ApexaiQ_Week2Task

Python syntax, variables, datatypes (int, float, string, list, tuple, dict, set)

Python is a high-level, interpreted programming language where variables are used to store values and datatypes define the type of value stored. Unlike languages like C or Java, Python does not require explicit declaration of variable type — it assigns automatically based on the value.

1. Variables in Python:

- A variable is a **name given to a memory location** that stores data.
- In Python, a variable is created when a value is assigned.
- Example:
- x = 10 # integer variable
- name = "A" # string variable
- price = 99.5 # float variable
- Variables are **dynamically typed**, meaning type is decided at runtime.

Rules for naming variables:

1. Must start with a letter or underscore (_value).
2. Cannot start with a number (2name ❌).
3. Cannot use keywords (if, while, etc.).
4. Should be meaningful (age, salary).

2. Datatypes in Python:
Python provides several built-in datatypes, divided into categories:

- **Numeric Types:**
- int → whole numbers (e.g., x = 100)
- float → decimal values (e.g., y = 3.14)
- **Text Type:**
- str → sequence of characters in quotes (e.g., "Hello")
- **Sequence Types:**
- list → ordered, mutable collection → [1,2,3]
- tuple → ordered, immutable collection → (1,2,3)
- range → sequence of numbers → range(5)
- **Mapping Type:**

- dict → key-value pairs → {"id": 1, "name": "Shreya"}
- **Set Types:**
- set → unordered unique items → {1,2,3}
- frozenset → immutable version of set
- **Boolean Type:**
- bool → True/False values

3. Example Code:

```
student = {
  "name": "Shreya",
  "age": 20,
  "marks": [85, 90, 92],
  "is_active": True
}
print(type(student))   # dict
print(type(student["marks"])) # list
```

4. Importance:

- Variables allow storage and reuse of values.
- Datatypes decide the kind of operations possible (e.g., numbers can be added, strings can be concatenated).
- Helps in writing structured and error-free programs.

Conditional statements (if-elif-else)

Programming is about making decisions. Conditional statements in Python allow a program to choose a path of execution depending on certain conditions. Instead of running all code sequentially, we can check conditions and run only the required block..

Explanation:

- **if statement** → executes a block if the condition is true.
- **elif statement** → (else if) checks multiple conditions.
- **else statement** → runs when all above conditions are false.

Syntax:

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

Example:

```
marks = 85
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
else:
    print("Grade C")
```

Output: Grade B

Importance / Applications:

- Used for **decision-making** in programs.
- Helps in **validation**, like checking passwords.
- Used in **real-world applications** such as login systems, billing systems, grading systems.

Loops (for, while, break, continue)

In programming, loops are used to repeat a block of code multiple times without writing it again and again. Python provides two main looping constructs: **for loop** and **while loop**. To control loop execution, we also use keywords like **break** and **continue**.

---

1. For Loop:

- Used when we know how many times we want to repeat.
- Works directly with sequences like lists, tuples, strings, or ranges.

Syntax:

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

Example:

```
for i in range(5):
    print(i)    # prints 0 1 2 3 4
```

---

2. While Loop:

- Used when we don’t know the number of repetitions in advance.
- It keeps running as long as the condition is True.

Syntax:

```
while condition:
    # block of code
```

Example:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

---

3. Break Statement:

- Used to **exit a loop immediately**, even if the condition is still true.

Example:

```
for i in range(10):
```

```
if i == 5:
    break
print(i)   # prints 0 to 4
```

---

4. Continue Statement:

- Skips the current iteration and moves to the next one.

