need to import the module using **the `import` statement.** Once imported, you can access the variables, functions, and classes defined in the module using **dot notation** (`module_name.item`).

Ex : def add(a, b)://math_oprations.py

return a + b

def subtract(a, b):

return a - b

Ex : import math_operations // main.py

result = math_operations.add(5, 3)

print(result) # Output: 8

2. Packages: Packages in Python are directories containing multiple Python modules. Each package directory contains an **`__init__.py`** file (which can be empty) to indicate that the directory should be treated as a package.**Packages allow you to organize related modules into hierarchical namespaces,** providing a way to structure large codebases into smaller, more manageable units. To import modules from a package, you can use dot notation **(`package_name.module_name.item`).**

- **- Ex: directory structure:**

my_package/

├── __init__.py

├── module1.py

└── module2.py

Ex: def greet()://my_package/module1.py

print("Hello from module1")

Ex : def farewell():

print("Goodbye from module2")

Ex: # main.py

import my_package.module1

import my_package.module2

my_package.module1.greet() # Output: Hello from module1

my_package.module2.farewell() # Output: Goodbye from module2

My-SQL:

- MySQL is an **open-source** relational database management system **(RDBMS).**
- It uses structured query language (SQL) for **managing and manipulating data.**
- Provides features like **ACID** (Atomicity, Consistency, Isolation, Durability) **properties for transaction management.**
- Supports **scalable and reliable data storage** for various applications.
- Offers features such **as indexing, replication, and security mechanism**s for data management.

Fundamental Basics of MySQL:

1. Database :

- MySQL is a relational database management system (**RDBMS**) that **stores data in tables.**

- A database in MySQL is a collection of related tables **that hold information**.

2. Tables :

- Tables are the basic units of **storage in MySQL**.

- Each table consists of **rows and columns**.

- Columns define the attributes or fields of the data, while rows represent individual records.

- Tables are defined with a **CREATE TABLE** statement and can be modified with **ALTER TABLE** statements.

3. Data Types:

- MySQL supports various data types to represent different types of **data, including integers,**

  floating-point numbers, strings, dates, and more.

- Data types determine the kind of data that can be stored in each **column of a table.**

4. Queries :

- MySQL uses Structured Query Language (**SQL**) for **interacting with the database.**

- SQL queries are used to perform operations such as retrieving data (**SELECT**), adding data (**INSERT**), updating data (**UPDATE**), and deleting data (**DELETE**) from tables.

5. Primary Keys and Foreign Keys:

- Primary keys are columns or combinations of columns **that uniquely identify each row in a table**. They ensure data integrity and enforce entity integrity.

- Foreign keys establish relationships between tables by referencing **the primary key of another table.** They enforce referential integrity, ensuring that rows in related tables remain consistent.

6. Indexes :

- Indexes are data structures that improve the **speed of data retrieval** operations on tables.

- They are created on columns to allow for **faster data retrieval** by providing **quick access to rows based on the indexed columns.**

7. Normalization :

- Normalization is the process of **organizing data in a database efficiently.**

- It involves dividing large tables into smaller, more manageable tables and defining relationships between them to reduce redundancy and dependency.

8. Transactions :

- Transactions are sequences of SQL operations that are treated as a single unit of work.

- They ensure data consistency and integrity by either committing all changes to the database or rolling back changes if an error occurs.

9. Security :

- MySQL provides features for securing the database, including **user authentication, access control, and encryption of data in transit and at rest.**

- User privileges can be granted or revoked to control access to databases, tables, and other resources.

