**STRING MANIPULATION**

**🔸 What is a String in Python?**

A **string** is a sequence of characters — text enclosed in either:

* 'single quotes'
* "double quotes"

Python treats both the same.

**✅ Code Example 1: Defining Strings**

name1 = "Aditya"

name2 = 'DevOpsShack'

print(name1)

print(name2)

**🔍 Explanation:**

* name1 = "Aditya"  
  → We define a string variable named name1 and assign it the value "Aditya". This string uses **double quotes**.
* name2 = 'DevOpsShack'  
  → Same idea here, but with **single quotes**. Python accepts both.
* print(name1)  
  → This will output:

Aditya

* print(name2)  
  → This will output:

DevOpsShack

📌 Both are valid string declarations. Use either based on context.

**✅ Code Example 2: Using Quotes Inside Strings**

quote = "He said, 'Python is awesome!'"

print(quote)

**🔍 Explanation:**

* The outer string uses **double quotes**.
* The **inner single quotes** around 'Python is awesome!' are preserved and don’t confuse Python.
* Output:

He said, 'Python is awesome!'

💡 If you use the same type of quote inside and outside, Python will throw an error unless escaped.

**✅ Code Example 3: Escaping Characters**

message = 'It\'s a DevOps world'

print(message)

**🔍 Explanation:**

* 'It\'s a DevOps world' → the \' is an **escape sequence**.
* It allows you to insert a single quote inside a single-quoted string.
* Output:

It's a DevOps world

📌 Escape sequences start with a backslash (\) and tell Python to treat the next character **literally**.

**🔸 Common Escape Sequences**

| **Escape Code** | **Meaning** | **Example Output** |
| --- | --- | --- |
| \n | New Line | Line1\nLine2 → 2 lines |
| \t | Tab (horizontal) | Hello\tWorld |
| \\ | Literal backslash | C:\\Users\\Name |
| \' | Single quote | 'It\'s OK' → It's OK |
| \" | Double quote | "He said: \"Yes\"" |

**✅ Code Example 4: Newline and Tab**

info = "Name:\tAditya\nRole:\tTrainer"

print(info)

**🔍 Explanation:**

* \t = tab
* \n = newline (goes to next line)

So this prints:

Name: Aditya

Role: Trainer

💡 Tabs and newlines help format multi-line output or column alignment in logs.

**✅ Strings Are Sequences**

Python treats a string as a **sequence** (like a list of characters), so you can:

* Index them
* Slice them
* Iterate over them

**✅ Code Example 5: Indexing**

text = "DevOps"

print(text[0]) # First character

print(text[3]) # 4th character (indexing starts at 0)

**🔍 Explanation:**

* text[0] → accesses the first character: 'D'
* text[3] → accesses the fourth character: 'O'
* Output:

D

O

💡 Indexing starts from **0**, like most programming languages.

**✅ Code Example 6: Negative Indexing**

word = "Automation"

print(word[-1]) # Last character

print(word[-3]) # Third from last

**🔍 Explanation:**

* -1 → last character: 'n'
* -3 → third-last: 'i'
* Output:

n

i

Negative indexing is useful to access the **end of strings** without knowing their exact length.

**✅ Code Example 7: Slicing Strings**

tool = "Prometheus"

print(tool[0:4]) # First 4 letters

print(tool[4:]) # From 5th to end

print(tool[:5]) # First 5 letters

print(tool[-4:]) # Last 4 letters

**🔍 Explanation:**

* tool[0:4] → start at index 0, stop before 4 → 'Prom'
* tool[4:] → from index 4 to end → 'etheus'
* tool[:5] → from beginning up to index 5 → 'Prome'
* tool[-4:] → last 4 characters → 'heus'

📌 String slicing is non-inclusive at the end index (like range()).

**✅ String Length**

**✅ Code Example 8: len()**

s = "Vault"

print(len(s))

**🔍 Explanation:**

* len() returns the number of characters in the string.
* Output:

5

This includes **spaces**, **special characters**, etc.