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)   # prints 0 1 3 4
```

Importance / Applications:

- Loops are used in **repetitive tasks** such as iterating over lists, processing user input, file reading, or generating reports.
- break and continue provide **control and flexibility** in execution.
- Without loops, we would have to write repetitive code, making programs inefficient.

Functions (parameters, return values, default args, *args, **kwargs)

A function is a reusable block of code that performs a specific task. Instead of writing the same code again and again, we can define it once and call it whenever needed. Functions make programs **modular, readable, and efficient**.

1. Defining a Function:

- In Python, a function is defined using the def keyword.
- Syntax:
- def function_name(parameters):
- # body of function
- return value

2. Parameters (Inputs to Function):

- Functions can take inputs called **parameters (arguments)**.
- Example:
- def greet(name):
-     print("Hello", name)
- greet("Shreya")

3. Return Values (Outputs from Function):

- A function can send output back using the return statement.
- Example:
- def add(a, b):
-     return a + b
- result = add(5, 7)   # result = 12

---

4. Default Arguments:

- Parameters can have default values, used when no value is passed.
- Example:
- def power(base, exp=2):
-     return base ** exp
- print(power(3))   # 9
- print(power(3,3)) # 27

**5. args (Variable Positional Arguments):*

- Allows passing multiple arguments without defining them explicitly.
- Stored as a tuple.
- Example:
- def total(*numbers):
-     return sum(numbers)
- print(total(1,2,3,4))   # 10

---

6. kwargs (Variable Keyword Arguments):**

- Allows passing key-value pairs.

- Stored as a dictionary.
- Example:
- def student_info(**details):
-     for k, v in details.items():
-         print(k, ":", v)
- student_info(name="Shreya", age=20, marks=85)

Importance / Applications:

- Functions help in **code reusability** and **modularity**.
- They make programs **easier to debug and maintain**.
- Default args, *args, and **kwargs give flexibility to handle any number of inputs.
- Widely used in real-world apps like **data processing, web APIs, machine learning models**.

Exception Handling (try-except-finally)

While running a program, unexpected errors can occur, such as dividing a number by zero, accessing an invalid file, or wrong user input. These runtime errors are called **exceptions**.
If not handled, they stop the program. Python provides **exception handling** to manage errors gracefully without crashing.

---

1. What is an Exception?

- An exception is an **error that occurs during program execution**.
- Example:
- x = 10 / 0   # ZeroDivisionError

---

2. Why Exception Handling?

- Without handling, the program stops immediately when an error occurs.
- With handling, we can **display user-friendly messages** and allow the program to continue.

---

3. try-except-finally Block:
Python provides three main keywords for handling exceptions:

- **try:** contains code that may cause an error.
- **except:** handles the error.
- **finally:** block always executes, whether error occurs or not.

Syntax:

```
try:
    # risky code
except ExceptionType:
    # handling code
finally:
    # always executed
```

---

4. Example:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero")
except ValueError:
    print("Error: Invalid input")
finally:
    print("Execution complete")
```

Explanation:

- If user enters 0 → ZeroDivisionError is caught.

- If user enters "abc" → ValueError is caught.
- The finally block always runs to release resources.

---

5. Importance / Applications:

- Prevents sudden **program crash**.
- Ensures **reliability** of applications (like banking systems, file handling, web apps).
- finally ensures **cleanup operations** such as closing files, freeing memory, disconnecting database connections.

Decorators

A **decorator** in Python is a special type of function that allows us to **modify or extend the behavior of another function or method without changing its actual code**. They are widely used in frameworks like Flask and Django to add extra features such as authentication, logging, or timing execution.

1. What is a Decorator?

- A decorator is a **higher-order function** (a function that takes another function as input and returns a new function).
- It is applied using the @decorator_name syntax above a function definition.

---

2. Working Principle:

- A decorator wraps another function and can run code **before and/or after** the original function.

Syntax:

```
def decorator(func):
    def wrapper():
        # code before function
        func()
        # code after function
    return wrapper
```

---

3. Example:

```
def my_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper
@my_decorator
def say_hello():
    print("Hello World")
say_hello()
```

