Python String Manipulation and File Handling

A Comprehensive Revision Session

Minor In AI, IIT Ropar
11th June, 2025

Welcome, budding AI enthusiasts!

Let's dive into two fundamental building blocks of Python that will become your constant companions as you build intelligent systems: handling text data and managing files.

Imagine you're working on an AI project that needs to read and analyze stories or articles. Where do these stories come from? Usually, they are stored in files on your computer. And what are stories made of? Text! Collections of letters, words, and sentences.

So, before we build complex AI models, we need to understand two crucial things:

1. How to work with and manipulate text itself (Strings).
2. How to read information from these story files and save new information back into them (File Handling).

This chapter is your revision session for these essential skills!

Part 1: Understanding and Working with Text (Strings)

In Python, text is represented using a data type called a **string**. Think of a string as a container holding a sequence of characters – anything from a single letter to a whole book chapter.

Unlike some other programming languages you might encounter, Python is quite flexible. It doesn't make a big fuss about whether you have a single character or many. Whether it's just `'A'` or `"Hello World"`, Python considers both of them strings.

```python
# Example of a single character string
char_a = 'A'
print(type(char_a))

# Example of a multi-character string
text_hello = "Hello World"
print(type(text_hello))
```

```
<class 'str'> <class 'str'>
```

Both outputs confirm they are indeed strings (`str`).

How to Write Down Strings in Python

You can enclose your text data in Python using either **single quotes (`'`)** or **double quotes (`"`)**. Python treats them exactly the same!

```python
# Using single quotes
my_string_single = 'This is a string in single quotes.'
print(my_string_single)

# Using double quotes
my_string_double = "This is a string in double quotes."
print(my_string_double)
```

```
This is a string in single quotes.  This is a string in double quotes.
```

This flexibility becomes super handy when your text *itself* contains quotes. For example, what if you want to store the phrase `"Let's party!"`? This phrase has a single quote in the word "Let's". If you try to enclose it in single quotes, Python gets confused, thinking the string ends after "Let".

```python
# Problematic example (will cause an error)
# lets_party = 'Let's party!'
# print(lets_party)
```

```
This would result in a SyntaxError because of the  stray single quote after 'Let'
```

The easiest fix? Use the *other* type of quote to enclose the string! Since the text contains a single quote, we'll use double quotes outside.

```python
# Using double quotes when the string contains a single quote
lets_party_correct = "Let's party!"
print(lets_party_correct)
```

```
Let's party!
```

Similarly, if your text contains double quotes (like someone speaking), you can enclose the string in single quotes.

An alternative, though slightly more technical way, is to use an **escape sequence**. You can tell Python to treat a special character (like a quote *within* a string enclosed by the same type of quote) as just plain text, not a string delimiter, by putting a backslash (\) before it.

```python
# Using escape sequence for the single quote
lets_party_escaped = 'Let\'s party!' # The backslash escapes the single quote
print(lets_party_escaped)

# Using escape sequence for a double quote
he_said_hi = "He said \"Hello!\" to me." # Escaping the double quotes inside
print(he_said_hi)
```

```
Let's party!  He said "Hello!" to me.
```

While escape sequences are powerful, using the alternate quote type is often simpler for this specific problem.

