String Functions in Python: An Introduction

Welcome to this comprehensive guide on string functions in Python. As one of the most versatile and commonly used data types, strings form the backbone of text processing in Python programming. This presentation will walk you through the extensive capabilities Python offers for working with strings, from basic operations to advanced manipulations.

Strings in Python are sequences of Unicode characters, making them perfect for handling text in virtually any human language. With over 51 built-in string methods available, Python provides developers with powerful tools for text manipulation, parsing, and data cleaning without requiring external libraries.

Whether you're a beginner looking to understand the basics or an experienced developer seeking to refine your string manipulation skills, this guide will provide you with practical code examples and real-world applications to enhance your Python programming toolkit.

What is a String in Python?

In Python, a string is a sequence of Unicode characters enclosed within quotation marks. This fundamental data type is used to represent and manipulate text data within your programs. Python strings have several key characteristics that set them apart from other data types:

Immutability

Once created, a string cannot be changed. Any operation that appears to modify a string actually creates a new string object. This immutability helps with memory management and string interning.

Quotation Flexibility

Strings can be defined using either single quotes (') or double quotes ("). This flexibility allows you to include quote characters within your strings without excessive escaping.

Unicode Support

Python strings fully support Unicode, allowing you to work with characters from virtually any human language or symbol set.

```
# Creating strings in Python
word = "Hello World“
name = 'Alice'# Both are equivalent
print(word) # Output: Hello World
print(name) # Output: Alice
# Accessing a string's type
print(type(word)) # Output: # Demonstrating immutability
try:
          word[0] = 'J’
e: print(f"Error: {e}")
# This will raise an errorexcept TypeError as
# Output: Error: 'str' object does not support item assignment
```

Creating Strings

Python offers multiple ways to create and initialize strings, each with its own use cases and advantages. The flexibility in string creation makes Python suitable for various text processing scenarios, from simple variable assignments to complex multi-line text handling.

1 Basic String Assignment

The most common way to create a string is by enclosing text within single or double quotes. Both methods are equivalent, and your choice often depends on whether you need to include quote characters within the string itself.

```
name = "Alice“
greeting = 'Hello there!'
```

2 Multi-line Strings

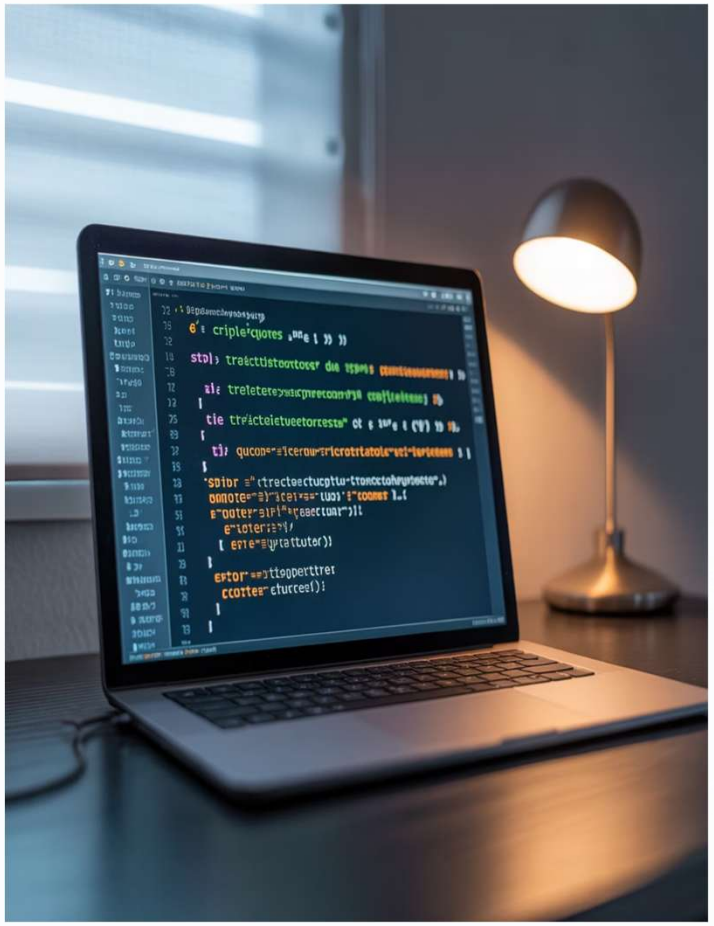
When you need to create strings that span multiple lines, triple quotes (either single or double) allow you to include line breaks without using escape characters.

```
paragraph = """This is a multi-linestring that preserves all line
breaksand formatting as written."""
```

3 Empty and Special Strings

You can create empty strings or convert other data types to strings using the str() constructor.

```
number_as_string = str(42)  # Converts integer to string
```



String Creation from Other Data Types

```
# Convert other data types to string
sage = 30
age_str = str(age)
print("Type of age:", type(age)) #
print("Type of age_str:", type(age_str))
# # String interpolation with f-strings (Python 3.6+)
name = "Bob“
age = 25
profile = f"Name: {name}, Age: {age}“
print(profile) # Output: Name: Bob, Age: 25
# Raw strings - useful for file paths
file_path = r"C:\Users\Documents\notes.txt“
print(file_path) # Output: C:\Users\Documents\notes.txt
```

Accessing Characters in Strings

Strings in Python are sequences of characters, and like other sequence types, individual characters can be accessed using indexing. Python uses zero-based indexing, meaning the first character is at position 0, the second at position 1, and so on.

Positive Indexing

Access characters from the beginning of the string using positive indices starting from 0.