Output:

```
Before function runs
Hello World
After function runs
```

---

4. Decorator with Arguments:

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
```

```
@repeat(3)
def greet(name):
    print("Hello", name)
greet("Shreya")
```

→ Prints greeting 3 times.

---

5. Importance / Applications:

- Used in **web frameworks** for tasks like authentication, logging, caching.
- Allows **separation of concerns** → core logic remains clean.
- Adds **reusability and flexibility** → the same decorator can be applied to multiple functions.
- Useful in **testing and debugging** to track execution.

OOPS

Object-Oriented Programming (OOPS) is a programming paradigm based on the concept of **objects**. Instead of writing code as separate functions and variables, OOPS allows us to bundle **data (attributes)** and **functions (methods)** together inside classes. Python is a **multi-paradigm language** that supports both procedural and object-oriented programming.

1. Key Concepts of OOPS in Python:

1. **Class:**

- A blueprint for creating objects.
- Contains attributes (variables) and methods (functions).

3. class Student:
4. def **init**(self, name, age):
5. self.name = name
6. self.age = age
7. **Object:**

- An instance of a class.

9. s1 = Student("Shreya", 20) # object creation
10. **Encapsulation:**

- Wrapping data and methods inside a class.

- Restricts direct access and provides **data hiding**.

```
class Account:
    def __init__(self, balance):
        self.__balance = balance   # private variable
```

Inheritance:

- One class (child) can reuse properties/methods of another (parent).

```
class Animal:
    def sound(self): print("Animal sound")
class Dog(Animal):    # inherits Animal
    def sound(self): print("Bark")
```

Polymorphism:

- Same method name but different behavior depending on object.

```
for animal in [Dog(), Animal()]:
    animal.sound()   # Output changes depending on object
```

Abstraction:

- Hiding implementation details and showing only necessary features.
- Achieved using **abstract classes** in Python (abc module).

---

2. Example Program (Demonstrating All Concepts):

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
```

```
    def area(self):
        return self.length * self.width
```

Object creation

```
rect = Rectangle(5, 3)
print("Area of rectangle:", rect.area())
```

---

3. Importance / Applications of OOPS:

- Provides **code reusability** (through inheritance).
- Improves **data security** with encapsulation.
- Makes code **organized and modular**.
- Polymorphism allows flexibility and scalability.
- Used in **real-world applications** like GUI systems, games, banking systems, and web development (Django, Flask).

List comprehension, dictionary comprehension

In Python, **comprehensions** are concise ways to create **lists, dictionaries, or sets** using a single line of code. They make code shorter, readable, and more **Pythonic** compared to traditional loops.

---

1. List Comprehension

- A method to create a **list** using a single line instead of multiple lines with loops.
- **Syntax:**

```
[expression for item in iterable if condition]
```

Example 1: Basic List Comprehension

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
print(squares)   # Output: [1, 4, 9, 16, 25]
```

Example 2: With Condition

```
even_numbers = [x for x in numbers if x % 2 == 0]

print(even_numbers)   # Output: [2, 4]
```

Advantages:

- Shorter and readable than traditional loops.
- Efficient for creating transformed lists quickly.

---

2. Dictionary Comprehension

- Similar to list comprehension but used to create **dictionaries**.
- **Syntax:**

```
{key_expression: value_expression for item in iterable if condition}
```

Example 1: Basic Dictionary Comprehension

```
numbers = [1, 2, 3, 4]

squares_dict = {x: x**2 for x in numbers}

print(squares_dict)   # Output: {1:1, 2:4, 3:9, 4:16}
```

Example 2: With Condition

```
even_squares = {x: x**2 for x in numbers if x % 2 == 0}

print(even_squares)   # Output: {2: 4, 4: 16}
```

Advantages:

- Creates dictionaries efficiently in a single line.
- Reduces code length and improves readability.

---

3. Importance / Applications:

- **Data transformation:** Apply operations to each element in a list or dictionary.
- **Filtering:** Easily include only required elements using conditions.
- **Pythonic code:** Used widely in data science, web development, and automation.

Iterators & Generators

In Python, **iteration** means accessing elements of a collection (like a list or tuple) one by one. Python provides **iterators** and **generators** to traverse data efficiently, especially for large datasets, without storing everything in memory at once.

---

1. Iterators

Definition:

- An **iterator** is an object that implements the methods **iter**() and **next**() to traverse through elements one by one.

How it Works:

1. iter() → converts a collection into an iterator.
2. next() → gets the next element from the iterator.

Example:

numbers = [1, 2, 3]

it = iter(numbers) # create iterator

print(next(it)) # 1

print(next(it)) # 2

print(next(it)) # 3

print(next(it)) # Raises StopIteration

Advantages:

- Efficient for traversing collections.
- Can be used with **for loops**, which internally use iterators.

---

2. Generators

Definition:

- A **generator** is a **special type of iterator** created using a function with the yield keyword.
- It produces values **one at a time**, saving memory.

Syntax:

```
def generator_function():
    for i in range(5):
        yield i
```

Example:

```
def squares(n):
    for i in range(n):
        yield i**2
gen = squares(5)
for val in gen:
    print(val)
```

Output:

0 1 4 9 16

Advantages:

- Generates values on demand (**lazy evaluation**) → memory efficient.
- Suitable for **large datasets or infinite sequences**.
- Can be iterated using for or next().

---

3. Differences Between Iterators and Generators

Feature	Iterator	Generator
Creation	Using iter() or class with **iter**	Using function with yield
Memory Usage	Stores all elements in memory	Produces elements one at a time
Syntax	Requires class with methods	Simple function with yield
Use Case	Small collections or custom classes	Large datasets or streams

---

4. Importance / Applications:

- Essential for **efficient data processing**.
- Widely used in **data science, web scraping, and machine learning** for handling large data.
- Saves memory and increases program performance.

Virtual environments & pip (optional)

When working on multiple Python projects, each project may require **different versions of libraries**. Installing all packages globally can create **conflicts**. Python provides **virtual environments** to isolate project dependencies, and **pip** to manage packages efficiently.

---

1. Virtual Environments

Definition:

- A virtual environment is a **self-contained directory** with its own Python interpreter and installed packages, separate from the system Python.

Why Use It:

- Avoids **dependency conflicts** between projects.
- Ensures **consistent development and deployment**.

Creating & Using Virtual Environments:

1. **Create a virtual environment:**

python -m venv myenv

2. **Activate the environment:**

- Windows: myenv\Scripts\activate
- Linux/Mac: source myenv/bin/activate

3. **Deactivate the environment:**

deactivate

Example:

Create environment

python -m venv project_env

Activate environment

source project_env/bin/activate # Linux/Mac

project_env\Scripts\activate # Windows

Install packages (example)

pip install requests

Deactivate environment

deactivate

---

2. pip (Python Package Installer)

Definition:

- pip is the **package manager for Python**, used to install, upgrade, or remove Python packages from the Python Package Index (PyPI).

Common pip Commands:

- Install a package: pip install package_name
- Upgrade a package: pip install --upgrade package_name
- Uninstall a package: pip uninstall package_name
- List installed packages: pip list

Example:

pip install numpy # installs numpy library

pip list # shows installed packages

pip uninstall numpy # removes numpy

---

3. Importance / Applications:

- **Virtual environments** allow multiple projects to have **different library versions** without conflicts.
- **pip** simplifies package management, enabling developers to quickly install and maintain libraries.
- Essential for **project deployment, collaborative development, and reproducibility**.

Standard libraries (optional)

Python comes with a rich set of **built-in modules and libraries** that provide ready-to-use functionalities. These are called **standard libraries**. They save time, avoid reinventing the wheel, and make Python suitable for **various applications** like math operations, file handling, date/time processing, web development, and more.

---

1. What are Standard Libraries?

- Pre-installed Python modules that can be **imported** and used without additional installation.
- Example syntax:

import math

import datetime

---

2. Important Standard Libraries and Their Uses:

Library	Purpose / Functionality
math	Mathematical operations (sqrt, sin, cos, factorial, etc.)
random	Generate random numbers, shuffle lists, select random elements
datetime	Work with dates and times (current date, timedelta, formatting)
os	Interact with operating system (file operations, paths)
sys	System-specific parameters and functions (argv, exit)
json	Handle JSON data (serialization and deserialization)
re	Regular expressions for text pattern matching
csv	Read/write CSV files easily
time	Time-related functions (sleep, time calculation)
collections	Specialized data structures (deque, Counter, OrderedDict)

3. Example Usage:

Math Library:

```
import math

print(math.sqrt(16))   # Output: 4.0

print(math.factorial(5)) # Output: 120
```

Random Library:

```
import random

print(random.randint(1, 10))   # Random integer between 1 and 10
```

Datetime Library:

```
from datetime import datetime

now = datetime.now()

print(now)   # Output: current date and time
```

---

4. Importance / Applications:

- Saves **development time** as we don't need to write common functions from scratch.
- Makes programs **robust and standardized**.
- Widely used in **data analysis, web development, automation, AI/ML, and system scripting**.

*Docstring
Comments

Comments

Introduction:
Coding standards are **rules and guidelines** that help programmers write **clean, readable, and maintainable code**. Following them is crucial in **team projects, debugging, and future updates**. Python emphasizes readability, and these standards are part of **PEP8 guidelines**.

---

1. Naming Conventions

Definition:

- Naming conventions are **rules for naming variables, functions, classes, constants, and modules** to make code readable and meaningful.

Guidelines:

Entity	Convention	Example
Variable	snake_case	total_marks = 100
Function	snake_case	calculate_sum()
Class	PascalCase or CamelCase	StudentDetails
Constant	UPPER_CASE	PI = 3.14
Module/File	lowercase	math_utils.py

Importance:

- Improves **readability and understanding** of code.
- Helps in **team collaboration**.
- Avoids **confusion with Python keywords**.

---

2. Docstrings

Definition:

- Docstrings (Documentation Strings) are **multi-line strings at the beginning of a module, class, or function** that describe its purpose.
- Accessible at runtime using the **doc** attribute.

Syntax:

```
def add(a, b):
    """This function returns the sum of two numbers."""
    return a + b
print(add.__doc__)
```

Output:

This function returns the sum of two numbers.

Importance:

- Explains **what the function/class/module does**.
- Helps in **maintaining large codebases**.
- Used by **IDE tools and documentation generators**.

---

3. Comments

Definition:

- Comments are **non-executable lines** in code meant to explain **logic, steps, or purpose**.

Types in Python:

1. **Single-line comment:** Starts with #
2. # This is a single-line comment
3. print("Hello World")
4. **Multi-line comment:** Enclosed in ''' ''' or """ """
5. """
6. This is a multi-line comment
7. explaining the logic of the program
8. """
9. print("Hello")

Importance:

- Makes code **easier to read and understand**.
- Useful for **debugging and collaboration**.
- Essential for **examining logic quickly** in large programs.

- Types of testing
- PEP8
- SOLID and DRY principles

Coding standards are essential to write **high-quality, readable, and maintainable code**. They include **testing practices, style guidelines, and programming principles**. Following them ensures **error-free, efficient, and scalable software**.

1. Types of Testing

Testing is the process of **verifying that software works as expected**. Key types:

Type	Description	Example / Purpose
Unit Testing	Tests individual functions or modules for correctness	Testing add(a,b) function independently
Integration Testing	Checks how multiple modules work together	Testing a login system (frontend + backend)
System Testing	Verifies the **complete system** against requirements	Testing the whole e-commerce application
Acceptance Testing	Determines if system meets **user requirements**	User tests a software before deployment
Regression Testing	Ensures that new code **doesn't break old functionality**	After adding a new feature, check old features

Importance:

- Detects bugs early and prevents **software failure**.
- Improves **quality, reliability, and user satisfaction**.
- Reduces **cost and time** of fixing issues later.

---

2. PEP8 (Python Enhancement Proposal 8)

Definition:

- PEP8 is the **official style guide for Python code**.
- Ensures code is **consistent, readable, and Pythonic**.

Key Guidelines:

- **Indentation:** 4 spaces per level (no tabs).
- **Line Length:** Max 79 characters per line.
- **Naming:** snake_case for variables/functions, PascalCase for classes.
- **Spacing:** Around operators and after commas.
- **Imports:** Each import on a separate line, standard libraries first.

Example:

```
def calculate_sum(a, b):
    return a + b
```

```
total = calculate_sum(5, 10)
```

Importance:

- Makes code **consistent across teams**.
- Helps in **maintaining large projects**.
- Used by tools like **PyCharm, flake8, black** for automated checks.

---

3. SOLID Principles

SOLID is a set of **OOP design principles** to write maintainable and scalable code:

Principle	Meaning & Purpose
S – Single Responsibility Principle	A class should have **one responsibility** only.
O – Open/Closed Principle	Code should be **open for extension, closed for modification**.
L – Liskov Substitution Principle	Child classes should **replace parent classes without errors**.
I – Interface Segregation	Prefer **smaller interfaces**, not one large interface for everything.
D – Dependency Inversion	High-level modules should **not depend on low-level modules** directly.

Importance:

- Ensures **modular, reusable, and flexible code**.
- Reduces **bugs** and improves **code maintenance**.
- Essential for **large-scale OOP projects**.

---

4. DRY Principle (Don’t Repeat Yourself)

Definition:

- A programming principle that **avoids duplication** of code by using functions, classes, or modules.

Example:

Instead of repeating

```
print("Hello")

print("Hello")

print("Hello")
```

Use DRY approach

```
def greet():
    print("Hello")

for _ in range(3):
    greet()
```

Importance:

- Reduces **code redundancy and errors**.
- Makes **maintenance easier**.
- Improves **readability and efficiency**.

- API
- Types of APIs

API (Application Programming Interface) is a set of rules that allows **different software applications to communicate with each other**. It acts as a bridge, enabling programs to **request and exchange data** without knowing the internal workings of the other system. APIs are widely used in **web development, mobile apps, cloud services, and software integration**.

1. What is an API?

- API allows one software application to **interact with another** using defined methods, inputs, and outputs.
- Example:
- A weather app uses an API to **get real-time weather data** from a weather service.

Key Features:

1. **Interface:** Provides a clear way for software to interact.
2. **Abstraction:** Hides the internal details of the system.

3. **Standardization:** Follows common protocols like HTTP, REST, or SOAP.

Example in Python:

```
import requests

response = requests.get("https://api.coindesk.com/v1/bpi/currentprice.json")

data = response.json()

print(data["bpi"]["USD"]["rate"])
```

---

2. Types of APIs

Type	Description	Example
Web API	Communicates over the internet using HTTP protocols	REST API for retrieving user data
REST API	Uses HTTP methods (GET, POST, PUT, DELETE) and JSON/XML for data exchange	Twitter API, GitHub API
SOAP API	Protocol-based API using XML messages	Payment gateways (PayPal, banking APIs)
Open/Public API	Available to external developers without restrictions	Google Maps API
Private API	Restricted for internal use within a company	Internal HR system APIs
Partner API	Shared with specific partners under agreement	Payment API shared with e-commerce partners
Library/Module API	Provides functions or classes that other programs can use	Python math or requests module

Importance / Applications:

- **Integration:** Enables apps and services to work together.
- **Automation:** Automates tasks across software systems.
- **Data Access:** Allows real-time access to external services and resources.
- **Scalability:** Makes software modular and extendable.

- HTTP Status codes
- Response Formats

When a client (like a web browser or mobile app) communicates with a server via an API, the server responds with **status codes** and **data formats**. These help the client understand whether the request was successful and how to process the returned data.

---

1. HTTP Status Codes

Definition:

- HTTP Status Codes are **three-digit numbers** sent by a server to indicate the **result of a client's request**.

Categories:

Code Range	Meaning	Examples / Description
1xx	Informational	100 Continue – request received, waiting
2xx	Success	200 OK – request succeeded, 201 Created – new resource created
3xx	Redirection	301 Moved Permanently – resource moved
4xx	Client Error	400 Bad Request – wrong syntax, 401 Unauthorized – invalid credentials, 404 Not Found – resource missing
5xx	Server Error	500 Internal Server Error, 503 Service Unavailable

Example in Python using requests:

```
import requests

response = requests.get("https://api.github.com")

print(response.status_code)  # 200 means success
```

Importance:

- Helps **clients understand request outcomes**.
- Enables **error handling and debugging**.
- Essential for **robust API integration**.

---

2. Response Formats

Definition:

- APIs return data in **structured formats** that the client can parse and use.
- Common response formats:

Format	Description	Example
JSON	JavaScript Object Notation, lightweight, human-readable	{"name": "Shreya", "age": 20}
XML	eXtensible Markup Language, structured, widely used in SOAP APIs	Shreya20
HTML	Used in web pages, sometimes returned by REST APIs	**Hello World**
Plain Text	Simple string, often used for logs or messages	Success

Example in Python (JSON response):

