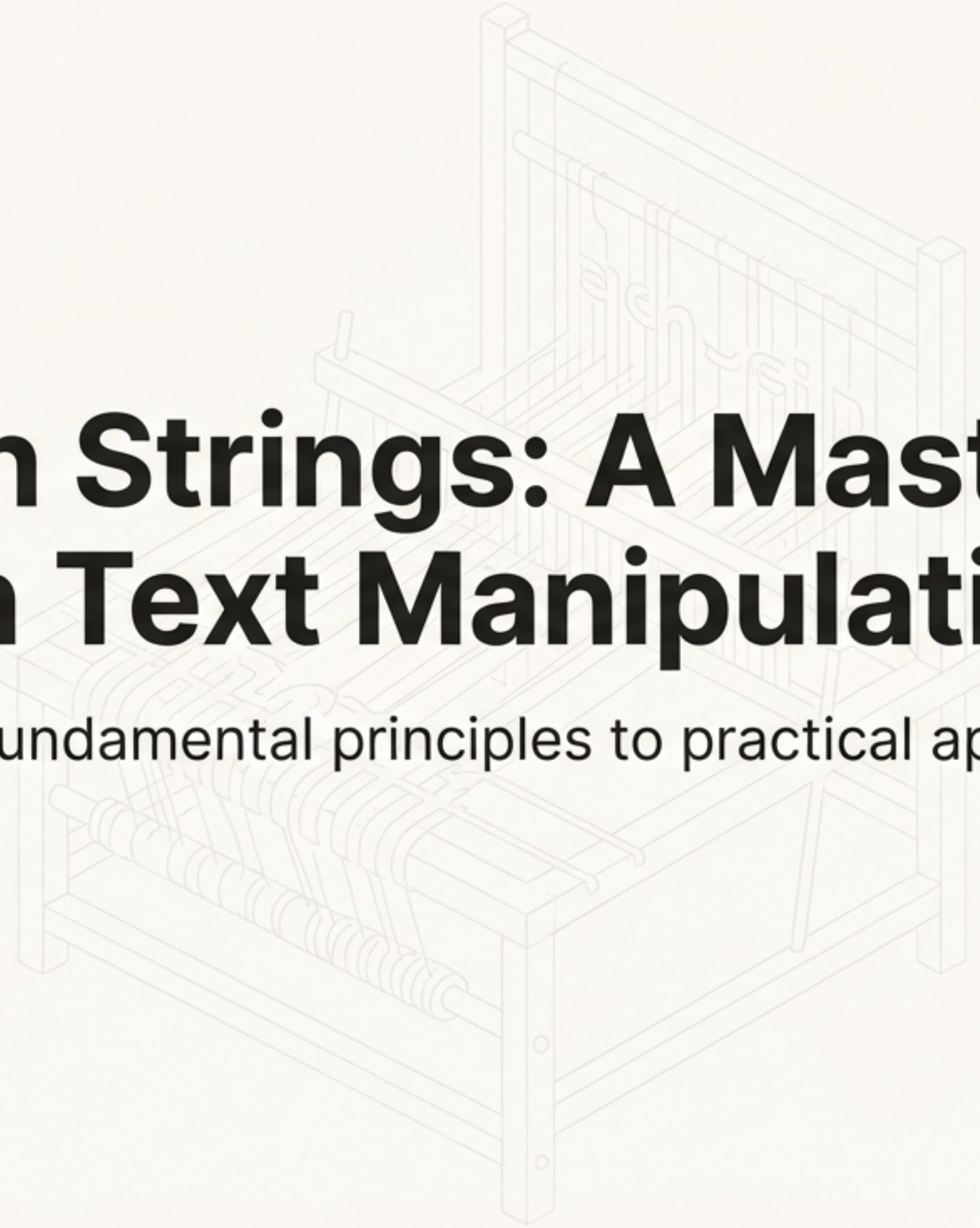


# Python Strings: A Masterclass in Text Manipulation

From fundamental principles to practical application.