What is SQL

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database

- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

SQL Syntax

SQL statement returns all records from a table named "**Customers**"

Ex : **SELECT * FROM Customers**;

SQL keywords are NOT case sensitive: **select** is the same as **SELECT**

SQL Commands

- SELECT - **extracts data from a database**: Ex: SELECT CustomerName, City FROM Customers;

Select ALL columns Eg: **SELECT * syntax**: Ex: SELECT * FROM Customers;

SELECT **DISTINCT** Ex: SELECT DISTINCT Country FROM Customers;//A,B,C,D

SELECT **Without DISTINCT** Ex: SELECT Country FROM Customers;//A,Z,B,H

WHERE Clause Ex: SELECT * FROM CustomersWHERE Country='Mexico';**//**Only Mexico Country

The following operators can be used in the WHERE clause:

= Equal

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

<> Not equal. Note: In some versions of SQL this operator may be written as !=

BETWEEN Between a certain range

LIKE Search for a pattern

IN To specify multiple possible values for a column

ORDER BY Eg: ascending or descending **Ex:** SELECT * FROM Products ORDER BY Price;

Using Both ASC and DESC Ex: SELECT * FROM Customers ORDER BY Country ASC, CustomerName DESC;

AND Operator Eg: SELECT * FROM Customers WHERE Country = 'Spain' AND CustomerName LIKE 'G%';//spain,Gain

- **The AND operator displays a record if all the conditions are TRUE.**
- **The OR operator displays a record if any of the conditions are TRUE.**

Combining AND and OR : Ex: SELECT * FROM Customers WHERE Country = 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');

OR Operator: SELECT *FROM Customers WHERE Country = 'Germany' OR Country = 'Spain';

OR vs AND:

-
- The OR operator displays a record if any of the conditions are TRUE.
- The AND operator displays a record if all the conditions are TRUE.

NOT Operator:Ex: SELECT * FROM Customers WHERE NOT Country = 'Spain';

NOT LIKE:Ex= SELECT * FROM Customers WHERE CustomerName NOT LIKE 'A%';

NOT BETWEEN:Ex= SELECT * FROM Customers WHERE CustomerID NOT BETWEEN 10 AND 60;

NOT IN:Ex= SELECT * FROM Customers WHERE City NOT IN ('Paris', 'London');

NOT Greater Than Ex= SELECT * FROM Customers WHERE NOT CustomerID > 50;

NOT Less Than Ex= SELECT * FROM Customers WHERE NOT CustomerId < 50;

INSERT INTO : Ex= INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

NULL Value: A field with a NULL value is a field with no value.

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax Ex= SELECT column_names FROM table_name WHERE column_name IS NULL;

IS NOT NULL Syntax Ex= SELECT column_names FROM table_name WHERE column_name IS NOT NULL;

- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table
- CREATE INDEX - creates an index (search key)

- DROP INDEX - deletes an index

---

1. CRUD Operation

(CRUD stands for Create, Read, Update, and Delete).

- **These are the basic operations used in database applications to manage data.**
- **Create: Adding new records to the database.**
- **Read: Retrieving existing records from the database.**
- **U**
pdate: Modifying existing records in the database.
- **Delete: Removing records from the database.**

2. Swagger (OpenAPI Specification)

(Swagger is a set of rules for describing and documenting RESTful APIs.)

- **It provides a standardized way to design, build, document, and consume APIs.**
- **Allows for automatic generation of interactive API documentation.**
- **Helps in understanding and working with APIs across different projects and platforms.**
- **Includes tools like Swagger UI for visualizing and interacting with API documentation.**

3. FastAPI and REST API:

- FastAPI is a modern, fast (hence the name), web framework for building APIs with Python.

- It supports asynchronous programming, making it highly performant.
- FastAPI automatically generates interactive API documentation using Swagger UI.
- REST (Representational State Transfer) is an architectural style for designing networked applications.
- REST APIs follow the principles of statelessness, uniform interface, and resource-based interactions.

Key Points:

- CRUD operations are fundamental for managing data in databases.
- Swagger is a specification for describing and documenting RESTful APIs.
- Python is a versatile programming language with a strong community and extensive ecosystem.
- MySQL is a popular open-source relational database management system.
- FastAPI is a modern web framework for building APIs with Python, known for its performance and automatic API documentation.

Features

- CRUD operations: Create, Read, Update, Delete.
- Swagger UI: Interactive API documentation.
- Python's simplicity and readability.
- MySQL's scalability and reliability.
- FastAPI's performance and support for asynchronous programming.

Applications

- E-commerce platform: Utilizes CRUD operations to manage product listings, orders, and customer data. Uses Swagger for API documentation.
- Social media application: Relies on CRUD operations for user profiles, posts, and interactions. Implements FastAPI for building RESTful APIs.