```
import requests

response = requests.get("https://api.coindesk.com/v1/bpi/currentprice.json")

data = response.json()

print(data["bpi"]["USD"]["rate"])
```

Importance:

- Standardized formats make **data exchange easy** between client and server.
- JSON is **lightweight, readable, and widely used** in modern APIs.
- Allows **automated parsing and processing** in applications.

- Types of API Auth
- Versioning and Security
- CRUD operations

APIs allow applications to **communicate and share data**, but this requires **secure access, version management, and standardized operations**. Understanding authentication, versioning, security, and CRUD operations is crucial for designing robust APIs.

---

1. Types of API Authentication

Definition:
API Authentication ensures that **only authorized clients** can access the API.

Common Types:

Type	Description	Example
API Key	Simple key passed in request header or URL	api_key=12345
Basic Auth	Username & password encoded in request header	Authorization: Basic
OAuth 2.0	Token-based authentication for secure access	Used in Google, Facebook APIs
JWT (JSON Web Token)	Encodes user info in token, verified by server	Authorization: Bearer

Importance:

- Ensures **only authorized users** can access sensitive data.
- Protects API from **unauthorized usage and abuse**.

2. API Versioning & Security

Versioning:

- Allows **updating APIs** without breaking existing clients.
- Common methods:
- **URL versioning:** /api/v1/users
- **Header versioning:** Accept: application/vnd.company.v1+json
- Ensures backward compatibility for older clients.

Security:

- **Encryption:** Use HTTPS for secure data transfer.
- **Rate Limiting:** Prevents abuse by limiting request frequency.
- **Input Validation:** Prevents malicious inputs (SQL injection, XSS).
- **Authentication & Authorization:** Ensures only valid users perform actions.

---

3. CRUD Operations

Definition:
CRUD stands for **Create, Read, Update, Delete** – the four basic operations of persistent storage. APIs often map **HTTP methods** to CRUD actions:

CRUD Operation	HTTP Method	Description	Example API Call
Create	POST	Add a new resource	POST /users
Read	GET	Retrieve existing resource(s)	GET /users/1
Update	PUT/PATCH	Modify an existing resource	PUT /users/1
Delete	DELETE	Remove a resource	DELETE /users/1

Example in Python using requests:

import requests

Read

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

print(response.json())

Importance:

- CRUD operations form the **foundation of API design**.
- Enables **standardized interaction** between client and server.
- Ensures **data integrity, consistency, and ease of maintenance**.

- Explore POSTMAN (optional)
- Optimization and Efficiency
- Requests lib in Python
- RBAC (optional)

APIs are crucial for software integration, and understanding **testing tools, performance optimization, Python integration, and access control** ensures efficient and secure API usage.

---

1. Explore POSTMAN (Optional)

Definition:

- POSTMAN is a **graphical API testing tool** used to send requests to APIs, inspect responses, and automate workflows.

Features:

- Supports **HTTP methods**: GET, POST, PUT, DELETE.
- Allows setting **headers, query parameters, and request body**.
- Can **save collections** of requests for reuse.
- Enables **automated testing and documentation generation**.

Example:

- Testing a REST API endpoint to get user data:
- Method: GET
- URL: https://api.example.com/users
- Response: JSON containing user details.

Importance:

- Helps developers **validate API functionality** before coding.
- Detects **errors, performance issues, and integration problems** early.

---

2. Optimization and Efficiency

Definition:

- Optimization ensures APIs respond **quickly, efficiently, and with minimal resource usage**.

Techniques:

1. **Caching:** Store frequent responses to reduce load.
2. **Pagination:** Limit data returned per request to improve performance.
3. **Data Minimization:** Send only necessary fields.
4. **Asynchronous Processing:** Avoid blocking operations for faster responses.
5. **Compression:** Reduce size of transmitted data.

Importance:

- Enhances **user experience** with faster responses.
- Reduces **server load and bandwidth usage**.

- Makes APIs **scalable and reliable**.

---

3. Requests Library in Python

Definition:

- requests is a Python library to **interact with APIs** using HTTP requests.