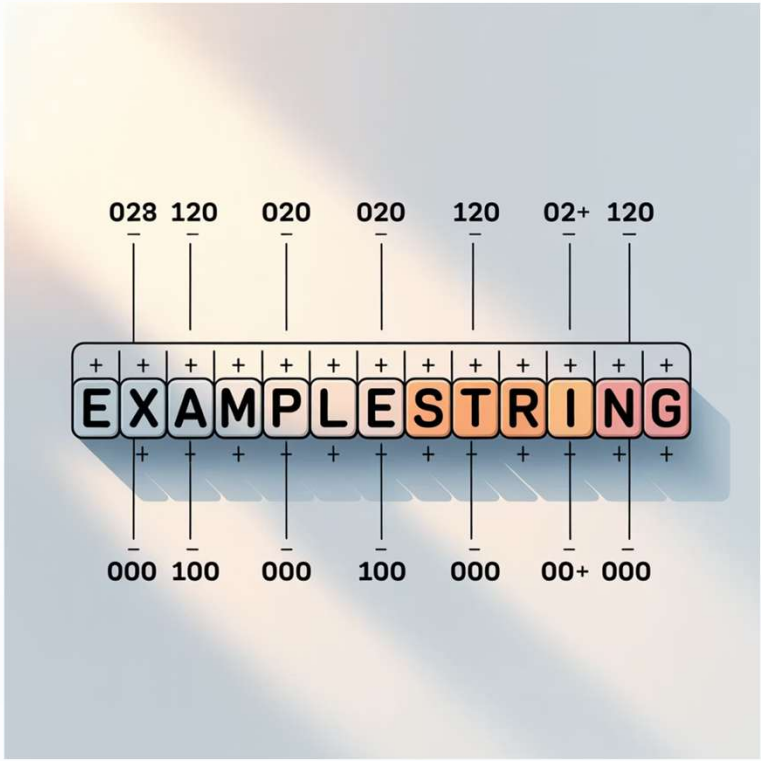
```
word = "Python“
first = word[0]    # 'P’
second = word[1]   # 'y’
third = word[2]    # 't'
```

Negative Indexing

Access characters from the end of the string using negative indices, where -1 refers to the last character.

```
word = "Python“
last = word[-1]     # 'n’
second_last = word[-2]  # 'o’
third_last = word[-3]   # 'h'
```

If you try to access an index that is out of range, Python will raise an IndexError. It's always a good practice to check if an index is valid before accessing it, especially when working with user input or data of unknown length.



Complete Example: String Indexing

```
# Demonstrating string indexing
word = "Hello World“
print(f"Original string: {word}")
# Positive indexing
print(f"First character: {word[0]}") # H
print(f"Second character: {word[1]}") # e
print(f"Sixth character: {word[5]}") # space
# Negative indexing
print(f"Last character: {word[-1]}") # d
print(f"Second-to-last: {word[-2]}") # l
print(f"Third-to-last: {word[-3]}") # r
```

```
# Handling index errors
try:
          print(word[20]) # This will raise an IndexError
except IndexError as e:
          print(f"Error: {e}") # Output: Error: string index out of range
```

Slicing Strings

String slicing is a powerful feature in Python that allows you to extract a substring (a portion of a string) by specifying a range of indices. Slicing uses the syntax `string[start:end:step]`, where:

- **start**: The index where the slice begins (inclusive). Defaults to 0 if omitted.
- **end**: The index where the slice ends (exclusive). Defaults to the string length if omitted.
- **step**: The stride between characters. Defaults to 1 if omitted.

Unlike indexing, slicing doesn't raise an error if the indices are out of range - it simply adjusts them to the nearest valid index or returns an empty string.

1

Basic Slicing

```
s = "Python Programming“
print(s[2:5])  # 'tho'
```

Gets characters from position 2 (inclusive) to position 5 (exclusive).

2

Omitting Indices

```
print(s[:3])   # 'Pyt’
print(s[3:])   # 'hon Programming'print(s[:])    # 'Python Programming'
```

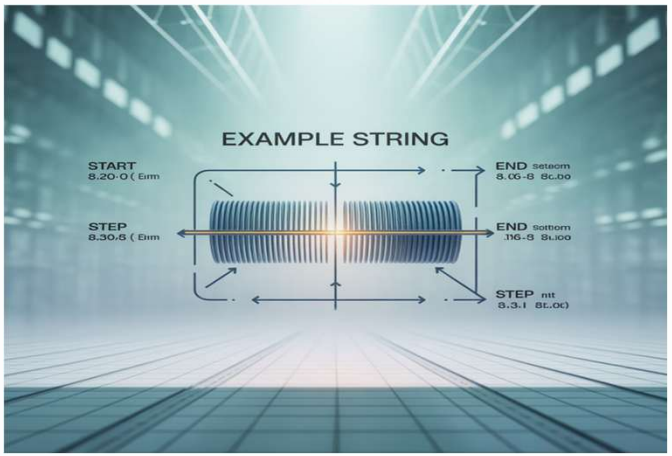
If you omit the start index, slicing begins from 0. If you omit the end index, slicing continues to the end of the string.

3

Using Step

```
print(s[::2])  # 'Pto rgamn'print(s[::-1]) # 'gnimmargorP nohtyP'
```

A step of 2 takes every other character, while a step of -1 reverses the string.



4

Advanced Slicing Techniques

```
text = "Python is amazing!"

# Extract specific words
first_word = text[:6] # 'Python'
last_word = text[10:] # 'amazing!'

# Negative indices in slicing
last_five = text[-6:-1] # 'zing!'

# Reverse a string
reversed_text = text[::-1] # '!gnizama si nohtyP'

# Extract every nth character
every_second = text[::2] # 'Pto saai!'
every_third = text[::3] # 'Ph mn!'

# Slicing with out-of-range indices (no error)
print(text[100:105]) # '' (empty string)
print(text[-100:5]) # 'Pytho'

# Extract the last word with rsplit()
last_word_alt = text.rsplit(' ', 1)[-1] # 'amazing!'
```

String Concatenation and Repetition

String concatenation and repetition are fundamental operations that allow you to combine strings or repeat them multiple times. Python provides intuitive operators for these operations, making it easy to build complex strings from simpler components.