**🔚 WRAP-UP: Part 1 Summary**

| **Concept** | **Example** | **Description** |
| --- | --- | --- |
| Create string | 'Hello' / "World" | Single or double quotes |
| Escape characters | \', \n, \t | Insert special characters |
| Indexing | s[0], s[-1] | Access characters by position |
| Slicing | s[:5], s[2:6] | Extract substrings |
| Length | len(s) | Count total characters |

**🔸 What Are String Methods?**

**String methods** are built-in functions in Python that you call **on string objects** to modify or analyze them.

They don’t change the original string (strings are **immutable**) — they return a **new** string.

**✅ SECTION A: Changing Case (upper(), lower(), title(), capitalize())**

**✅ Code Example 1: upper() and lower()**

text = "DevOps For Everyone"

print(text.upper())

print(text.lower())

**🔍 Explanation:**

* text.upper()  
  → Converts all characters to **uppercase**:

DEVOPS FOR EVERYONE

* text.lower()  
  → Converts all characters to **lowercase**:

devops for everyone

These are helpful when:

* Normalizing user input
* Doing case-insensitive comparison

**✅ Code Example 2: capitalize() and title()**

sentence = "devops is awesome"

print(sentence.capitalize())

print(sentence.title())

**🔍 Explanation:**

* capitalize()  
  → Makes only the **first letter** uppercase, rest lowercase:

Devops is awesome

* title()  
  → Capitalizes the **first letter of every word**:

Devops Is Awesome

Used in report titles, names, UI formatting, etc.

**✅ SECTION B: Removing Whitespace (strip(), lstrip(), rstrip())**

**✅ Code Example 3: strip() to remove leading/trailing spaces**

line = " Hello, DevOps! "

print(line.strip())

**🔍 Explanation:**

* strip() removes **both leading and trailing whitespace**
* Output:

Hello, DevOps!

**✅ Code Example 4: lstrip() and rstrip()**

line = ">>> Jenkins Build <<<"

print(line.lstrip("> ")) # remove from left

print(line.rstrip(" <")) # remove from right

**🔍 Explanation:**

* lstrip("> ")  
  → Removes > and space from the **start**
* rstrip(" <")  
  → Removes < and space from the **end**

Output:

Jenkins Build <<<

>>> Jenkins Build

**✅ SECTION C: Splitting and Joining Strings**

**✅ Code Example 5: split()**

log = "ERROR:Disk Full:10.0.0.5"

parts = log.split(":")

print(parts)

**🔍 Explanation:**

* split(":") → breaks the string wherever : is found
* Returns a **list**:

['ERROR', 'Disk Full', '10.0.0.5']

Used often in log parsing and CSV reading.

**✅ Code Example 6: join()**

words = ["DevOps", "is", "awesome"]

sentence = " ".join(words)

print(sentence)

**🔍 Explanation:**

* " ".join(words) → Joins list elements using " " as a separator
* Output:

DevOps is awesome

You can use any separator (-, ,, |, etc.)

**✅ SECTION D: Searching in Strings (in, find(), startswith(), endswith())**

**✅ Code Example 7: Using in**

log = "WARNING: CPU usage high"

if "WARNING" in log:

print("⚠️ Warning Found")

**🔍 Explanation:**

* "WARNING" in log → Checks if substring exists
* Outputs:

⚠️ Warning Found

✅ Simple and readable. Returns True or False.

**✅ Code Example 8: find()**

email = "admin@devopsshack.com"

index = email.find("@")

print(index)

**🔍 Explanation:**

* find() returns the **index** of the first match of a substring
* If not found, returns -1
* Output:

5

Useful for splitting username and domain.

**✅ Code Example 9: startswith() and endswith()**

file = "config.yaml"

print(file.endswith(".yaml")) # True

print(file.startswith("config")) # True

**🔍 Explanation:**

* endswith(".yaml") → Checks if string ends with .yaml
* startswith("config") → Checks prefix

✅ Ideal for validating filenames, commands, etc.