Fundamentals:

- Understanding of CRUD operations.
- Familiarity with Swagger/OpenAPI Specification.
- Proficiency in Python programming.
- Knowledge of relational database concepts and SQL.
- Experience with web frameworks like FastAPI for building RESTful APIs.

FAST API

An API, or **Application Programming Interface**, is **a set of rules, protocols**, **and tools** that allows different software **applications to communicate and interact with each other.** It defines the methods and data formats that applications can use to request and **exchange information**.

key components and concepts related to APIs:

1. Interface: An API defines the interface through **which two software systems can interact.** This interface includes the **methods, endpoints, and data formats** that applications can use to **communicate with each other.**

2. Protocols and Standards: APIs often rely on **specific protocols and standards for communication**, such as **HTTP** (Hypertext Transfer Protocol) for **web APIs** or **SOAP** (Simple Object Access Protocol) for web services. These protocols define how data is transmitted between the client and the server.

3. Endpoints and Methods: An API typically exposes a **set of endpoints**, which represent specific functionalities or resources that can be accessed by clients. Each endpoint supports **one or more HTTP methods (e.g., GET, POST, PUT, DELETE),** which define the actions that clients can perform on the resource.

4. Request and Response: When a client sends a request to an API endpoint, it includes information such as the desired action (e.g., **retrieve data, update a record**) and any required parameters. The API processes the request and returns a response, which may include the requested data or indicate the outcome of the operation **(e.g., success or failure).**

5. Data Formats: APIs use specific data formats for representing information in requests and responses. Common formats include JSON **(JavaScript Object Notation**) and XML **(eXtensible Markup Language),** which are both widely used for **exchanging structured data between applications.**

6. Authentication and Authorization: Many APIs require authentication to ensure that only authorized users or applications can access protected resources. **Authentication mechanisms such as API keys,** OAuth tokens, **or username/password credentials** are commonly used to secure API endpoints.

Features of APIs:

1. Accessibility: APIs should be **easily accessible** and well-documented so that developers can quickly understand how to use them and integrate them into their applications. This includes providing **clear documentation**, code examples, and support resources.

2. Versatility: APIs should support a wide range of **use cases and scenarios,** allowing developers to **perform various tasks** and access different functionalities through a **single interface**.

3. Scalability: APIs should be able to handle a large **volume of requests and scale horizontally** as the demand for the service grows. This includes implementing efficient request handling mechanisms, load balancing, and distributed computing techniques.

4. Reliability: APIs should be reliable and available, with minimal downtime and consistent performance. This requires implementing robust error handling, monitoring, and fault tolerance mechanisms to ensure that the API remains operational under different conditions.

5. Security: APIs should be secure and protect sensitive data from unauthorized access, manipulation, or interception. This includes implementing **authentication, authorization, encryption,** and **other security** measures to prevent security breaches and protect user privacy.

6. Flexibility: APIs should **be flexible and adaptable** to accommodate changes in requirements, technologies, and business needs over time. This includes supporting **versioning, backward compatibility, and extensibility** to enable seamless evolution and maintenance of the API.

7. Performance: APIs should be **performant and efficient**, with **low latency and fast response** times to ensure a smooth user experience. This requires optimizing resource usage, minimizing network **overhead, and leveraging caching** and other optimization techniques

8. Monitoring and Analytics: APIs should provide visibility into their usage and performance through **monitoring, logging, and analytics tools**. This allows developers to track usage patterns, identify bottlenecks, and optimize **the API for better performance and scalability**.

9. Developer Experience (DX): APIs should prioritize a positive developer experience, with well-designed interfaces, intuitive workflows, and developer-friendly tools and resources. This includes providing **SDKs (Software Development Kits), code generators**, and other utilities to streamline the integration process and reduce development time.

10. Compliance and Standards: APIs should adhere **to industry standards, best practices**, and regulatory requirements to ensure interoperability, compatibility, and compliance with legal and security standards. This includes following established protocols, data formats, and security guidelines such **as RESTful principles, JSON or XML formatting**, **and GDPR compliance**

Fast Api

- **Modern Web Framework:** FastAPI is a modern web framework designed for building APIs with Python 3.7+.