String Concatenation with + Operator

The + operator combines two or more strings into a single string. This is the most common way to join strings in Python.

```
first_name = "John“
last_name = "Doe“
full_name = first_name + " " + last_name # Output: John Doe
```

String Repetition with * Operator

The * operator repeats a string a specified number of times, creating a new string with the repeated content.

```
pattern = "=" * 10# Output: ==========
cheer = "Hip " * 2 + "Hooray!"# Output: Hip Hip Hooray!
```

Concatenation with join() Method

For joining multiple strings, especially from a list, the join() method is more efficient than repeated + operations.

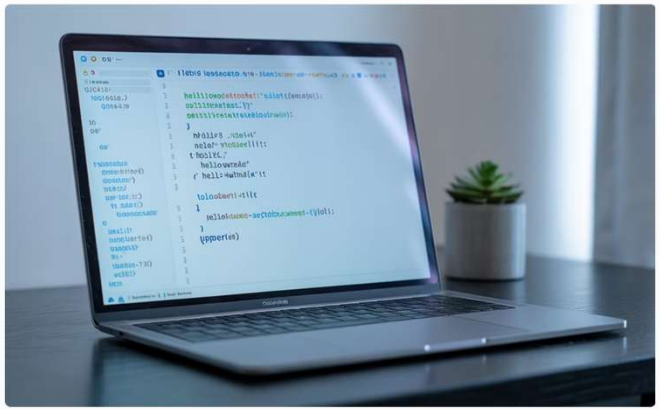
```
words = ["Python", "is", "powerful"]
sentence = " ".join(words)# Output: Python is powerful
```

Performance Considerations

When concatenating many strings, especially in loops, avoid using the + operator repeatedly as it creates new string objects each time. Instead, use the join() method or string formatting for better performance.

```
# Inefficient way (creates many temporary strings)
result = ""
for i in range(1, 6):
 result = result + str(i) + " "
print(result) # "1 2 3 4 5 "

# Efficient way using join
result = " ".join(str(i) for i in range(1, 6))
print(result) # "1 2 3 4 5"

# String repetition in patterns
line = "-" * 20
box = "+" + "-" * 10 + "+\n" + \
 "|" + " " * 10 + "|\n" + \
 "+" + "-" * 10 + "+"
print(box)
# Output:
# +----------+
# | |
# +----------+
```

Changing Case: upper(), lower(), title()



upper()

The upper() method converts all characters in a string to uppercase. This is particularly useful for case-insensitive comparisons or when you need to display text in all capitals for emphasis.

```
text = "Hello, World!“
uppercase = text.upper()
print(uppercase)  # "HELLO, WORLD!“
# Case-insensitive comparison
user_input = "yes“
if user_input.upper() == "YES":
print("User agreed")
```

lower()

The lower() method converts all characters in a string to lowercase. This is commonly used for standardizing text input or ensuring case-insensitive matching.

```
text = "Hello, World!“
lowercase = text.lower()print(lowercase)
# "hello, world!“
# Normalize email addresses
email = User@Example.COM
normalized = email.lower()
print(normalized)  # "user@example.com"
```

title()

The title() method capitalizes the first letter of each word in a string. This is ideal for formatting names, titles, and headings according to common capitalization conventions.

```
text = "python string methods“
titled = text.title()
print(titled)  # "Python String Methods“
# Format a person's name
name = "john smith“
formatted = name.title()
print(formatted)  # "John Smith"
```

Additional Case Methods

capitalize(): Capitalizes only the first character of the string, leaving the rest in lowercase.

```
text = "hello world“
print(text.capitalize())  # "Hello world"
```

swapcase(): Swaps the case of each character (uppercase becomes lowercase and vice versa).

```
text = "Hello World“
print(text.swapcase())  # "hELLO wORLD"
```

It's important to note that all these methods create and return new strings rather than modifying the original string, since strings in Python are immutable. Also, these methods handle non-alphabetic characters appropriately - they remain unchanged.

Trimming Whitespace: strip(), lstrip(), rstrip()

Whitespace characters (spaces, tabs, newlines) often appear at the beginning or end of strings, especially when dealing with user input or data from external sources. Python provides three methods to remove unwanted whitespace from strings:

strip()

Removes whitespace from both the beginning and end of a string. This is the most commonly used trimming method when you want to clean up a string completely.

```
text = "   Hello, World!   “
cleaned = text.strip()
print(cleaned)  # "Hello, World!"
```

lstrip()

Removes whitespace only from the beginning (left side) of a string. This is useful when you want to preserve trailing whitespace for formatting purposes.

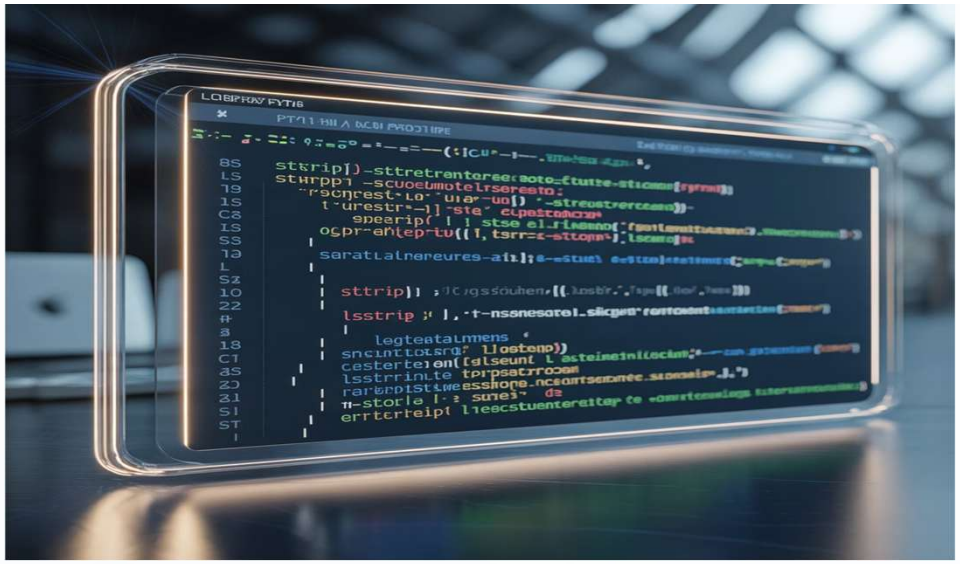
```
text = "   Hello, World!   “
left_cleaned = text.lstrip()
print(left_cleaned)  # "Hello, World!   "
```

rstrip()

Removes whitespace only from the end (right side) of a string. This is useful when you want to preserve leading whitespace for indentation.

```
text = "   Hello, World!   “
right_cleaned = text.rstrip()
print(right_cleaned)  # "   Hello, World!"
```

All these methods can also take an optional argument specifying which characters to remove instead of whitespace. This makes them even more versatile for cleaning up strings.