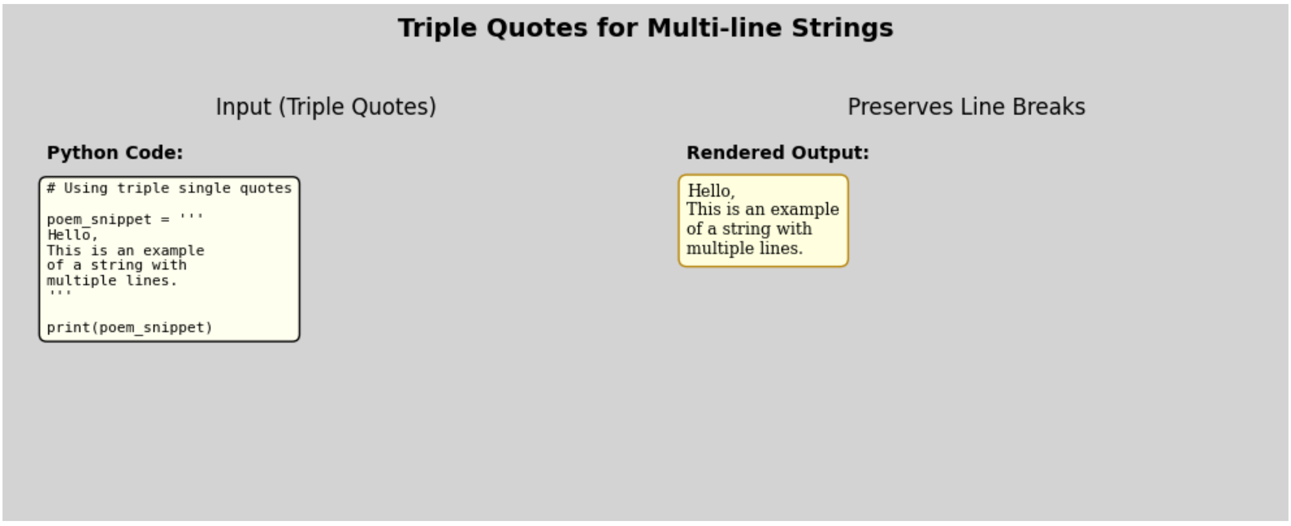
Strings Across Multiple Lines

Sometimes, you might have a long piece of text, like a paragraph or a poem, that spans several lines. For this, Python offers **triple quotes** (either three single quotes `'''...'''` or three double quotes `"""..."""`). These preserve the line breaks and formatting exactly as you type them.

```python
# Using triple single quotes for a multi-line string
poem_snippet = '''
Hello,
This is an example of a string
with multiple lines.
It will print in separate lines.
'''
print(poem_snippet)

# Using triple double quotes for a multi-line string
another_multiline = """
This is another
example of a string
with multiple lines.
See? It works too!
"""
print(another_multiline)
```

```
Hello, This is an example of a string with multiple lines.  It will print in separate
lines.
This is another example of a string with multiple lines.  See?  It works too!
```

Triple quotes are mainly for text that naturally has line breaks. For single-line text, single or double quotes are sufficient.



```python
# Using triple single quotes

poem_snippet = '''
Hello,
This is an example
of a string with
multiple lines.
'''

print(poem_snippet)
```

```
Hello,
This is an example
of a string with
multiple lines.
```

Figure 1: Triple quotes are perfect for preserving line breaks in text like poems or paragraphs.

Basic String Operations

Once you have strings, you can perform various actions or "operations" on them.

1. Finding the Length

How many characters are in a string? The built-in `len()` function tells you. Remember that spaces and punctuation count as characters too!

```python
my_string = "Hello World!" # Our example string
string_length = len(my_string)
print(f"The string is: '{my_string}'")
print(f"The length of the string is: {string_length}")
```

```
The string is:  'Hello World!'  The length of the string is:  12
```

2. Joining Strings (Concatenation)

You can stitch strings together using the + operator. This is called **concatenation**.

```python
part1 = "Hello"
part2 = " " # A space string
part3 = "World!"
full_string = part1 + part2 + part3
print(f"Concatenated string: {full_string}")

# You can concatenate any number of strings
greeting = "Hi" + ", " + "how" + " " + "are" + " " + "you?"
print(f"Another concatenated string: {greeting}")
```

```
Concatenated string:  Hello World!  Another concatenated string:  Hi, how are you?
```

Notice how Python understands + differently depending on what it's operating on. If it sees numbers, it adds them. If it sees strings, it concatenates them. This ability of an operator to behave differently based on the data type is sometimes called operator overloading (or a form of polymorphism).

