স্ট্রিং এবং স্ট্রিং ম্যানিপুলেশন

■ Strings

- Strings in Python are sequences of characters enclosed within single (`' '`), double (`" "`), or triple quotes (`''' '''` or `""" """`).
- They are immutable, meaning they cannot be changed once created.

```
# Single quotes
string1 = 'Hello, World!'

# Double quotes
string2 = "Hello, World!"

# Triple quotes
string3 = '''Hello,
World!'''

# Triple quotes can span multiple lines
string4 = """Hello,
World!"""
```

■ Single Quotes (`' '`)

- Used to create string literals.
- Typically used for short strings or when the string itself contains double quotes.
- Can be escaped using a backslash (`\`).
- Best for short strings, especially when the string contains double quotes.

```
single_quote_str = 'Hello, World!'
print(single_quote_str)  # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = 'He said, "Hello, World!"'
print(quote_in_str)  # Output: He said, "Hello, World!" - no need to escape since the inner quotes are
double quotes.

# Using single quotes inside the string with escaping
escaped_quote_in_str = 'He said, \'Hello, World!\''
print(escaped_quote_in_str)  # Output: He said, 'Hello, World!' - escaped using a backslash (\).
```

■ Double Quotes (`" "`)

- Also used to create string literals.
- Preferred when the string contains single quotes to avoid escaping.
- Can be escaped using a backslash (`\`).
- Best for short strings, especially when the string contains single quotes

```
double_quote_str = "Hello, World!"
print(double_quote_str)  # Output: Hello, World!

# Using single quotes inside the string
quote_in_str = "It's a wonderful day!"
print(quote_in_str)  # Output: It's a wonderful day! - no need to escape since the inner quotes are
single quotes.

# Using double quotes inside the string with escaping
escaped_quote_in_str = "He said, \"Hello, World!\""
print(escaped_quote_in_str)  # Output: He said, "Hello, World!" - escaped using a backslash (\).
```

■ Triple Single Quotes (`''' '''`)

- Used for multi-line strings or docstrings.
- Can contain both single and double quotes without escaping.
- Preserves the formatting, including line breaks and indentation.
- Ideal for multi-line strings and when the string contains both single and double quotes

```
triple_single_quote_str = '''This is a string
that spans multiple lines.
It can contain both "double quotes" and 'single quotes' without escaping.'''
print(triple_single_quote_str)

# Output:
# This is a string
# that spans multiple lines.
# It can contain both "double quotes" and 'single quotes' without escaping.
```

■ Triple Double Quotes (`""" """`)

- Functionally identical to triple single quotes.
- Often used for docstrings (multi-line comments) in functions, classes, and modules.
- Preserves the formatting, including line breaks and indentation.
- Also ideal for multi-line strings and commonly used for docstrings

```
triple_double_quote_str = """This is another string
that spans multiple lines.
It also can contain both "double quotes" and 'single quotes' without escaping."""
print(triple_double_quote_str)

# Output:
# This is another string
# that spans multiple lines.
# It also can contain both "double quotes" and 'single quotes' without escaping.
```

■ String Indexing

Positive Indexing

- Starts from `0` and goes up to `len(string) - 1`.
- Index `0` refers to the first character, index `1` to the second character, and so on.

```
text = "Hello, World!"
print(text[0])  # Output: 'H'  (first character)
print(text[7])  # Output: 'W'  (eighth character)
```

Negative Indexing

- Starts from `-1` and goes backwards from the end of the string.
- Index `-1` refers to the last character, index `-2` to the second last character, and so on.

```
text = "Hello, World!"
print(text[-1])  # Output: '!'  (last character)
print(text[-2])  # Output: 'd'  (second last character)
```

String Indexing Use Case

- **Extracting Substrings**: Retrieve specific parts of a string, such as a substring or a single character.
- **Reversing Strings**: Access characters in reverse order.
- **Manipulating User Input**: Modify or analyze parts of user-provided strings, like form inputs.
- **Parsing Data**: Extract specific fields from structured data formats.
- **Validation and Formatting**: Check and adjust the format of strings, such as dates or IDs.

■ String Slicing

String slicing in Python allows you to extract a portion of a string using a colon (`:`) syntax. The basic form of slicing is `string[start:stop:step]`, where:

- `start` is the index where the slice starts (inclusive).
- `stop` is the index where the slice ends (exclusive).
- `step` determines the step size or the increment between each index.

Here are the detailed examples based on the given string `text = "Hello, World!"`:

- Extracts a Substring from Index 0 to 4

```
text = "Hello, World!"
print(text[0:5])
```

- Extracts from Index 7 to the End

```
print(text[7:]) # Output: 'World!'
```

- Extracts from the Start to Index 4

```
print(text[:5])
```

- Extracts Every Second Character

```
print(text[::2])
```

- Reverses the String

```
print(text[::-1])
```

- Extracting a Substring with a Specific Step

```
text = "Hello, World!"
# Extract every third character starting from index 0
print(text[0::3])  # Output: 'Hl r!'
```

- Extracting a Substring from the Middle

```
text = "Hello, World!"
# Extract substring from index 3 to 8
print(text[3:8])  # Output: 'lo, W'
```

Slicing Use Case

- **Extracting Substrings**: Retrieve specific parts of a string, such as words or sentences.
- **Reversing Strings**: Easily reverse the entire string or specific parts of it.
- **Formatting Strings**: Modify parts of a string to fit a certain format or extract meaningful data.
- **Analyzing Data**: Extract specific fields from structured data formats like dates or file paths.
- **Cleaning Data**: Remove unwanted parts of a string or reformat it.