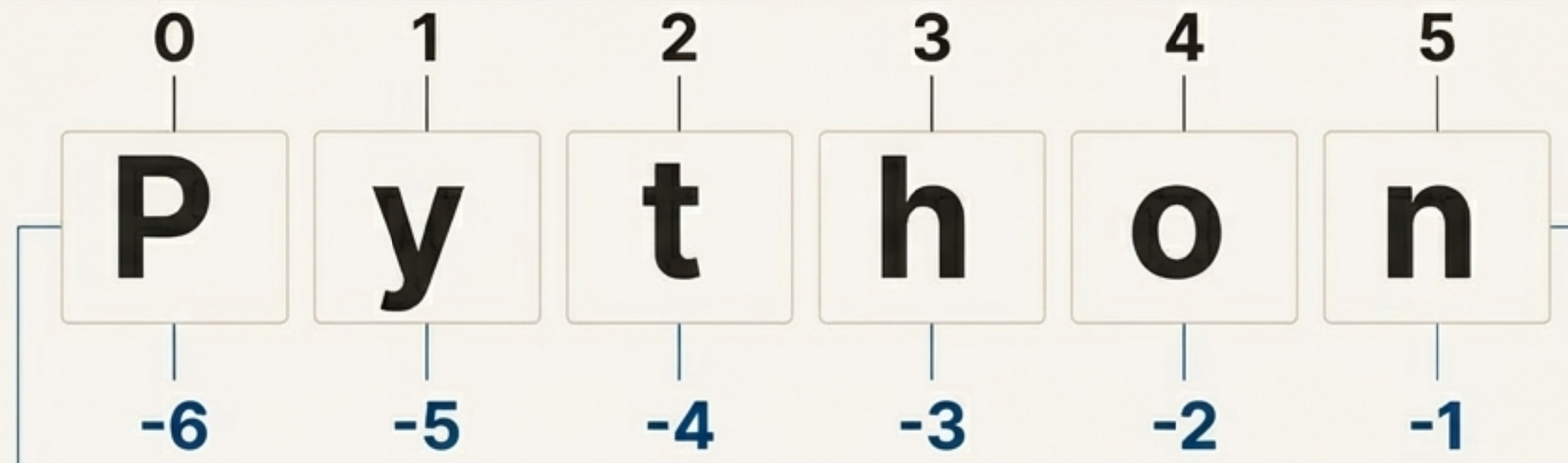
# At its Core, a String is an Immutable Sequence of Characters

In Python, a string is an ordered collection of characters, used to represent all text-based data. Once a string is created, it cannot be changed. This core principle, known as **immutability**, is crucial. Any operation that appears to “modify” a string actually creates a new one in memory.

```
# Using double quotes
s = "Python"

# Using single quotes
s2 = 'Hello'

# Using triple quotes for
multi-line strings
s3 = """This is a
multi-line string."""
```




# Accessing Characters by Position with Indexing

Every character in a string has a unique position, or index. Python uses zero-based indexing, meaning the first character is at index 0. You can also use negative indices to access characters from the end of the string.

```
txt = "Python"

# Accessing the first character
txt[0]  # Output: 'P'

# Accessing the 'h'
txt[3]  # Output: 'h'

# Accessing the last character
txt[-1] # Output: 'n'

# Accessing the second-to-last character
 JetBrains Mono Regular
txt[-2] # Output: 'o'
```

# Extracting Substrings with Precision Slicing

Slicing allows you to extract a portion of a string (a "substring") using the syntax `string[start:end:step]`. The start index is inclusive, and the end index is exclusive.

```
txt = "Python Programming"

# Extract 'Python' (from index 0 up to 6)
txt[0:6]  # Output: 'Python'
```

Python Programming

```
# Extract 'Programming' (from index 7 to the end)
txt[7:]  # Output: 'Programming'
```

Python Programming

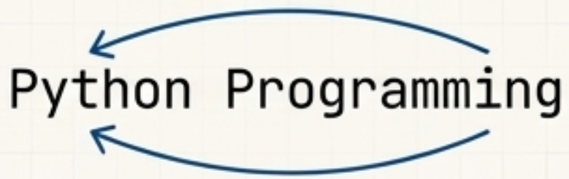
```
# Extract 'Python' (from the beginning up to 6)
txt[:6]  # Output: 'Python'
```

Python Programming

```
# Get every second character
txt[::2]  # Output: 'Pto rgamn'
```

Python Programming

```
# A classic trick: reverse the string
txt[::-1] # Output: 'gnimmargorP nohtyP'
```



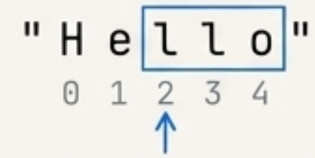
# The Immutable Rule in Practice: You Can't Change a String, You Replace It

Because strings are immutable, you cannot change a character by assigning to an index. Attempting this will result in a `TypeError`. The correct approach is to build a *new* string using slices and concatenation.