**✅ SECTION E: Replace and Count**

**✅ Code Example 10: replace()**

message = "The server is down"

new\_msg = message.replace("down", "up")

print(new\_msg)

**🔍 Explanation:**

* replace("down", "up") → Returns a new string with the **first** replaced by the **second**
* Output:

The server is up

Does **not** modify original string.

**✅ Code Example 11: count()**

note = "Kubernetes is scalable. Kubernetes is flexible."

print(note.count("Kubernetes"))

**🔍 Explanation:**

* count("Kubernetes") → returns **how many times** the word appears
* Output:

2

**✅ WRAP-UP: Methods Covered So Far**

| **Method** | **Description** |
| --- | --- |
| upper(), lower() | Change text case |
| capitalize(), title() | First word or each word capitalization |
| strip(), lstrip(), rstrip() | Trim spaces or characters |
| split(), join() | Break or build string lists |
| in, find(), startswith(), endswith() | Searching tools |
| replace(), count() | Edit or count strings |

**✅ SECTION A: Why Format Strings?**

Formatting strings is crucial for:

* **Displaying structured data**
* **Generating logs**
* **Aligning output**
* **Injecting variables into templates**

**✅ SECTION B: Using f-strings (Recommended in Python 3.6+)**

**✅ Code Example 1: Injecting Variables into Strings**

name = "Aditya"

tool = "Jenkins"

print(f"{name} is using {tool}")

**🔍 Explanation:**

* The f before the string tells Python to interpret **variables inside {}**.
* {name} is replaced by "Aditya"
* {tool} is replaced by "Jenkins"

✅ Output:

Aditya is using Jenkins

🧠 f-strings are fast and readable. Use them instead of .format() or %.

**✅ Code Example 2: Inline Calculations**

a = 7

b = 3

print(f"{a} + {b} = {a + b}")

**🔍 Explanation:**

* {a + b} performs the addition **inside the string**
* Output:

7 + 3 = 10

**✅ SECTION C: Using .format() (older but still valid)**

**✅ Code Example 3: .format() with Position**

template = "Service: {}, Status: {}"

print(template.format("Vault", "Running"))

**🔍 Explanation:**

* {} are placeholders.
* .format("Vault", "Running") fills them in **left to right**
* Output:

Service: Vault, Status: Running

**✅ Code Example 4: .format() with Named Arguments**

template = "Tool: {tool}, Version: {ver}"

print(template.format(tool="Prometheus", ver="2.47.1"))

**🔍 Explanation:**

* {tool} and {ver} are **named placeholders**
* .format(tool="Prometheus", ver="2.47.1") injects values
* Output:

Tool: Prometheus, Version: 2.47.1

**✅ SECTION D: Alignment, Width, and Padding**

**✅ Code Example 5: Aligning Output**

service = "Grafana"

status = "Running"

print(f"{service:<10} → {status:>10}")

**🔍 Explanation:**

* {service:<10} → Left-align the word in **10-character width**
* {status:>10} → Right-align in **10-character width**
* Output:

Grafana → Running

💡 Helps build aligned tables/log output.

**✅ Code Example 6: Padding Numbers with Zeros**

build\_number = 42

print(f"Build #{build\_number:04}")

**🔍 Explanation:**

* :04 means **pad to 4 digits with leading zeros**
* Output:

Build #0042

Used in filenames like build\_0005.log, ticket numbers, etc.

**✅ SECTION E: Multiline Strings**

**✅ Code Example 7: Triple Quotes for Multiline**

msg = """Hello Team,

Deployment completed successfully.

Regards,

DevOps Shack

"""

print(msg)

**🔍 Explanation:**

* """ allows multi-line strings
* Preserves formatting as written
* Output:

Hello Team,

Deployment completed successfully.

Regards,

DevOps Shack

✅ Great for logs, templates, emails.

