Apexa_02

REGEX:

Regex (short for **regular expression**) is a sequence of characters that define a **search pattern**. It is used for pattern matching in text, commonly in string searching, validation, and text manipulation.

Common Uses of Regex

- Searching for specific patterns in text
- Replacing text based on patterns
- Validating input formats (e.g., emails, phone numbers, dates)
- Extracting information from structured data

Basic Regex Syntax

Pattern	Meaning	Example Match
`.`	Any single character	`c.t` matches "cat", "cut", "cot"
`^`	Start of a string	`^Hello` matches "Hello, World" but not "Say Hello"
`$`	End of a string	`world$` matches "Hello, world"
`\d`	Any digit (0-9)	`\d\d\d` matches "123"
`\w`	Any word character (a-z, A-Z, 0-9, _)	`\w+` matches "hello123"
`\s`	Any whitespace (space, tab, newline)	`\s+` matches spaces in "hello world"
`*`	0 or more occurrences	`a*` matches "", "a", "aaa"
`+`	1 or more occurrences	`a+` matches "a", "aaa" but not ""
`?`	0 or 1 occurrence	`colou?r` matches "color" and "colour"
`{n,m}`	Between n and m occurrences	`\d{2,4}` matches "12", "123", "1234"
`	`	`	OR operator

`()`	Grouping	`(abc)+` matches "abc", "abcabc"

Multithreding:

What is Multithreading?

Multithreading is a technique where **multiple threads run concurrently** within a single process, sharing the same memory space. It helps in performing **multiple tasks simultaneously**, making programs more efficient.

However, in Python, **due to the Global Interpreter Lock (GIL)**, only one thread executes Python bytecode at a time. This means Python threads are more useful for **I/O-bound tasks** (like network requests, file I/O) rather than CPU-bound tasks.

Why Use Multithreading?

Efficient for I/O-bound tasks (file handling, web scraping, network calls)

Faster execution (for tasks that involve waiting for resources)

Better resource utilization (multiple tasks sharing CPU time)

Not useful for CPU-intensive tasks (e.g., heavy computations like ML training, which require multiprocessing instead)

SDLC (Software Development Life Cycle):

SDLC (Software Development Life Cycle) is a **step-by-step process** used to design, develop, test, and deploy software. It ensures that software is built **efficiently, with high quality, and meets user requirements**.

Phases of SDLC

Phase	Description
1. Planning	Define project scope, budget, and timeline. Identify risks and feasibility.

2. Requirement Analysis	Gather and document software requirements from stakeholders.
3. Design	Create architecture, UI/UX design, and system models. Choose technologies.
4. Development	Write the actual code using programming languages.
5. Testing	Identify and fix bugs through unit, integration, and system testing.
6. Deployment	Release the software to users (cloud, servers, app stores).
7. Maintenance	Fix issues, release updates, and improve performance over time.

SDLC Models

There are different ways to follow SDLC:

- **Waterfall Model** → One step at a time, no going back.
- **Agile Model** → Continuous development & quick updates.
- **V-Model** → Testing at every stage.
- **Spiral Model** → Risk-based development with iterations.

Agile/Scrum:

Agile and Scrum are **project management** and **software development** methodologies. They help teams deliver software **faster, with flexibility and continuous improvement**.

If you're working on a **Python project** (e.g., web app, AI model, automation script), Agile & Scrum help you manage tasks efficiently.

- **Agile** = A broad **framework** that focuses on **iterative development** and continuous feedback.
- **Scrum** = A **specific Agile framework** that organizes work into short development cycles called **Sprints** (2-4 weeks).
- **Frequent Releases** → Deliver small, working parts **every few weeks**.
- **Customer Collaboration** → Get feedback early and make changes.

- **Continuous Improvement** → Optimize Python code after each iteration.
- **Small, Cross-Functional Teams** → Developers, testers, and designers work together.

Unit Testing and penetration checks :

Unit Testing (Ensuring Code Works Correctly)

Unit testing is like **checking each ingredient in a recipe** before making the final dish. It ensures that **small pieces of code (functions, classes, modules)** work correctly **before** combining them.

Key Idea: Test small parts of the code **independently** to catch errors early.

Example: If you're building a **calculator app**, you should test whether `add(2,3)` correctly returns `5`.

Why Unit Testing?

Catches **bugs early** before they break the whole system.

Makes code **easier to debug** and maintain.

Helps in **automating** testing instead of manual checks.

Penetration Testing (Checking Security Vulnerabilities)

Penetration testing (or **pen testing**) is like **hiring a hacker to attack your system** and see where it is weak.

Key Idea: Find & fix **security loopholes** before **real hackers** exploit them.

Example: If your website has a login form, pen testing checks if a hacker can **bypass it using SQL injection**.

Why Penetration Testing?