## The Error

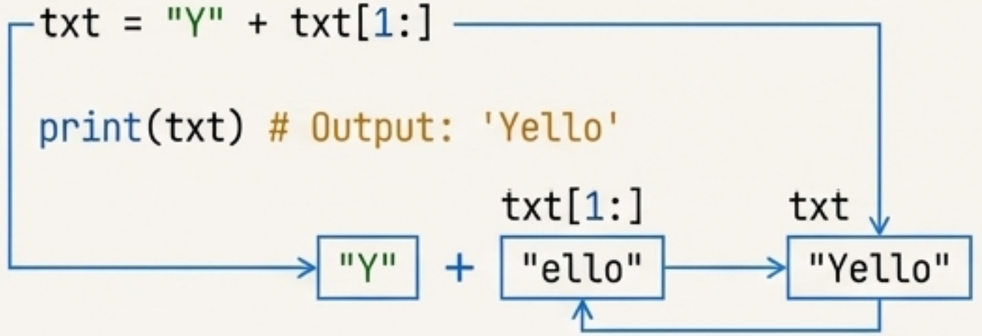
```
txt = "Hello"
txt[0] = "Y"  # This will raise a TypeError!
# TypeError: 'str' object does not support item assignment
```

## The Solution



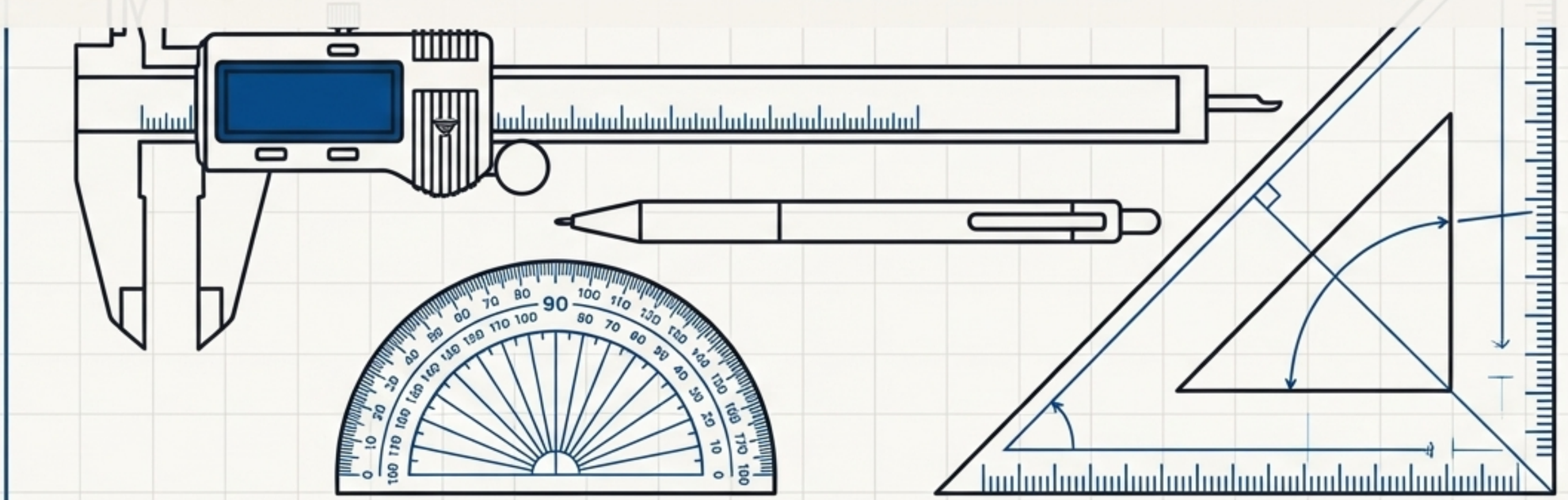

```
txt = "Hello"
# Create a new string by combining 'Y' with the res
txt = "Y" + txt[1:]

print(txt) # Output: 'Yello'
```