3. Repeating Strings (Repetition)

Want to repeat a string multiple times? Use the multiplication operator `*` with an integer.

```python
repeat_word = "Python"
repeated_string = repeat_word * 3
print(f"Original word: {repeat_word}")
print(f"Repeated string: {repeated_string}")
```

```
Original word:  Python Repeated string:  PythonPythonPython
```

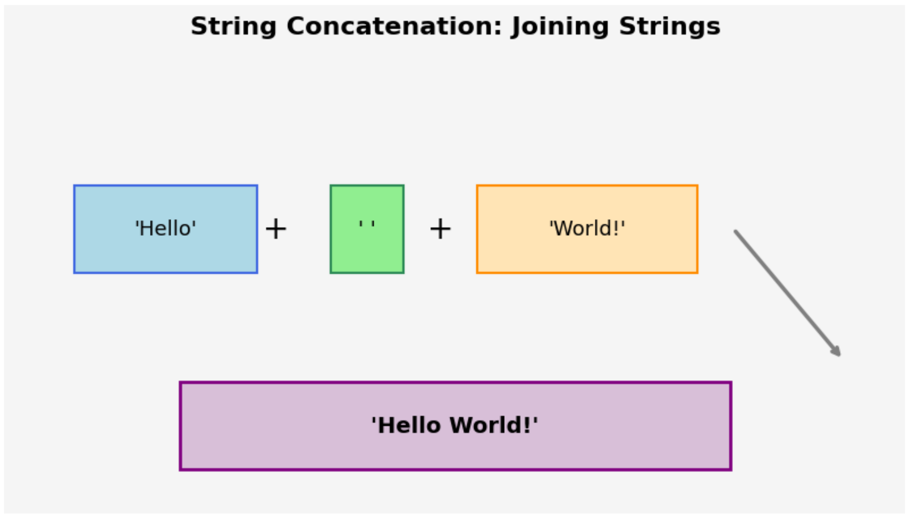


Figure 2: Concatenation joins strings end-to-end to create a new, longer string.

4. Accessing Individual Characters (Indexing)

Strings are sequences, meaning the characters are in a specific order. You can access a single character using its position, or **index**. Python uses **zero-based indexing**, meaning the first character is at index 0, the second at index 1, and so on. You use square brackets `[]` with the index number.

```python
my_string = "Hello World!"
print(f"String: '{my_string}'")
print(f"Character at index 0: {my_string[0]}") # The first character
print(f"Character at index 1: {my_string[1]}") # The second character
print(f"Character at index 4: {my_string[4]}") # The fifth character
print(f"Character at index 11: {my_string[11]}") # The last character
```

```
String: 'Hello World!' Character at index 0: H Character at index 1: e Character at
index 4: o Character at index 11: !
```

What happens if you ask for an index that doesn't exist? Just like asking for the 13th character in our 12-character string, Python will raise an error.

```python
# Trying to access an invalid index
# print(my_string[12]) # This will cause an error
```

```
Output would be: IndexError: string index out of range
```

Python also has a cool feature called **negative indexing**. This allows you to count from the *end* of the string. The last character is at index -1, the second to last at -2, and so on.

```python
my_string = "Hello World!"
print(f"String: '{my_string}'")
print(f"Character at index -1: {my_string[-1]}") # The last character
print(f"Character at index -2: {my_string[-2]}") # Second last character
print(f"Character at index -12: {my_string[-12]}") # First character
```

```
String: 'Hello World!' Character at index -1: ! Character at index -2: d Character
at index -12: H
```

5. Extracting Parts of a String (Slicing)

Indexing gets you a single character. **Slicing** gets you a *portion* or "slice" of the string. You use the square brackets again, but provide a range using colons `[:]`. The general syntax is `[start : end : step]`.

- `start`: The index where the slice begins (inclusive).
- `end`: The index where the slice *ends* (exclusive - the character at this index is *not* included).
- `step`: How many characters to skip between each character taken (default is 1).

```python
my_string = "Hello World!"
print(f"String: '{my_string}'")

# Slice from index 0 up to (but not including) index 5
slice1 = my_string[0:5]
print(f"Slice [0:5]: {slice1}")

# Omitting the start index (default is 0)
slice3 = my_string[:5]
print(f"Slice [:5]: {slice3}")

# Omitting the end index (default is until the end)
slice4 = my_string[6:]
print(f"Slice [6:]: {slice4}")
```

```
String: 'Hello World!' Slice [0:5]: Hello Slice [:5]: Hello Slice [6:]: World!
```

Strings Are Immutable

This is a very important property of strings in Python. **Immutable** means that once a string is created, you cannot change any of its characters directly. You can't modify it "in place".