- **Speed:** FastAPI is known for its speed, offering significant development speed compared to traditional frameworks.

- **Automatic Documentation:** FastAPI automatically generates documentation for your web service based on your code, making it easier for other developers to understand how to use your API.

- **Simplified Testing:** The generated documentation simplifies testing of the web service by providing clear information on the required data and available endpoints.

- **Reduced Errors:** FastAPI helps reduce human errors in code by leveraging standard Python type hints for data validation and serialization.

- **Ease of Learning:** FastAPI is easy to learn, making it accessible for developers of varying skill levels.

- **Production-Ready:** FastAPI is production-ready, meaning it's robust and reliable for deploying applications in real-world environments.

- **Compatibility:** FastAPI is fully compatible with well-known API standards such as OpenAPI and JSON Schema, ensuring interoperability with other systems and tools**.**

Features of FastAPI

- **Automatic Documentation:-** FastAPI generates interactive API documentation automatically using the **OpenAPI standard**.
  - Accessible through a specific endpoint, facilitating **easy understanding and testing** of the API without extensive manual documentation.
- **Python Type Hints:** Utilizes Python-type hints to annotate function parameters and **return types, enhancing code readability.**
  - Enables automatic validation of incoming data and accurate **generation of API documentation, reducing errors** and **enhancing self-documentation**.

3. Data Validation: - Employs Pydantic models for data validation, allowing definition of data models with schema and validation capabilities.

- Ensures automatic validation, serialization, and deserialization of incoming data, minimizing risks associated with handling invalid data.

4. Asynchronous Support: - Fully embraces asynchronous operations, leveraging Python's async and await keywords. - Ideal for handling I/O-bound tasks, enhancing application responsiveness.

5. Dependency Injection:- Supports dependency injection, enabling declaration of dependencies for endpoints.- Facilitates modularity, testability, and maintainability by seamlessly injecting dependencies such as database connections and authentication into routes.

6. Security Features:- Includes various security features out of the box, such as OAuth2 and JWT support.- Automatically validates request data to mitigate common security vulnerabilities like SQL injection and cross-site scripting (XSS) attacks.

Advantage of FastAPI

- **Easy to Learn and Use**: FastAPI is designed to be straightforward, especially for Python developers. Its simple and intuitive syntax, along with automatic documentation generation, makes it easy to get started and maintain.

- **High Performance**: FastAPI is built for speed. It's one of the fastest Python web frameworks available, thanks to its asynchronous support and efficient data handling. This means your web applications can handle a large number of requests without slowing down.

- **Automatic Data Validation**: With FastAPI, you can use Python type hints to define the data structure you expect for your API requests and responses. FastAPI automatically validates the data, reducing the chances of errors caused by incorrect input.

- **Authentication and Authorization:** It provides simple ways to handle authentication and authorization, whether using OAuth2, JWT tokens, or custom methods.

- **Middleware:** We can easily add middleware to your FastAPI application for tasks like logging, authentication, or request/response modification.

Installation and Setup of FastAPI

Install fast API using the following command ***pip install fastapi***

You also need to install uvicorn ***pip install uvicorn***

Create a Simple API

Here, we are creating a simple web service that says “Hello” when you visit a specific web address. With FastAPI, you can do this in just a few lines of code, To run this code, you can save it in a Python file, here we are saving the file as main.py.