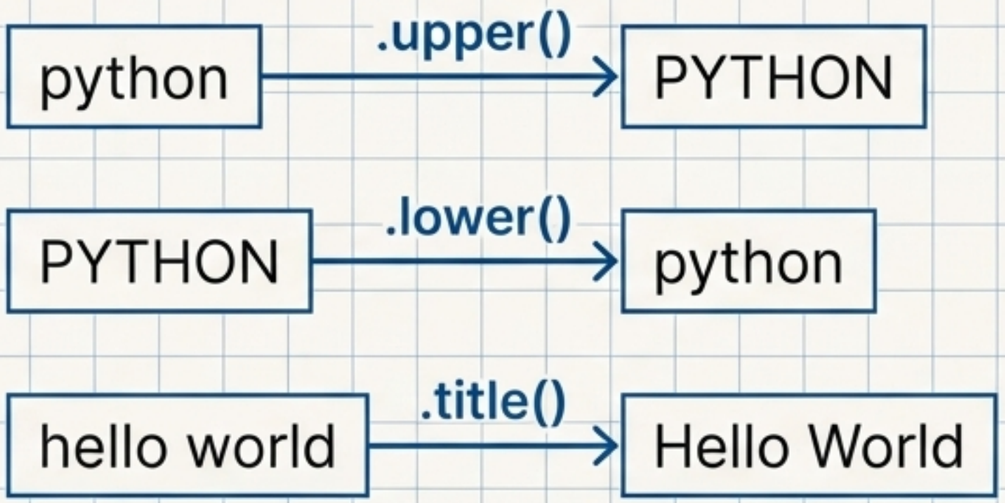
# The Toolkit: Manipulation & Transformation

A string's raw form is just the beginning. The real power lies in Python's rich, built-in toolkit of **methods** for transforming, parsing, and cleaning any text data you encounter. Let's explore the most essential tools.



# Standardising Text by Controlling Case

Easily change the case of your strings to standardise them for comparison or display.




```python
# Convert to all uppercase
"python".upper()
# Output: 'PYTHON'
```

```python
# Convert to all lowercase
"PYTHON".lower()
# Output: 'python'
```

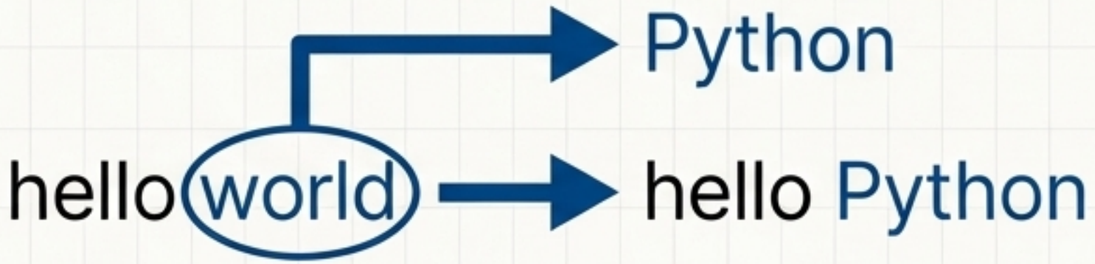
```python
# Capitalise the first letter of each word
"hello world".title()
# Output: 'Hello World'
```

# Cleaning and Reshaping Content with Strip and Replace

Prepare raw text for processing by removing unwanted whitespace or by replacing specific substrings with new content.

```
# .strip() removes leading/trailing whitespace
"  hello   ".strip()
# Output: 'hello'
```


```
# .replace() finds and replaces all occurrences of a substring
"hello world".replace("world", "Python")
# Output: 'hello Python'
```