Advanced Trimming Examples

```
# Remove specific characters
url = "https://example.com/"
clean_url = url.rstrip("/")
print(clean_url) # "https://example.com"
# Clean CSV data
data = " 42, John, New York "
clean_data = ",".join([field.strip() for field in data.split(",")])
print(clean_data) # "42,John,New York"
# Remove multiple types of characters
text = "###Hello, World!***"
clean_text = text.strip("#*")
print(clean_text) # "Hello, World!"
# Handle multiline text
multiline = """
 This is a multi-line text
 with leading whitespace.
 """
clean_multiline = "\n".join([line.lstrip() for line in
 multiline.strip().split("\n")])
print(clean_multiline)
# Output:
# This is a multi-line text
# with leading whitespace.
```

Finding Substrings: find(), rfind(), index()

Python provides several methods to search for substrings within a string. These methods are essential for text parsing, data extraction, and string manipulation tasks.

1

find()

The `find()` method searches for the first occurrence of a substring and returns its starting index. If the substring is not found, it returns -1.

```
text = "Python is amazing. Python is powerful."
position = text.find("Python")
print(position)  # 0 (found at the beginning)

position = text.find("amazing")
print(position)  # 10

position = text.find("Java")
print(position)  # -1 (not found)
```

You can also specify start and end positions for the search:

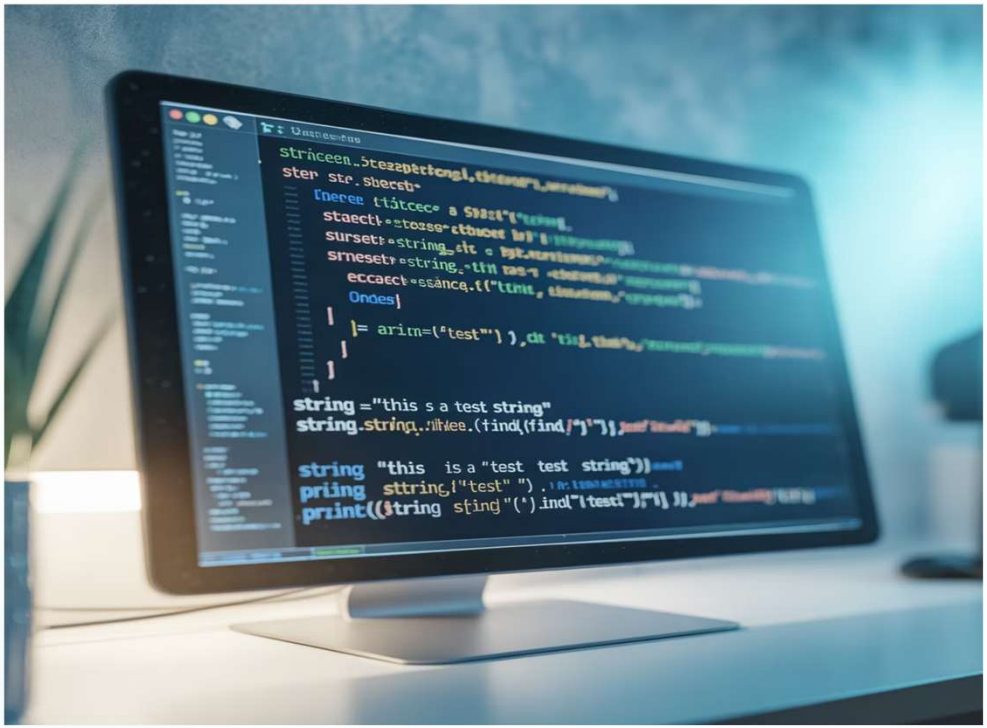
```
# Find the second occurrence of "Python"
first_pos = text.find("Python")
second_pos = text.find("Python", first_pos + 1)
print(second_pos)  # 19
```

2

rfind()

The `rfind()` method works like `find()` but searches backward from the end of the string, returning the position of the last occurrence.

```
text = "Python is amazing. Python is powerful."
position = text.rfind("Python")
print(position)  # 19 (last occurrence)

position = text.rfind("is")
print(position)  # 22 (last "is")
```



index()

3

The `index()` method works like `find()` but raises a `ValueError` if the substring is not found, rather than returning -1.

```
# Extract domain from email
email = "user@example.com"
at_position = email.find("@")
if at_position != -1:
    domain = email[at_position + 1:]
    print(f"Domain: {domain}")  # Domain: example.com
```

There's also an `rindex()` method that works like `rfind()` but raises an exception if the substring is not found.

Counting Occurrences: count()

The `count()` method in Python strings provides a straightforward way to count how many times a substring appears within a string. This method is invaluable for text analysis, data validation, and content processing tasks.