Protects sensitive data (user credentials, credit cards, etc.).

Prevents **hacking, data breaches, and financial losses**.

Helps comply with **security standards** like **OWASP, ISO 27001**.

PEP8 Coding standard:

- Use 4 spaces per indentation level (no tabs).
- Keep line length ≤ 79 characters (72 for docstrings).
- Use blank lines to separate functions and classes.
- Variables and functions: snake_case.
- Classes: PascalCase.
- Constants: ALL_CAPS.
- One import per line.
- Use absolute imports instead of relative imports.
- Place imports at the top of the file.
- No spaces inside parentheses, brackets, or braces.
- One space before and after operators.
- Use `#` for inline and single-line comments.
- Use triple quotes for docstrings in functions and classes.
- Keep comments short and meaningful.
- Avoid mutable default arguments in functions.
- Keep functions short and focused.
- Use descriptive names for variables and functions.
- Use `is` for comparing with `None`.
- Use `==` for numerical and string comparisons.
- Use `try-except` for handling errors.
- Catch specific exceptions instead of generic `Exception`.
- Follow DRY (Don't Repeat Yourself) principle.
- Write modular code with reusable functions.

- Use list comprehensions instead of loops when possible.

Wrapper function:

wrapper function is a function that **wraps** another function to modify or extend its behavior **without changing the original function's code**. It is commonly used in Python **decorators** to dynamically alter the behavior of functions.

Why Use Wrapper Functions?

- **Code Reusability** – Avoid repeating the same logic in multiple functions.
- **Enhancing Functionality** – Modify or add features without modifying the original function.
- **Logging and Debugging** – Automatically log function calls and execution times.
- **Access Control** – Restrict access based on conditions like authentication.
- **Exception Handling** – Automatically catch and handle errors.

How Does a Wrapper Function Work?

A wrapper function:

1. **Accepts** another function as input.
2. **Executes extra logic** before or after calling the original function.
3. **Returns** the modified function.

Logging:

Logging in Python is a way to **track events** that happen while a program runs. It helps in **debugging, monitoring, and auditing** applications by recording messages in a structured format.

Instead of using print() statements, logging provides a **flexible and configurable** way to handle messages at different levels (info, warning, error, etc.).

- **Debugging & Troubleshooting** – Helps identify issues in the code.

- **Monitoring & Performance Analysis** – Tracks application behavior.
- **Error Tracking** – Captures runtime errors.
- **Security & Auditing** – Logs critical system events.
- **Production Readiness** – Helps maintain logs in production environments.

Ruff:

Ruff is a **fast Python linter and code formatter** designed to check for coding errors, enforce style guidelines, and improve performance. It is an alternative to tools like **Flake8, pylint, and Black**, but is significantly faster and supports many linting rules out of the box.

Super Fast– Written in Rust, making it much faster than other Python linters.

All-in-One– Combines functionalities of Flake8, Black, isort, and more.

Highly Configurable– Supports custom rules and exclusions.

Auto-fix Feature– Can automatically fix many issues.

Works with Large Codebases– Optimized for speed and efficiency.

When to Use Ruff?

If you need **fast** linting and formatting.

If you work on **large codebases** and need **efficient** tools.

If you want **one tool** instead of multiple (Black, isort, Flake8, etc.).

If you need **automatic fixes** for code issues.

1. **Performance and Speed:**
    - Ruff is **10-100x faster** than Flake8 or Pylint due to its implementation in Rust.
    - Ideal for **large projects** where linting speed is crucial.
2. **Multiple Functionality in One Tool:**
    - **Linting:** Detects syntax errors, code smells, and security vulnerabilities.
    - **Formatting:** Ensures code follows PEP 8 style guidelines, similar to Black.

- **Import Sorting:** Automatically arranges import statements, replacing `isort`.
- **Autofixing Issues:** It can automatically correct many detected problems.

3. **Highly Configurable:**
   - Supports customization via `pyproject.toml`.
   - Allows enabling/disabling specific linting rules.
   - Can integrate with various development environments.
4. **PEP 8 Compliance:**
   - Ensures adherence to Python's official style guide (PEP 8).
   - Identifies long lines, missing docstrings, redundant spaces, etc.
5. **Security & Best Practices:**
   - Detects potential **security vulnerabilities** in code.
   - Identifies **unused variables, imports, and bad coding patterns**.

Ruff analyzes Python code files and checks them against a set of predefined rules. These rules are based on:

- **PEP 8 (Python Enhancement Proposal 8):** Official style guide for Python.
- **PEP 257:** Docstring conventions.
- **PEP 484:** Type hinting rules.
- **Common Python Best Practices:** Avoiding unnecessary loops, optimizing string operations, etc.

When Ruff is run, it performs **static code analysis**, meaning it does not execute the code but examines its structure and syntax to detect potential errors.