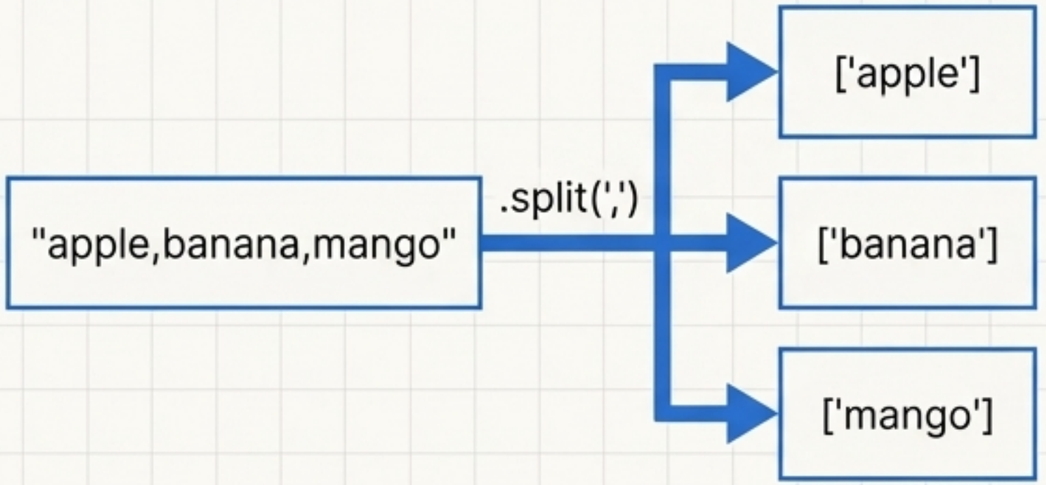


# Deconstructing and Rebuilding Strings with Split and Join

These two methods provide a powerful way to move between strings and lists of strings. Use .split() to break a string apart based on a delimiter, and .join() to assemble a string from a list of items.

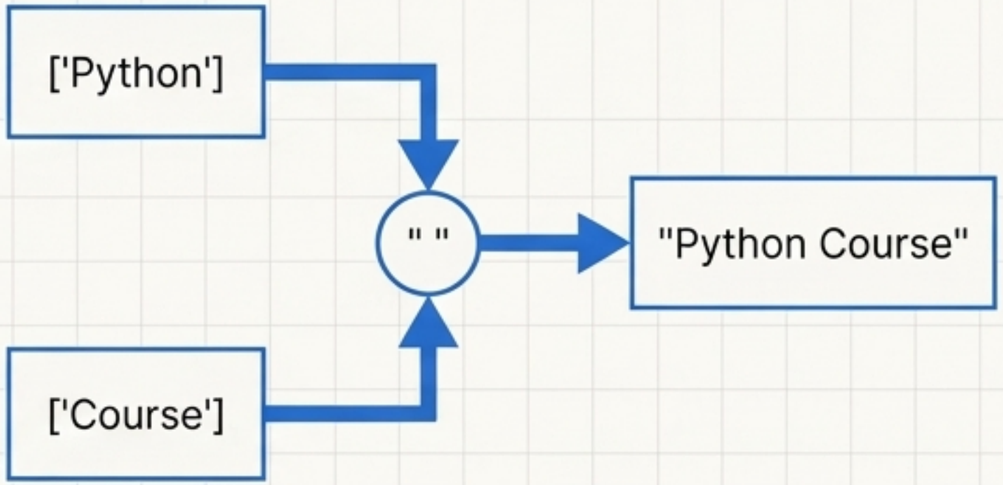
## .split() Section

```
# .split() deconstructs a string into a list of strings
"apple,banana,mango".split(",")
# Output: ['apple', 'banana', 'mango']
```




## .join() Section

```
# .join() builds a string from a list (or any iterable)
" ".join(["Python", "Course"])
# Output: 'Python Course'
```