■ String Concatenation

String concatenation is the process of combining two or more strings into one. In Python, this can be done using the `+` operator.

Using the `+` operator:

```
string1 = "Hello"
string2 = "World"
combined = string1 + ", " + string2 + "!"
print(combined)  # Output: Hello, World!
```

Using `join()` method:

```
string1 = "Hello"
string2 = "World"
combined = ", ".join([string1, string2]) + "!"
print(combined)  # Output: Hello, World!
```

Using formatted string literals (f-strings) (Python 3.6+):

```
string1 = "Hello"
string2 = "World"
```

```
combined = f"{string1}, {string2}!"
print(combined)  # Output: Hello, World!
```

Using the `format()` method:

```
string1 = "Hello"
string2 = "World"
combined = "{}, {}!".format(string1, string2)
print(combined)  # Output: Hello, World!
```

Using `%` formatting:

```
string1 = "Hello"
string2 = "World"
combined = "%s, %s!" % (string1, string2)
print(combined)  # Output: Hello, World!
```

String Concatenation Use Case

- **Building Dynamic Messages**: Combine strings to create dynamic text for user messages or logs.
- **URL Construction**: Assemble URLs from different parts, such as base URLs and query parameters.
- **File Paths**: Construct file paths by combining directory names and file names.
- **Template Strings**: Create templates by merging fixed text with dynamic data.
- **Data Formatting**: Combine multiple pieces of data into a formatted string for display or storage.

■ String Repetition

String repetition is the process of repeating a string a specified number of times. This can be done using the `*` operator.

```
# Defining a string
repeat_str = "Hello! "

# Repeating the string 3 times
repeat = repeat_str * 3

# Printing the repeated string
print(repeat)  # Output: 'Hello! Hello! Hello! '
```

String Repetition Use Case

- **Generating Patterns**: Create repeated patterns or borders for text-based interfaces or displays.
- **Formatting Output**: Repeat characters or strings to format output consistently, like underlining headings.
- **Initialization**: Quickly initialize a string with repeated characters for placeholders or data preparation.
- **Creating Repeated Messages**: Generate repeated warning or notification messages for emphasis.
- **Visual Separators**: Use repeated strings as visual separators in logs or reports.

■ String Methods

```
# Define a string for demonstration
text = "hello world"


# Convert to uppercase
print("Uppercase:", text.upper())  # Output: 'HELLO WORLD'


# Convert to lowercase
text = "HELLO WORLD"
print("Lowercase:", text.lower())  # Output: 'hello world'


# Capitalize the first letter
text = "hello world"
print("Capitalize:", text.capitalize())  # Output: 'Hello world'


# Title case (capitalize first letter of each word)
print("Title case:", text.title())  # Output: 'Hello World'


# Swap case (invert case of each letter)
text = "Hello World"
print("Swap case:", text.swapcase())  # Output: 'hELLO wORLD'


# Replace a substring
text = "hello world"
print("Replace:", text.replace("world", "Python"))  # Output: 'hello Python'


# Split the string into a list
text = "hello-world"
words = text.split("-") # Splits on hyphen print(words) # Output: ['hello', 'world']


# Join a list into a string
words = ['hello', 'world']
print("Join:", ' '.join(words))  # Output: 'hello world'


# Strip whitespace from both ends
text = "   hello world   "
print("Strip:", text.strip())  # Output: 'hello world'


# Remove leading whitespace
print("Left strip:", text.lstrip())  # Output: 'hello world   '


# Remove trailing whitespace
print("Right strip:", text.rstrip())  # Output: '   hello world'


# Check if string starts with a substring
text = "hello world"
print("Starts with 'hello':", text.startswith("hello"))  # Output: True
```

```
# Check if string ends with a substring
print("Ends with 'world':", text.endswith("world"))  # Output: True


# Find the position of a substring
print("Find 'world':", text.find("world"))  # Output: 6


# Count occurrences of a substring
print("Count 'o':", text.count("o"))  # Output: 2


# Check if all characters are alphanumeric
print("Is alphanumeric:", text.isalnum())  # Output: False


# Check if all characters are alphabetic
text = "hello"
print("Is alphabetic:", text.isalpha())  # Output: True


# Check if all characters are digits
text = "12345"
print("Is digit:", text.isdigit())  # Output: True


# Check if the string contains only whitespace
text = "   "
print("Is whitespace:", text.isspace())  # Output: True


# Check if the string is titlecased
text = "Hello World"
print("Is titlecased:", text.istitle())  # Output: True


# Example of combining methods
# Capitalizing each word in a sentence
sentence = "this is a sample sentence."
capitalized_sentence = sentence.title()
print("Capitalized sentence:", capitalized_sentence)  # Output: 'This Is A Sample Sentence.'


# Removing extra spaces and converting to uppercase
text = "   hello world   "
cleaned_text = text.strip().upper()
print("Cleaned and uppercase:", cleaned_text)  # Output: 'HELLO WORLD'
```

String Methods Practical Use Case

- **Data Cleaning**: Remove unwanted characters, trim whitespace, and standardize text formats.
- **Text Analysis**: Count occurrences, find substrings, and analyze text content.
- **User Input Processing**: Validate and sanitize user inputs from forms or other sources.
- **Formatting Output**: Prepare and format strings for display or reporting.

- **Generating Dynamic Text**: Construct dynamic messages, URLs, or file paths based on variable data.