Key Features:

- Supports GET, POST, PUT, DELETE methods.
- Handles **headers, query parameters, authentication, and JSON** easily.
- Returns **response objects** for easy access to data.

Example:

import requests

GET request

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

print(response.status_code) # 200

print(response.json())   # JSON response

POST request

data = {"title": "Hello", "body": "World", "userId": 1}

response = requests.post("https://jsonplaceholder.typicode.com/posts", json=data)

print(response.status_code) # 201 Created

Importance:

- Simplifies **API testing and integration** in Python projects.
- Reduces **complexity in handling HTTP requests and responses**.

---

4. RBAC (Role-Based Access Control) (Optional)

Definition:

- RBAC is a **security model** where users are assigned roles, and each role has **specific permissions**.

Example:

- Roles and permissions in a system:
- Admin → Create, Read, Update, Delete resources.
- Editor → Read and Update only.
- Viewer → Read only.

Importance:

- Ensures **security and privacy** of sensitive data.
- Prevents **unauthorized actions** and reduces risk of data breaches.

- SDLC
- Agile Basics
- Version Control
- Software Architecture

In software development, understanding **how projects are planned, executed, managed, and maintained** is crucial. Topics like **SDLC, Agile methodology, version control, and software architecture** provide a structured approach to build **efficient, scalable, and maintainable software**.

1. SDLC (Software Development Life Cycle)

Definition:

- SDLC is a **structured process** for planning, creating, testing, deploying, and maintaining software.
- Ensures **quality, efficiency, and predictability**.

Phases of SDLC:

Phase	Description
Requirement Analysis	Understand client needs and gather requirements
Design	Prepare system and software architecture/design
Implementation	Write code and develop software modules
Testing	Verify software functionality and fix defects
Deployment	Release software to production environment

Phase	Description
Maintenance	Update and fix software post-deployment

Importance:

- Reduces **errors and delays**.
- Improves **quality, consistency, and predictability**.
- Ensures **successful delivery of software projects**.

---

2. Agile Basics

Definition:

- Agile is a **flexible software development methodology** focusing on **incremental delivery, collaboration, and adaptability**.

Key Principles:

- Deliver **small, working software** frequently.
- Encourage **customer collaboration** over strict contracts.
- Respond to **change quickly** instead of following rigid plans.
- Promote **self-organizing teams**.

Common Agile Practices:

- **Scrum:** Work divided into sprints (2–4 weeks), with daily standups and reviews.
- **Kanban:** Visualize tasks and workflow to improve efficiency.

Importance:

- Faster delivery of functional software.
- Better alignment with **changing requirements**.
- Enhances **team collaboration and transparency**.

---

3. Version Control

Definition:

- Version control systems track **changes in code or documents** over time.

Types:

- **Centralized Version Control (CVCS):** One central repository (e.g., SVN).
- **Distributed Version Control (DVCS):** Multiple repositories, each developer has a copy (e.g., Git).

Key Concepts:

- **Commit:** Save changes.
- **Branch:** Create a separate line of development.
- **Merge:** Combine branches.
- **Pull/Push:** Sync changes with remote repository.

Importance:

- Enables **team collaboration** without overwriting code.
- Keeps **history of changes** for tracking and rollback.
- Facilitates **continuous integration and deployment (CI/CD)**.

---

4. Software Architecture

Definition:

- Software architecture defines the **high-level structure of a system**, including components, relationships, and design principles.

Common Architectural Patterns:

Pattern	Description
Monolithic	Single unified codebase
Client-Server	Server provides resources, clients request
Microservices	Small independent services communicating via APIs
MVC (Model-View-Controller)	Separates data, user interface, and control logic

Importance:

- Provides **scalability, maintainability, and modularity**.
- Improves **system performance and reliability**.

- Guides developers to **build structured and robust software**.