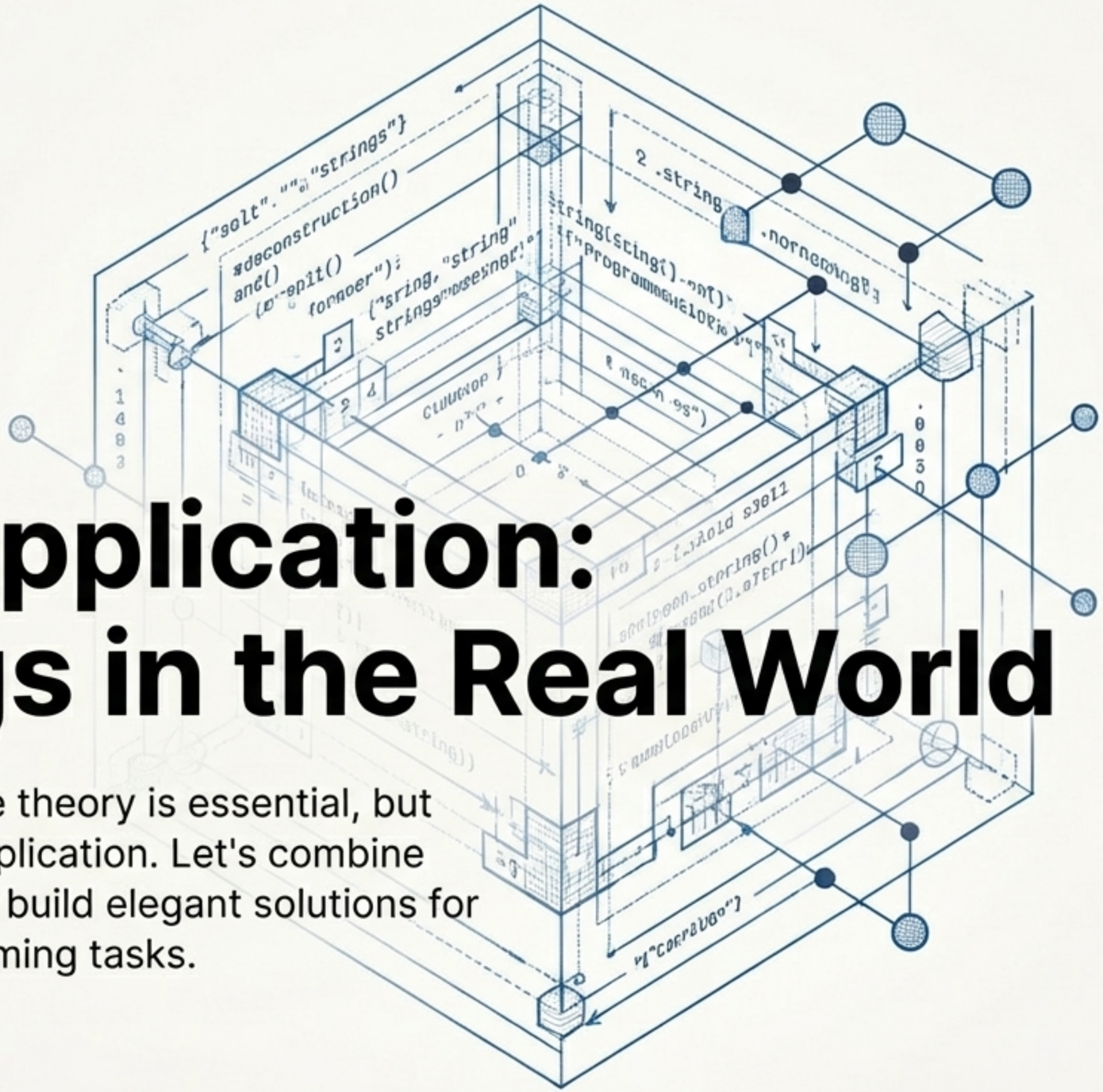
# Searching and Verifying String Content

Find the position of substrings or check if a string begins or ends with specific characters. These methods are fundamental for conditional logic.

```
# .find() returns the starting index of
the first occurrence (-1 if not found)
"Programming".find("gram")
# Output: 3

# .startswith() returns True or False
"Python".startswith("Py")
# Output: True

# .endswith() returns True or False
"Programming".endswith("ing")
# Output: True
```

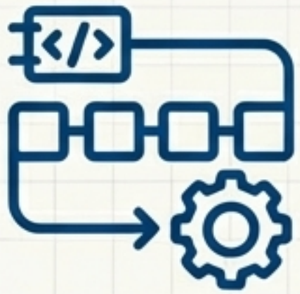


# The Application: Strings in the Real World

Understanding the theory is essential, but the true goal is application. Let's combine these concepts to build elegant solutions for common programming tasks.