**✅ SECTION F: Escape Sequences Refresher**

| **Escape** | **Meaning** | **Example Result** |
| --- | --- | --- |
| \n | New line | Splits line |
| \t | Horizontal tab | Adds space |
| \' | Single quote inside ' | 'It\'s OK' → It's OK |
| \" | Double quote inside " | "He said: \"Go\"" |
| \\ | Literal backslash | \\n → shows \n |

**✅ Code Example 8: Combining Escape + Format**

username = "devops"

role = "admin"

print(f"User:\t{username}\nRole:\t{role}")

**🔍 Explanation:**

* \t adds **tab** spacing
* \n moves to next line

Output:

User: devops

Role: admin

**✅ WRAP-UP: What You Mastered in This Part**

| **Feature** | **Example** |
| --- | --- |
| f"{var}" | Insert variable inline |
| .format() | Template-based string injection |
| Width, padding | f"{x:>10}", f"{num:04}" |
| Escape sequences | \n, \t, \\ |
| Multiline strings | """...""" |

**✅ SECTION A: Looping Through Strings (Character by Character)**

A string is **iterable**, so you can loop through each character using a for loop.

**✅ Code Example 1: Print Each Character**

text = "DevOps"

for char in text:

print(char)

**🔍 Explanation:**

* text = "DevOps"  
  → We define a string to loop through.
* for char in text:  
  → Python takes each **character one by one** and assigns it to char.
* print(char)  
  → Prints each character on a **separate line**

✅ Output:

D

e

v

O

p

s

**✅ SECTION B: Count Characters Using a Loop**

**✅ Code Example 2: Count vowels**

sentence = "Kubernetes is powerful"

vowels = "aeiou"

count = 0

for letter in sentence.lower():

if letter in vowels:

count += 1

print("Vowel Count:", count)

**🔍 Explanation:**

* sentence.lower()  
  → We lowercase the entire sentence for case-insensitive matching.
* if letter in vowels:  
  → Checks if the current character is a vowel.
* count += 1  
  → Increments the count if the condition is true.

✅ Output:

Vowel Count: 8

**✅ SECTION C: Remove/Skip Certain Characters**

**✅ Code Example 3: Remove Digits from a String**

log = "Vault123 is at port 8200"

cleaned = ""

for ch in log:

if not ch.isdigit():

cleaned += ch

print(cleaned)

**🔍 Explanation:**

* ch.isdigit()  
  → Returns True if the character is a digit (0–9)
* if not ch.isdigit()  
  → Keep only **non-digit** characters
* cleaned += ch  
  → Appends the allowed character to a new string

✅ Output:

Vault is at port

**✅ SECTION D: Masking Sensitive Data**

**✅ Code Example 4: Masking Everything Except Last 4 Characters**

token = "sk\_live\_abc123xyz456"

masked = "\*" \* (len(token) - 4) + token[-4:]

print("Masked Token:", masked)

**🔍 Explanation:**

* len(token) - 4  
  → Number of characters to mask
* "\*" \* n  
  → Create a string of asterisks of length n
* token[-4:]  
  → Get last 4 characters
* Combine them using +

✅ Output:

Masked Token: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*3456

🧠 This is a real-world pattern used in logs, dashboards, etc.

**✅ SECTION E: Filtering and Replacing in Loops**

**✅ Code Example 5: Replace All Spaces with Hyphens**

name = "DevOps For Life"

result = ""

for ch in name:

if ch == " ":

result += "-"

else:

result += ch

print(result)

**🔍 Explanation:**

* If the character is a space, add -
* Otherwise, keep the original character

✅ Output:

DevOps-For-Life

✅ Equivalent to name.replace(" ", "-"), but shows how to do it manually

**✅ SECTION F: Tokenizing with .split() + Loop**

**✅ Code Example 6: Count Words in Sentence**

sentence = "DevOps is scalable and powerful"

words = sentence.split()

count = 0

for word in words:

count += 1

print("Word Count:", count)