Basic Syntax

```
string.count(substring, start, end)
```

- **substring**: The string whose occurrences you want to count (required)
- **start**: The position to start searching from (optional, defaults to 0)
- **end**: The position to end the search (optional, defaults to end of string)

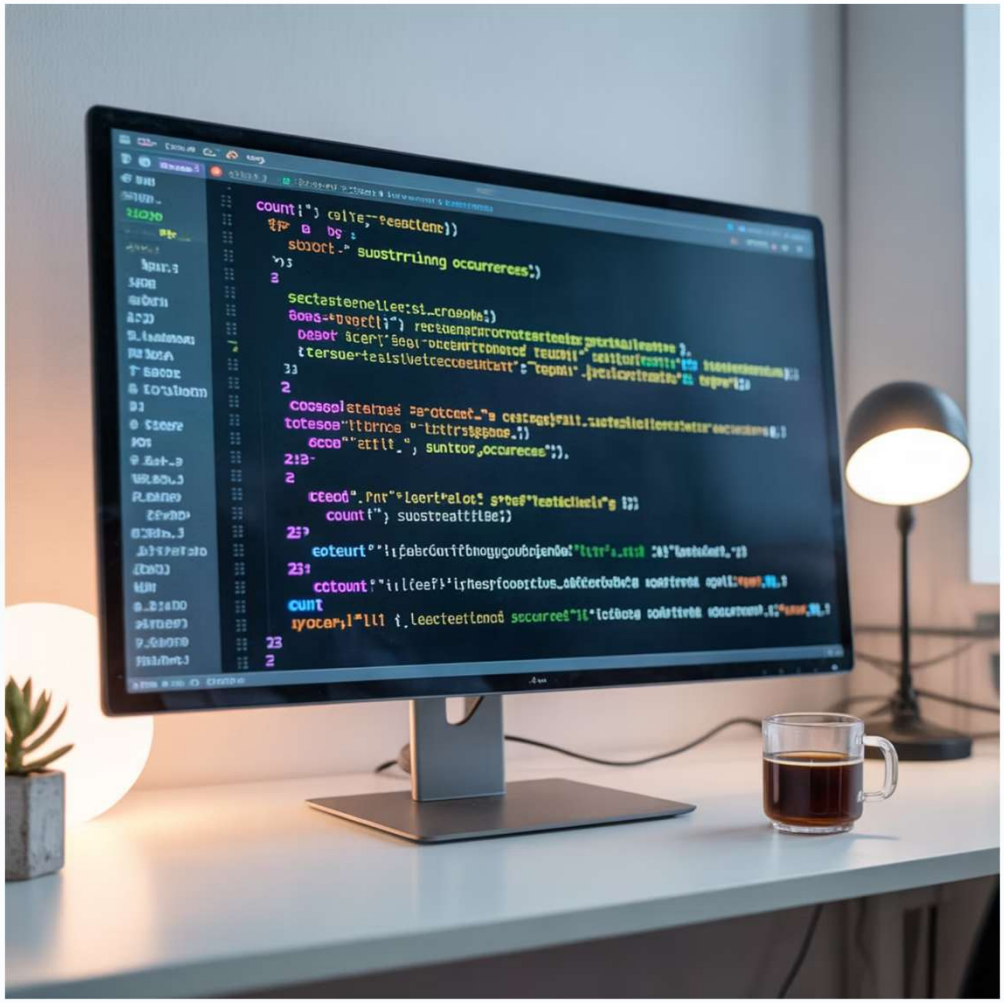
The method returns an integer representing the number of non-overlapping occurrences of the substring.

Basic Usage

```
text = "banana"
occurrences = text.count("a")
print(occurrences)  # 3

sentence = "Python is powerful and Python is easy to learn."
python_count = sentence.count("Python")
print(python_count)  # 2
```

With Optional Parameters

```
text = "one two one two one three"
# Count in a substring
count = text.count("one", 5, 15)
print(count)  # 1 (only counts "one" between positions 5 and 15)
```



Replacing Substrings: replace()

The `replace()` method in Python is a powerful tool for substituting parts of a string with another string. It's commonly used for text cleanup, data normalization, censoring, and many other string manipulation tasks.

Basic Syntax

```
string.replace(old, new, count)
```

- **old**: The substring you want to replace (required)
- **new**: The string to replace it with (required)
- **count**: Maximum number of replacements to make (optional)

If count is not specified, all occurrences are replaced. The method returns a new string with the replacements made.

Simple Replacement

```
text = "Python is amazing. Python is powerful."
new_text = text.replace("Python", "JavaScript")
print(new_text)
# Output: JavaScript is amazing. JavaScript is powerful.
```

Limited Replacement

```
text = "Python is amazing. Python is powerful."
new_text = text.replace("Python", "JavaScript", 1)
print(new_text)
# Output: JavaScript is amazing. Python is powerful.
```

Here, only the first occurrence is replaced because we specified count=1.

Remember that strings in Python are immutable, so `replace()` doesn't modify the original string - it creates and returns a new string with the replacements made.

Advanced Replacement Techniques

```
# Multiple replacements
text = "Hello, world! Hello, Python!"
replacements = {"Hello": "Hi", "world": "there", "Python": "everyone"}

for old, new in replacements.items():
 text = text.replace(old, new)

print(text) # Output: Hi, there! Hi, everyone!

# Censoring text
def censor_text(text, bad_words):
 for word in bad_words:
 # Replace each character with '*'
 censored = '*' * len(word)
 text = text.replace(word, censored)
 return text

message = "This damn code won't work!"
clean_message = censor_text(message, ["damn"])
print(clean_message) # Output: This **** code won't work!

# Clean formatting
def normalize_whitespace(text):
 # Replace multiple spaces with a single space
 return ' '.join(text.split())

messy_text = "This has too many spaces."
clean_text = normalize_whitespace(messy_text)
print(clean_text) # Output: This has too many spaces.

# Data cleaning - remove special characters
data = "Price: $1,234.56"
numbers_only = data.replace("$", "").replace(",", "")
print(float(numbers_only)) # Output: 1234.56
```

Splitting Strings: split(), rsplit()

Splitting strings is one of the most common text processing operations, allowing you to break a string into a list of substrings based on a delimiter. Python provides powerful methods for this purpose: `split()` and `rsplit()`.