# Building Dynamic Strings: Formatting, Escaping, and Raw Strings

Dynamically inserting variables into text is a core task. **F-strings** (formatted string literals) are the modern, recommended way to do this due to their clarity and performance.



```
# f-string (Recommended)                    Recommended
name = "Yaswant"
f"Hello {name}"  # Output: 'Hello Yaswant'

# Older methods: .format() and %-formatting
"Hello {}".format(name)
"Value is %d" % 10

# Escape characters allow for special formatting
print("First line\nSecond line\t-indented")
↳ First line
        Second line
# Raw strings ignore escape characters, useful
  for file paths
print(r"C:\newfolder\user")
```

Standard String (Escaped) → `C:ewfolderuser`

Raw String (Literal) → `C:\newfolder\user`

# Practical Recipes for Common Scenarios

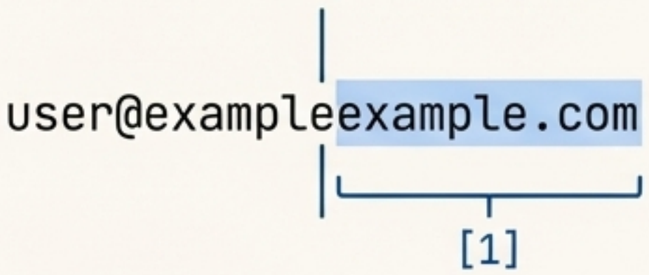
By chaining methods together, you can write concise and powerful code to handle everyday text processing tasks.

## Scenario 1: Clean and standardise user input.

```
# Raw input might be: "  AdminUser  "
user_input = input().strip().lower()
# Result: "adminuser"
```


## Scenario 2: Extract a domain name from an email address.

```
email = "user@example.com"
domain = email.split('@')[1]
# Result: "example.com"
```

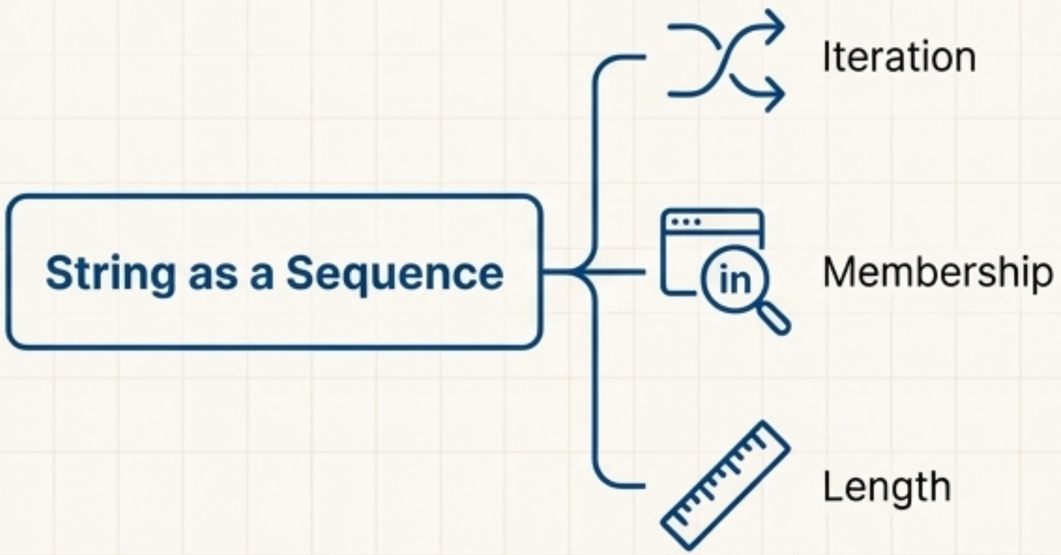



## Scenario 3: Check if a URL uses a secure protocol.

```
url = "https://www.website.com"
is_secure = url.startswith("https")
# Result: True
```

# Mastering the Sequence: Iteration, Membership, and Length

Beyond methods, remember that strings are sequences. This means you can iterate over them directly, test for membership with the `in` keyword, and get their size with `len()`—powerful patterns that apply to other Python sequences like lists and tuples.




## Iteration

```
# Iteration
for char in "Python":
    print(char) # Prints each character on a new li
```

## Membership Testing

```
# Membership Testing
"a" in "Java"         # Output: True
"x" not in "Python"   # Output: True
```

## Length

```
# Length
len("Python")         # Output: 6
```

# The String is Your Foundational Tool for Data


**Blueprint:** Understand strings as ordered, **immutable** sequences.

**Toolkit:** Master methods like `.split()`, `.join()`, `.strip()`, and `.replace()` to manipulate any text.

**Application:** Combine these tools with f-strings and sequence operations to write clean, effective, and professional Python code.

**The ability to confidently and efficiently handle text is not just a feature of Python—it is a cornerstone of versatile and powerful programming.**