**🔍 Explanation:**

* .split() splits string by whitespace → returns a list of words
* Looping over that list counts how many words there are

✅ Output:

Word Count: 5

**✅ SECTION G: Real Use Case – Log Sanitizer**

**✅ Code Example 7: Redact Tokens in Logs**

log = "Vault Token: s.x3y2z7abcd"

parts = log.split(": ")

if "s." in parts[1]:

parts[1] = "[REDACTED]"

new\_log = ": ".join(parts)

print(new\_log)

**🔍 Explanation:**

* log.split(": ") → separates Vault Token and the actual token
* if "s." in parts[1] → checks if it looks like a token (Vault tokens start with s.)
* parts[1] = "[REDACTED]" → hide the sensitive value
* ": ".join(parts) → reassemble the string

✅ Output:

Vault Token: [REDACTED]

This is a DevSecOps logging best practice.

**✅ SECTION H: Case-Insensitive Comparison**

**✅ Code Example 8: Normalize and Match**

env = "PROD"

if env.lower() == "prod":

print("✅ Production Environment")

**🔍 Explanation:**

* .lower() ensures we compare lowercase → avoids case mismatch
* Helps with user input, YAML parsing, config matching

✅ Output:

✅ Production Environment

**🔚 WRAP-UP: Part 4 Summary**

| **Pattern Used** | **Use Case** |
| --- | --- |
| Loop over string | Analyze/match every character |
| Conditions with if | Skip or process specific chars |
| Building new string | Manual masking or replacement |
| split() + for loop | Tokenization, word count |
| .isdigit(), .lower() | Validation/cleaning |

**✅ SECTION A: What Is Regex?**

**Regular Expressions (regex)** are patterns used to **match strings**.  
Python provides regex support via the built-in re module.

**📦 Import Syntax:**

import re

**✅ SECTION B: Simple Pattern Matching with re.search()**

**✅ Code Example 1: Match a Keyword in Text**

import re

log = "Error: Connection failed at 10.0.0.12"

match = re.search(r"Error", log)

if match:

print("❌ Error Found")

**🔍 Explanation:**

* re.search() looks for the **first match** of "Error" in the log.
* r"Error" is a **raw string literal**, used to write regex patterns without needing to escape backslashes.
* If a match is found, match is not None.

✅ Output:

❌ Error Found

**✅ SECTION C: Extract IP Address with re.search()**

**✅ Code Example 2: Extract IPv4 from String**

import re

log = "Host connected to 192.168.56.10 at 08:00"

match = re.search(r"\d{1,3}(\.\d{1,3}){3}", log)

if match:

print("IP Address:", match.group())

**🔍 Explanation:**

* \d{1,3} → matches 1 to 3 digits
* (\.\d{1,3}){3} → matches three .xxx parts (e.g., .168, .56, .10)
* Together → full IPv4 match
* match.group() returns the **matched string**

✅ Output:

IP Address: 192.168.56.10

✅ Real-world use: extracting server IPs from logs, alerts, etc.

**✅ SECTION D: Find All Matches with re.findall()**

**✅ Code Example 3: Find All Email Addresses**

import re

text = "Send to dev@shack.com or admin@domain.org"

emails = re.findall(r"[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+", text)

print("Emails:", emails)

**🔍 Explanation:**

* The regex matches:
  + [a-zA-Z0-9\_.+-]+ → email username part
  + @ → literal at sign
  + [a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+ → domain name and extension
* findall() returns **all matching substrings** as a list.

✅ Output:

Emails: ['dev@shack.com', 'admin@domain.org']

**✅ SECTION E: Replace Sensitive Data with re.sub()**

**✅ Code Example 4: Mask Token in Vault Log**

import re

log = "Vault Token: s.XYZ123secretvalue"

masked\_log = re.sub(r"s\.[A-Za-z0-9]+", "s.[REDACTED]", log)

print(masked\_log)

**🔍 Explanation:**

* s\.[A-Za-z0-9]+ → matches s. followed by alphanumeric characters
* re.sub() replaces that token with s.[REDACTED]

✅ Output:

Vault Token: s.[REDACTED]

✅ Real-world use: sanitizing logs before output/email.