1

split() Method

The `split()` method divides a string into a list of substrings based on a separator.

```
# Basic syntax: string.split(separator, maxsplit)

# Split by spaces (default)
text = "Python is a programming language"
words = text.split()
print(words)
# ['Python', 'is', 'a', 'programming', 'language']

# Split by specific character
csv_data = "apple,orange,banana,grape"
fruits = csv_data.split(',')
print(fruits)
# ['apple', 'orange', 'banana', 'grape']

# Limit the number of splits
sentence = "Python is powerful and easy to learn"
parts = sentence.split(' ', 3)
print(parts)
# ['Python', 'is', 'powerful', 'and easy to learn']
```

2

rsplit() Method

The `rsplit()` method works like `split()` but starts splitting from the right end of the string.

```
# Basic syntax: string.rsplit(separator, maxsplit)
path = "/home/user/documents/file.txt"
parts = path.rsplit('/', 1)
print(parts)
# ['/home/user/documents', 'file.txt']
# This is useful for getting file extensions
filename = "document.report.pdf"
name, extension = filename.rsplit('.', 1)
print(f"Name: {name}, Extension: {extension}")
# Name: document.report, Extension: pdf
```

3

splitlines() Method

For multiline text, the `splitlines()` method breaks a string at line boundaries.

```
text = """Line 1
Line 2
Line 3"""
lines = text.splitlines()
print(lines)
# ['Line 1', 'Line 2', 'Line 3']

# Keep the line breaks
lines_with_breaks = text.splitlines(True)
print(lines_with_breaks)
# ['Line 1\n', 'Line 2\n', 'Line 3']
```

Practical Applications

```
# Parsing CSV data
csv_line = "John,Doe,35,New York,Engineer"
name, surname, age, city, profession = csv_line.split(',')
print(f"Name: {name} {surname}, Age: {age}")
# Output: Name: John Doe, Age: 35
# Processing log files
log_entry = "2023-09-15 14:30:45 [INFO] User login successful"
timestamp, level, message = log_entry.split(' ', 2)
date, time = timestamp.split(' ')
print(f"Date: {date}, Level: {level.strip('[]')}")
# Output: Date: 2023-09-15, Level: INFO
# Extracting domain from email
email = "user@example.com"
username, domain = email.split('@')
print(f"Username: {username}, Domain: {domain}")
# Output: Username: user, Domain: example.com
# Parsing command-line arguments
command = "--file=report.txt --verbose --output=summary.csv"
args = {}
for arg in command.split():
 if '=' in arg:
 key, value = arg.split('=', 1)
 args[key.lstrip('-')] = value
 else:
 args[arg.lstrip('-')] = True

print(args)
# Output: {'file': 'report.txt', 'verbose': True, 'output': 'summary.csv'}
```

Joining Strings: join()

The `join()` method is one of Python's most powerful string methods, allowing you to combine a list of strings into a single string with a specified delimiter. Unlike most other string methods, `join()` is called on the delimiter string, not on the strings being joined.

Basic Syntax

```
delimiter.join(iterable)
```

Where `delimiter` is the string to insert between elements of the iterable, and `iterable` is a sequence of strings to be joined (like a list, tuple, or any iterable that produces strings).

Simple Examples

```
# Join a list of strings with a space
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence)  # "Python is awesome"

# Join with a comma and space
fruits = ["apple", "orange", "banana"]
fruit_list = ", ".join(fruits)
print(fruit_list)  # "apple, orange, banana"

# Join with no delimiter
chars = ["H", "e", "l", "l", "o"]
word = "".join(chars)
print(word)  # "Hello"
```

Common Use Cases

```
# Create CSV row
data = ["John", "Doe", "30", "New York"]
csv_row = ",".join(data)
print(csv_row)  # "John,Doe,30,New York"

# Build a path
path_parts = ["home", "user", "documents", "file.txt"]
file_path = "/".join(path_parts)
print(file_path)  # "home/user/documents/file.txt"

# Create HTML list
items = ["Item 1", "Item 2", "Item 3"]
html_list = ",".join(items)
print(html_list)
```

Item 1,Item 2,Item 3

Advanced Join Techniques

```
# Join with list comprehension
numbers = [1, 2, 3, 4, 5]
number_str = ", ".join(str(num) for num in numbers)
print(number_str) # "1, 2, 3, 4, 5"

# Create multiline string
lines = ["Line 1", "Line 2", "Line 3"]
multiline = "\n".join(lines)
print(multiline)
# Output:
# Line 1
# Line 2
# Line 3

# Join dictionary keys and values
user = {"name": "John", "age": "30", "city": "New York"}
user_info = ", ".join(f"{k}={v}" for k, v in user.items())
print(user_info) # "name=John, age=30, city=New York"
```

Partitioning Strings: partition(), rpartition()

The `partition()` and `rpartition()` methods provide a unique way to split a string into three parts: the part before the separator, the separator itself, and the part after the separator. These methods are particularly useful for parsing structured data and extracting specific segments from strings.

partition() Method

The `partition()` method splits a string at the first occurrence of the specified separator and returns a 3-tuple containing:

1. The part before the separator
2. The separator itself
3. The part after the separator

If the separator is not found, it returns a 3-tuple containing the original string and two empty

```
text = "Python is amazing"
before, sep, after = text.partition(" is ")
print(before)  # "Python"
print(sep)     # " is "
print(after)   # "amazing"

# Separator not found
text = "Hello World"
result = text.partition(":")
print(result)  # ('Hello World', '', '')
```

rpartition() Method

The `rpartition()` method works exactly like `partition()`, but it searches for the last occurrence of the separator instead of the first.

```
path = "home/user/documents/report.pdf"
before, sep, after = path.rpartition("/")
print(before)  # "home/user/documents"
print(sep)     # "/"
print(after)   # "report.pdf"

email = "user@example.com"
username, sep, domain = email.partition("@")
print(f"Username: {username}")  # Username: user
print(f"Domain: {domain}")      # Domain: example.com
```

Unlike `split()`, which can create a list of variable length, `partition()` and `rpartition()` always return a 3-tuple. This consistency makes them ideal for unpacking into variables when you know exactly what structure you're expecting.