Let's see what happens if we try to change a character:

```python
my_string = "Hello World!"
# Attempting to change the first character from 'H' to 'Z'
# my_string[0] = 'Z' # This will cause an error
```

```
Output would be: TypeError: 'str' object does not support item assignment
```

If you need a string with a different character, you must create a *new* string based on the old one, perhaps using slicing and concatenation.

```python
# How to "change" a character (by creating a new string)
my_string = "Hello World!"
new_string = 'Z' + my_string[1:] # Concatenate 'Z' with the rest
print(f"Original string: {my_string}")
print(f"New string: {new_string}")
```

```
Original string: Hello World! New string: Zello World!
```

Useful Built-in String Methods

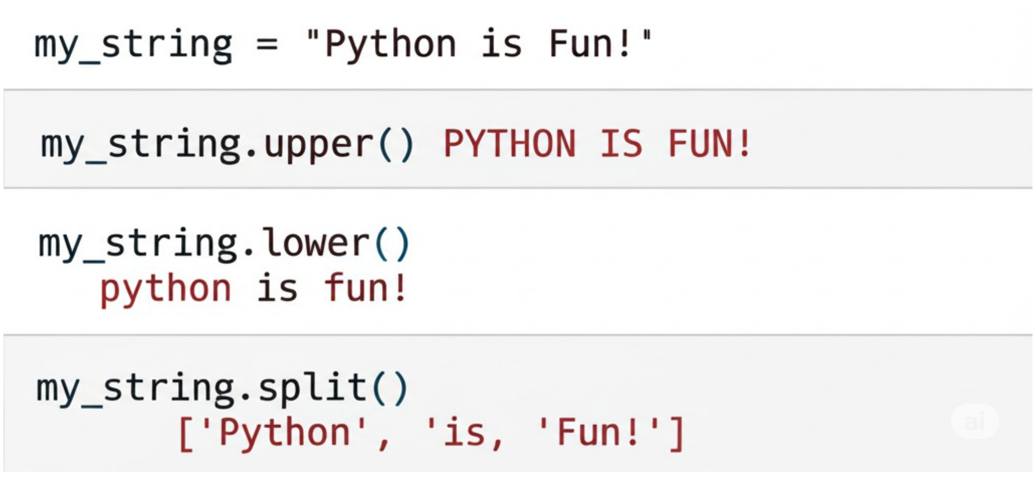
Strings in Python come with many pre-built functions called **methods**. You call them using a dot (.) after the string variable name.

```python
my_string = "Hello World!"

# Convert to uppercase
upper_string = my_string.upper()
print(f"Uppercase: {upper_string}")

# Convert to lowercase
lower_string = my_string.lower()
print(f"Lowercase: {lower_string}")
```

```
Uppercase: HELLO WORLD! Lowercase: hello world!
```

A few other useful methods:

- `.capitalize()`: Converts the first character to uppercase and the rest to lowercase.
- `.title()`: Converts the first character of *each word* to uppercase.
- `.split()`: Splits the string into a list of smaller strings based on a separator. This is extremely useful!
- `.index(substring)`: Finds the index of the *first* occurrence of a substring.
- `.count(substring)`: Counts how many times a substring appears.



```
my_string = "Python is Fun!"

my_string.upper() PYTHON IS FUN!

my_string.lower()
   python is fun!

my_string.split()
       ['Python', 'is, 'Fun!']
```

Figure 3: Exploring string methods in an interactive notebook can show you all the available tools.

Part 2: Storing Information Permanently (File Handling)

The primary function for working with files is `open()`. It takes two main arguments: the file name and the **mode**.