**✅ SECTION F: Extract Key-Value Pairs from Logs**

**✅ Code Example 5: Parse Key=Value from a Log**

import re

log = "env=prod region=ap-south-1 instance=i-123456"

pairs = re.findall(r"(\w+)=([\w-]+)", log)

for key, value in pairs:

print(f"{key} → {value}")

**🔍 Explanation:**

* (\w+)=([\w-]+) → captures key=value format:
  + \w+ matches word characters (letters, digits, \_)
  + [\w-]+ allows - in values
* findall() returns a list of tuples: [("env", "prod"), ...]

✅ Output:

env → prod

region → ap-south-1

instance → i-123456

✅ Use case: config parsing, logging middleware, CI/CD metadata extraction.

**✅ SECTION G: Greedy vs. Non-Greedy Matching**

**✅ Code Example 6: Extract Only First Tag (Non-Greedy)**

import re

html = "<tag>Hello</tag><tag>World</tag>"

greedy = re.search(r"<tag>.\*</tag>", html)

non\_greedy = re.search(r"<tag>.\*?</tag>", html)

print("Greedy:", greedy.group())

print("Non-Greedy:", non\_greedy.group())

**🔍 Explanation:**

* .\* → greedy: grabs as much as possible
* .\*? → non-greedy: grabs the **first** match only

✅ Output:

Greedy: <tag>Hello</tag><tag>World</tag>

Non-Greedy: <tag>Hello</tag>

✅ Important in XML/HTML parsing or parsing logs with delimiters.

**✅ WRAP-UP: Summary of Regex Techniques**

| **Function** | **Use Case** |
| --- | --- |
| re.search() | First match + .group() |
| re.findall() | List of all matches |
| re.sub() | Replace pattern with something |
| re.match() | Match must be at string start |
| () | Capture groups |
| .\*, .\*? | Greedy / non-greedy wildcard |
| \d, \w, . | Digit, word char, any char |

**✅ SECTION A: Redacting Secrets from .env Files**

Imagine you’re working with a .env file:

DATABASE\_URL=postgres://user:password@host/db

API\_KEY=sk\_live\_12345abcd

We want to mask sensitive data but **retain structure**.

**✅ Code Example 1: Redact All Secrets**

import re

env\_data = """

DATABASE\_URL=postgres://user:password@host/db

API\_KEY=sk\_live\_12345abcd

"""

# Replace sensitive parts after '='

sanitized = re.sub(r'=(.+)', '=\*\*\*\*\*\*', env\_data)

print(sanitized)

**🔍 Explanation:**

* =(.+) → regex that matches = followed by **any content**
* re.sub(..., '=\*\*\*\*\*\*', ...) → replaces values with stars

✅ Output:

DATABASE\_URL=\*\*\*\*\*\*

API\_KEY=\*\*\*\*\*\*

💡 Can be modified to only redact tokens selectively (sk\_ pattern etc.)

**✅ SECTION B: Parse Logs for Error Codes and Messages**

Let’s say we have DevOps logs like:

[ERROR] Code 503: Service Unavailable

[INFO] Server started

[ERROR] Code 401: Unauthorized

We want to extract all error **codes + messages**.