Practical Applications

```
# Parse key-value pairs
entry = "name=John Doe"
key, sep, value = entry.partition("=")
print(f"Key: {key}, Value: {value}")
# Output: Key: name, Value: John Doe

# Extract file name and extension
filename = "document.report.pdf"
name_part, sep, ext = filename.rpartition(".")
print(f"Name: {name_part}, Extension: {ext}")
# Output: Name: document.report, Extension: pdf

# Parse HTTP headers
header = "Content-Type: application/json"
name, _, value = header.partition(": ")
print(f"Header: {name}, Value: {value}")
# Output: Header: Content-Type, Value: application/json

# Extract hostname from URL
url = "https://www.example.com/path/to/page.html"
# First partition by ://
_, _, address = url.partition("://")
# Then partition by the first /
hostname, _, path = address.partition("/")
print(f"Hostname: {hostname}") # Hostname: www.example.com
print(f"Path: /{path}") # Path: /path/to/page.html

# Parsing log entries
log = "2023-09-15 14:30:45 [INFO] User login successful"
timestamp, _, rest = log.partition(" [")
level, _, message = rest.partition("] ")
print(f"Timestamp: {timestamp}") # Timestamp: 2023-09-15 14:30:45
print(f"Level: {level}") # Level: INFO
print(f"Message: {message}") # Message: User login successful
```

Checking Start/End: startswith(), endswith()

The `startswith()` and `endswith()` methods provide simple yet powerful ways to check if a string begins or ends with specific substrings. These methods are essential for validating formats, filtering files, and performing conditional operations based on string prefixes or suffixes.

startswith() Method

The `startswith()` method checks if a string begins with the specified prefix and returns `True` or `False`.

```
# Basic syntax: string.startswith(prefix, start, end)

filename = "document.pdf"
if filename.startswith("doc"):
    print("It's a document file")

url = "https://example.com"
is_secure = url.startswith("https://")
print(is_secure)  # True

# Check with start position
text = "Python is amazing"
print(text.startswith("is", 7))  # True
```

You can also check for multiple prefixes by passing a tuple:

```
file = "image.jpg"
if file.startswith(("image", "photo", "img")):
    print("It's an image file")
```

endswith() Method

The `endswith()` method checks if a string ends with the specified suffix and returns `True` or `False`.

```
# Basic syntax: string.endswith(suffix, start, end)

filename = "report.pdf"
if filename.endswith(".pdf"):
    print("It's a PDF file")  # This will print

email = "user@example.com"
is_org = email.endswith(".org")
print(is_org)  # False

# Check with range
text = "Python programming"
print(text.endswith("Python", 0, 6))  # True
```

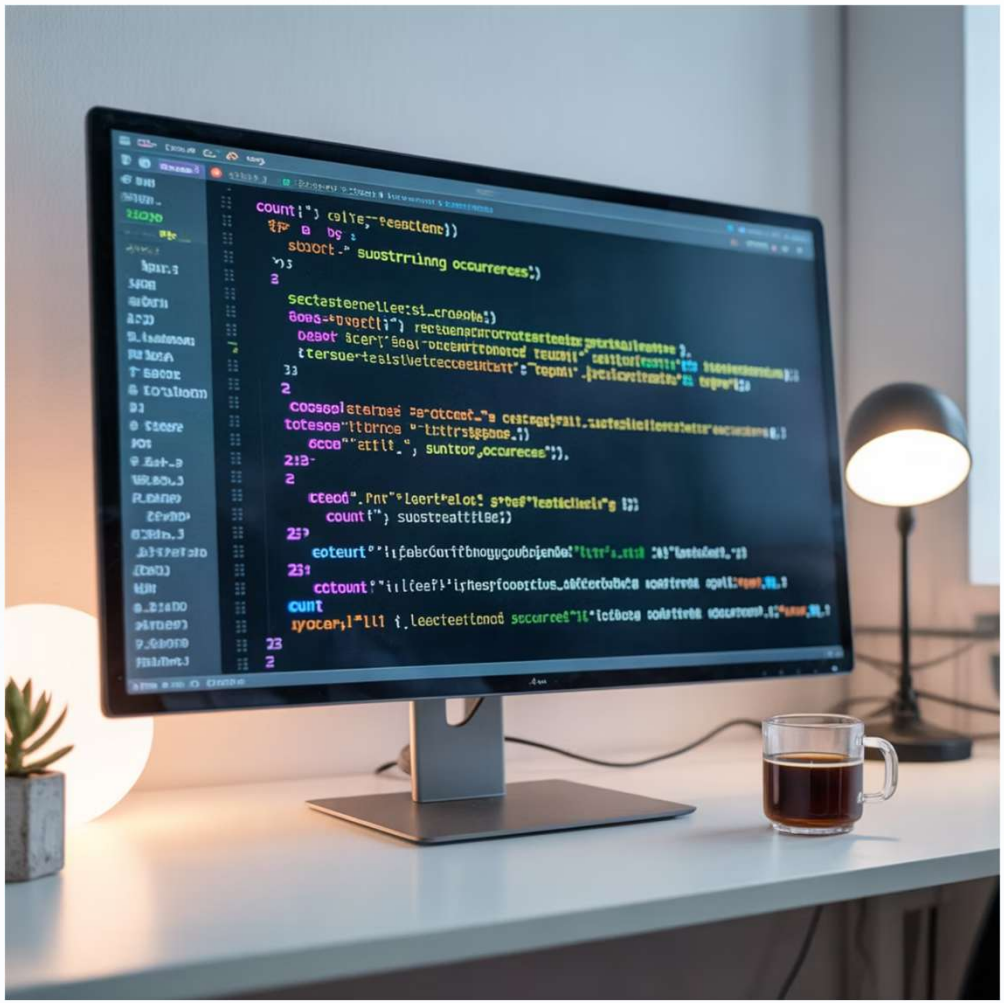
Like `startswith()`, you can check for multiple suffixes:

```
file = "vacation.jpg"
if file.endswith((".jpg", ".jpeg", ".png", ".gif")):
    print("It's an image file")
```