MoSCoW principle

The **MoSCoW principle** is a **prioritization technique** used in software development, including Python projects, to classify requirements or coding standards based on their importance. The term **MoSCoW** stands for:

- **M**ust-Have
- **S**hould-Have
- **C**ould-Have
- **W**on’t-Have (for now)

This method is useful when applying **Python coding standards** because it helps developers **prioritize** best practices and ensure the most critical aspects of coding are implemented first.

1. Applying MoSCoW to Python Coding Standards

MoSCoW Category	Python Coding Standard Application
Must-Have (Essential)	Follow **PEP 8** (Python's official style guide), use meaningful variable names, maintain indentation, and avoid syntax errors.
Should-Have (Important but not critical)	Use **type hints**, follow **PEP 257** for docstrings, write unit tests, and handle exceptions properly.
Could-Have (Nice to have, but not mandatory)	Optimize performance, use **code linters** (e.g., Ruff, Flake8), follow **PEP 20 (Zen of Python)**, and use logging for debugging.
Won’t-Have (For Now)	Less crucial aspects like micro-optimizations, strict adherence to advanced formatting rules, or complex design patterns that aren’t necessary for the project.

Black code formatter :

Black is an **opinionated code formatter** for Python that automatically formats code to follow a **consistent style**. It is widely used in the Python community because it enforces **PEP 8 standards** while making code more readable and maintainable.

Black is known as the "**uncompromising code formatter**" because it automatically reformats code **without needing manual style decisions**, improving **code consistency** across teams.

2. Key Features of Black

Automatic Formatting: Reformats Python code **without manual intervention**.

PEP 8 Compliant: Ensures the code follows Python's official **style guide**.

Speed & Efficiency: Quickly reformats even large projects.

Consistent Style: Removes debates over code formatting within teams.

Stable and Reliable: Ensures that running Black multiple times on the same file doesn’t change anything further (idempotent).

3. How Black Works

Black reformats code by applying:

- **Consistent indentation** (4 spaces, no tabs).
- **Line length limit** (default **88 characters**, configurable).
- **Single quotes to double quotes** where necessary.
- **Proper spacing around operators and brackets**.
- **Ensuring functions and class definitions have proper newlines**.

Exception handling

Exception handling in Python is a mechanism to **handle runtime errors** and prevent programs from crashing unexpectedly. It allows the program to **gracefully recover** from errors and continue execution.

An **exception** is an error that occurs during the execution of a program. If not handled, it stops the program abruptly.

Handling Important?

Prevents program crashes due to unexpected errors.

Improves user experience by showing meaningful error messages.

Helps debug code by identifying and handling errors properly.

Ensures reliability in applications, especially in production environments.

Python provides four main keywords for handling exceptions:

Keyword	Purpose

`try`	Contains the code that may raise an exception.
`except`	Catches and handles exceptions.
`else`	Executes code **if no exceptions occur**.
`finally`	Executes code **no matter what happens** (used for cleanup).

6. Common Python Exceptions

Exception	Cause
`ZeroDivisionError`	Division by zero (`5 / 0`).
`TypeError`	Incorrect data type (`"5" + 2`).
`ValueError`	Invalid value for a function (`int("abc")`).
`IndexError`	Accessing an invalid list index (`my_list[10]`).
`KeyError`	Accessing a non-existent dictionary key (`my_dict["missing"]`).
`AttributeError`	Calling an invalid attribute on an object (`None.length`).
`FileNotFoundError`	Trying to open a non-existent file.

File handling:

File handling in Python allows programs to **read, write, update, and manipulate files** stored on a computer. Python provides built-in functions to work with files, making it easy to store and retrieve data.

Python supports handling **text files (`.txt`) and binary files (`.bin` , `.jpg` , `.pdf` , etc.)**.

Python provides several operations for file handling:

Operation	Description
Opening a file (`open()`)	Opens a file in different modes.
Reading a file (`read()`)	Reads the contents of a file.
Writing to a file (`write()`)	Writes data to a file.
Appending to a file (`append()`)	Adds new data to an existing file.
Closing a file (`close()`)	Closes an open file to free resources.

Deleting a file (`os.remove()`)	Deletes a file from the system.

Mode	Description
`'r'`	Read mode (default). Raises an error if the file does not exist.
`'w'`	Write mode. Creates a new file if it doesn't exist, or overwrites the existing file.
`'a'`	Append mode. Adds data to an existing file. Creates a new file if it doesn’t exist.
`'r+'`	Read and write mode. File must exist.
`'w+'`	Write and read mode. Overwrites existing content.
`'a+'`	Append and read mode. Retains content and adds new data.
`'rb'`, `'wb'`	Read/write binary files.
`'rb+'`, `'wb+'`	Read and write binary files.