Python3

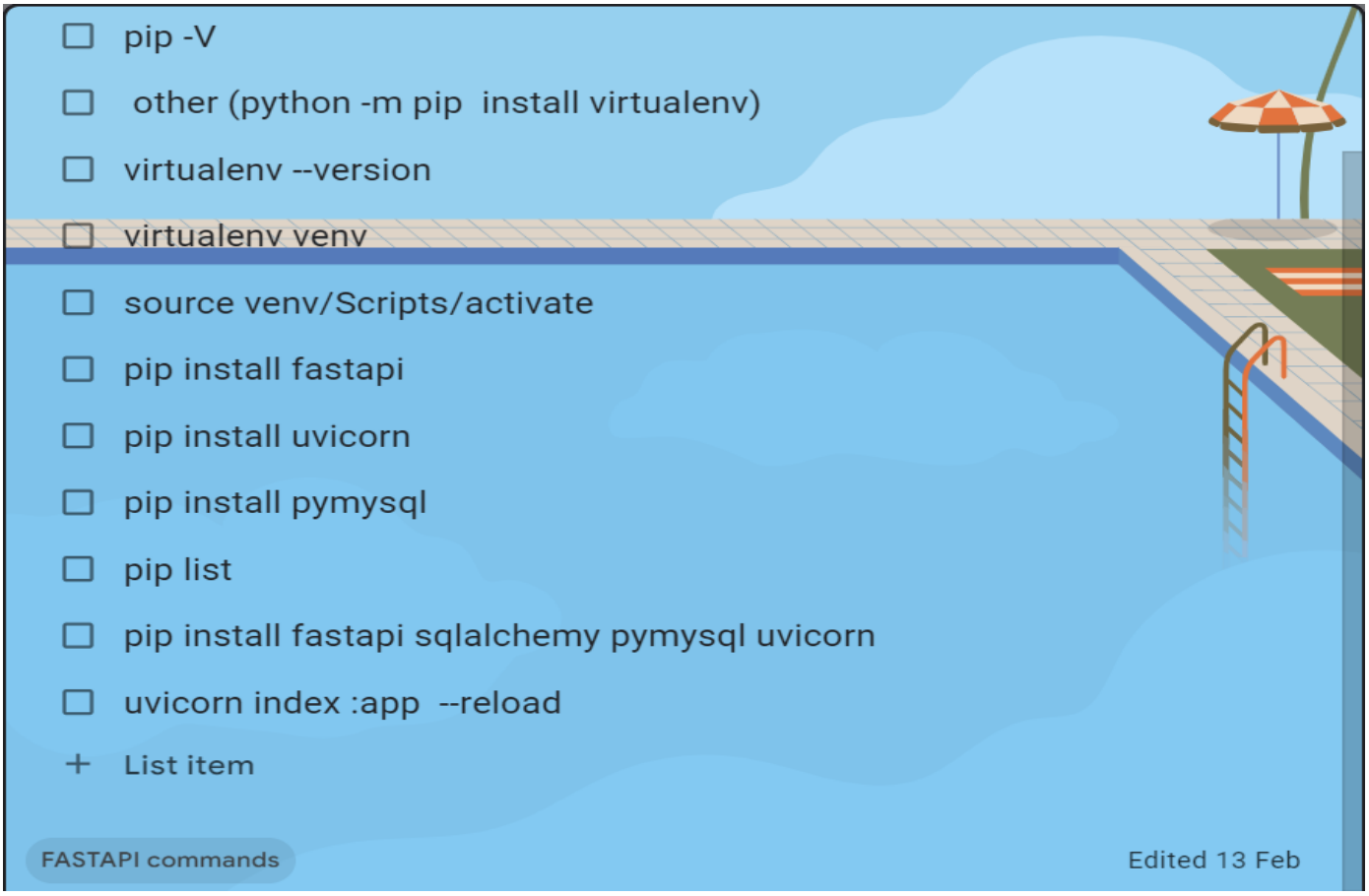
```
from fastapi import FastAPI

app = FastAPI() # Create a FastAPI application

@app.get("/") # Define a route at the root web address ("/")

def read_root():

    return {"message": "Hello, FastAPI!"}
```

Now, execute the following command in your terminal: ***uvicorn main:app --reload***

Once the application is running, open your web browser and navigate to : ***http://localhost:8000/***

You should see a message displayed in your browser or the response if you are using an API testing tool like curl or Postman. **{"message": "Hello, FastAPI!"}**

Commands


pip -V
other (python -m pip install virtualenv)
virtualenv --version
virtualenv venv
source venv/Scripts/activate
pip install fastapi
pip install uvicorn
pip install pymysql
pip list
pip install fastapi sqlalchemy pymysql uvicorn
uvicorn index :app --reload
List item
FASTAPI commands
Edited 13 Feb

Create the folders in VS code