Writing to Files ('w')

The write mode, `'w'`, creates a new file or **erases all content** of an existing file before writing.

```python
# Open 'sample.txt' in write mode ('w')
# If it exists, its content is cleared. If not, it's created.
with open('sample.txt', 'w') as file_object:
    file_object.write("Hello World")
# The 'with' statement automatically closes the file
```

After running this, `sample.txt` will contain only "Hello World". If you run it again with different text, the old text will be gone.

Appending to Files ('a')

The append mode, `'a'`, adds content to the **end of the file** without erasing what's already there.

```python
# Let's reset the file first
with open('sample.txt', 'w') as f:
    f.write("Hello World\n") # \n is a newline character

# Now, open in append mode ('a') to add more text
with open('sample.txt', 'a') as f:
    f.write("Appending this line\n")
```

Now, `sample.txt` contains both lines. The `\n` ensures the new text starts on a new line.

Reading from Files ('r')

The read mode, `'r'`, is used to get information from an existing file. Trying to open a non-existent file in this mode will cause an error.

```python
# Open the file in read mode ('r')
with open('sample.txt', 'r') as file_object:
    # Method 1: read() - Reads the entire file into one string
    print("--- Using read() ---")
    full_content = file_object.read()
    print(full_content)

# To use other read methods, you must re-open the file
with open('sample.txt', 'r') as file_object:
    # Method 2: readlines() - Reads all lines into a list of strings
    print("--- Using readlines() ---")
    all_lines = file_object.readlines()
    print(all_lines)
```

```
-- Using read() -- Hello World Appending this line
-- Using readlines() -- ['Hello World', 'Appending this line']
```

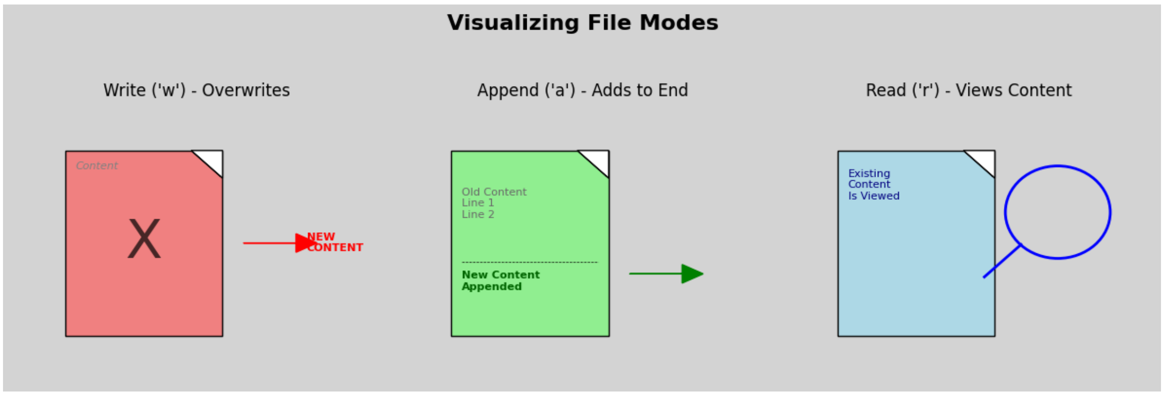


Figure 4: Visualizing the different file modes: Write (overwrite), Append (add), and Read (view).

Combining Read and Write Modes

There are also modes for combined operations:

- `'r+'`: Read and write. The file must exist. The pointer starts at the beginning.
- `'w+'`: Write and read. **Clears the file first**. Creates the file if it doesn't exist.
- `'a+'`: Append and read. Adds to the end. Creates the file if it doesn't exist.

The 'with' Statement: Best Practice

It's best practice to use the `with open(...) as ...:` syntax. It automatically handles closing the file for you, even if errors occur. This prevents data corruption and resource leaks.

```python
# The 'with' statement is the preferred way to handle files
with open('another_file.txt', 'w') as f:
    f.write("This is the safe and easy way to work with files.")
    # No need to call f.close(), it happens automatically!
```

We've covered the basics of working with text using Python strings and how to interact with files for permanent data storage. These skills are fundamental for nearly any AI task, from processing text data for natural language processing to saving model results.

Keep practicing these concepts, and they will become second nature!

Good luck with your AI journey!