**✅ Code Example 2: Extract Errors from Logs**

import re

logs = """

[ERROR] Code 503: Service Unavailable

[INFO] Server started

[ERROR] Code 401: Unauthorized

"""

matches = re.findall(r"Code (\d+): (.+)", logs)

for code, message in matches:

print(f"Error {code} → {message}")

**🔍 Explanation:**

* Code (\d+): (.+):
  + (\d+) captures the error code
  + (.+) captures the message after the colon

✅ Output:

Error 503 → Service Unavailable

Error 401 → Unauthorized

✅ Real-world use: dashboards, monitoring, Slack alert parsers

**✅ SECTION C: Trim and Clean CSV Data**

Let's say you read CSV data that looks dirty:

"name ", "email ", " age"

" Alice", " alice@xyz.com", " 23"

**✅ Code Example 3: Strip All Fields**

raw = '"name ", "email ", " age"\n" Alice", " alice@xyz.com", " 23"'

rows = raw.split("\n")

for row in rows:

fields = [f.strip(' "') for f in row.split(',')]

print(fields)

**🔍 Explanation:**

* split('\n') → separates rows
* split(',') → separates columns
* f.strip(' "') → removes both **quotes and spaces** from both ends

✅ Output:

['name', 'email', 'age']

['Alice', 'alice@xyz.com', '23']

**✅ SECTION D: Slicing Real Tokens (e.g., AWS ARN)**

arn:aws:iam::123456789012:user/DevOpsShack

Let’s extract the **username** at the end.

**✅ Code Example 4: Slice from ARN**

arn = "arn:aws:iam::123456789012:user/DevOpsShack"

username = arn.split("/")[-1]

print("Username:", username)

**🔍 Explanation:**

* split("/") splits at every /
* [-1] gets the **last** part — the username

✅ Output:

Username: DevOpsShack

✅ Used often in IAM parsing, AWS log parsing

**✅ SECTION E: Detect Secrets in Source Code**

Let’s scan a Python file and detect any line that contains potential secrets:

**✅ Code Example 5: Find Secrets in .py Files**

import re

source = """

vault\_token = "s.x123y456z789"

debug = True

aws\_secret = "AKIAIOSFODNN7EXAMPLE"

"""

matches = re.findall(r'["\'](s\.[\w\d]+|AKIA[\w\d]+)["\']', source)

for secret in matches:

print("🔒 Secret Found:", secret)

**🔍 Explanation:**

* Pattern matches:
  + Vault-style tokens: s.xxxxx
  + AWS keys: AKIAxxxxxxx
* Uses alternation | to match either pattern
* Wraps both with ["\']...["\'] to confirm it's a string

✅ Output:

🔒 Secret Found: s.x123y456z789

🔒 Secret Found: AKIAIOSFODNN7EXAMPLE

✅ Use case: local gitleaks/trivy checks before commit

**✅ SECTION F: Normalize CLI Input (Trimming + Lowercase + Validation)**

**✅ Code Example 6: Parse User CLI Input Safely**

raw\_input = " DePLoyMent "

clean = raw\_input.strip().lower()

if clean == "deployment":

print("✅ Starting deployment pipeline...")

else:

print("❌ Invalid option")

**🔍 Explanation:**

* strip() removes spaces
* lower() standardizes case
* Useful for CLI tools, shell wrappers, user prompts

✅ Output:

✅ Starting deployment pipeline...

**✅ SECTION G: Multi-line Matching (DOTALL flag)**

**✅ Code Example 7: Match Block of Text**

import re

doc = """

BEGIN

This is a

multiline string

END

"""

match = re.search(r"BEGIN(.\*)END", doc, re.DOTALL)

if match:

print("Content:", match.group(1).strip())

**🔍 Explanation:**

* re.DOTALL allows . to match **newlines**
* .group(1).strip() gives just the content between markers

✅ Output:

Content: This is a

multiline string

✅ Real-world: YAML blocks, Kubernetes manifests, config parsing

**🔚 FINAL WRAP-UP: MASTERING STRING MANIPULATION**

**✅ You have now learned:**

| **Skill** | **Use Case** |
| --- | --- |
| f-strings and formatting | CLI, logs, output templates |
| Escaping and multiline strings | Message formatting, scripts |
| Loops and slicing | Filtering characters, tokenizing |
| Regex matching (search, findall) | Log parsing, validation, secret scan |
| Real-world redaction + replacements | DevSecOps, observability, security |
| Cleanup + structure parsing | .env, .csv, .py, .yaml, .json |