Checking Start/End: startswith(), endswith()

Practical Applications

```
import os
# File filtering
def get_python_files(directory):
 """Get all Python files in a directory."""
 return [f for f in os.listdir(directory)
 if f.endswith(".py")]

# URL validation
def is_valid_url(url):
 """Check if a URL starts with http:// or https://."""
 return url.startswith(("http://", "https://"))

print(is_valid_url("https://python.org")) # True
print(is_valid_url("www.example.com")) # False

# Email domain check
def is_company_email(email):
 """Check if an email belongs to the company domain."""
 return email.lower().endswith("@company.com")

print(is_company_email("user@company.com")) # True
print(is_company_email("user@gmail.com")) # False

# Command processing
def process_command(cmd):
 """Process different types of commands."""
 if cmd.startswith("GET "):
 return f"Retrieving: {cmd[4:]}"
 elif cmd.startswith("SET "):
 return f"Setting: {cmd[4:]}"
 elif cmd.startswith("DEL "):
 return f"Deleting: {cmd[4:]}"
 else:
 return "Unknown command"

print(process_command("GET user/profile")) # Retrieving: user/profile
print(process_command("SET theme/dark")) # Setting: theme/dark

# String parsing with optional parameters
text = "The quick brown fox jumps over the lazy dog"
print(text.endswith("fox", 0, 19)) # True - checks only up to "fox"
print(text.startswith("brown", 4)) # False - "brown" doesn't start at position
4
print(text.startswith("brown", 10)) # True - "brown" starts at position 10
```



Character Test Functions: isalpha(), isdigit(), isalnum()

A

isalpha()

The `isalpha()` method checks if all characters in a string are alphabetic (letters a-z, A-Z and Unicode letters). It returns `True` if all characters are alphabetic and there is at least one character, otherwise `False`.

```
print("Hello".isalpha())      # True
print("Hello123".isalpha())   # False
print("Hello World".isalpha()) # False (contains
a space)
print("".isalpha()) # False (empty string)

# Use case: validating names
name = "John"
if name.isalpha():
    print("Valid name")  # This will print
```

#

isdigit()

The `isdigit()` method checks if all characters in a string are digits (0-9). It returns `True` if all characters are digits and there is at least one character, otherwise `False`.

```
print("12345".isdigit()) # True
print("123.45".isdigit() # False (contains a
period)
print("123a".isdigit()) # False (contains a
letter)
print("".isdigit()) # False (empty string)

# Use case: validating numeric input
age = input("Enter your age: ")
if age.isdigit():
    age = int(age)
    print(f"You are {age} years old")
else:
    print("Please enter a valid number")
```

⌨

isalnum()

The `isalnum()` method checks if all characters in a string are alphanumeric (letters or digits). It returns `True` if all characters are alphanumeric and there is at least one character, otherwise `False`.

```
print("Hello123".isalnum())    # True
print("Hello 123".isalnum())   # False (contains
a space)
print("Hello-123".isalnum())   # False (contains
a hyphen)
print("".isalnum())            # False (empty
string)

# Use case: validating usernames
username = "User123"
if username.isalnum():
    print("Valid username")  # This will print
```

Character Test Functions: isalpha(), isdigit(), isalnum()

Additional Character Test Methods

- **isspace()**

  Checks if all characters are whitespace characters (spaces, tabs, newlines).

```
print("   \t\n".isspace())   # True
print("Hello ".isspace())    # False
```

- **islower()**

  Checks if all alphabetic characters are lowercase.

```
print("hello".islower())     # Trueprint("Hello".islower())     #
Falseprint("hello123".islower())  # True (digits are ignored)
```

- **isupper()**

  Checks if all alphabetic characters are uppercase.

```
print("HELLO".isupper())     # True
print("Hello".isupper())     # False
print("HELLO123".isupper())  # True (digits are ignored)
```

- **istitle()**

  Checks if the string is title-cased (all words start with an uppercase letter and the rest are lowercase).

```
print("Hello World".istitle())     # True
print("Hello world".istitle())    # False
```

- **isdecimal()**

  Checks if all characters are decimal characters (0-9). Stricter than isdigit().

```
print("12345".isdecimal())        # True
print("²³".isdecimal())           # False (superscripts)
```

- **isidentifier()**

  Checks if the string is a valid Python identifier (could be used as a variable name).

```
print("variable_name".isidentifier()) # True
print("123variable".isidentifier()) # False
```

String Formatting: format(), f-strings

Python offers several powerful ways to format strings, allowing you to incorporate variables and expressions into string literals. These formatting techniques provide flexibility and readability when constructing strings for output or display.

Old-style % Formatting

The oldest method, similar to C's printf(). While still supported, it's generally considered outdate

```
name = "Alice"
age = 30
print("Name: %s, Age: %d" % (name, age))
# Output: Name: Alice, Age: 30
```

str.format() Method

A more versatile and readable method introduced in Python 2.6. It uses curly braces as placeholders.

```
name = "Bob"
age = 25
print("Name: {}, Age: {}".format(name, age))
# Output: Name: Bob, Age: 25

# With positional arguments
print("{1} is {0} years old".format(age, name))
# Output: Bob is 25 years old

# With named arguments
print("Name: {n}, Age: {a}".format(n=name, a=age))
# Output: Name: Bob, Age: 25
```

f-strings (Formatted String Literals)

Introduced in Python 3.6, this is the most concise and often most readable approach, allowing direct embedding of expressions inside string literals.

```
name = "Charlie"
age = 35
print(f"Name: {name}, Age: {age}")
# Output: Name: Charlie, Age: 35

# With expressions
print(f"In 5 years, {name} will be {age + 5}")
# Output: In 5 years, Charlie will be 40

# With formatting specifications
pi = 3.14159
print(f"Pi rounded to 2 decimals: {pi:.2f}")
# Output: Pi rounded to 2 decimals: 3.14
```

String Formatting: format(), f-strings

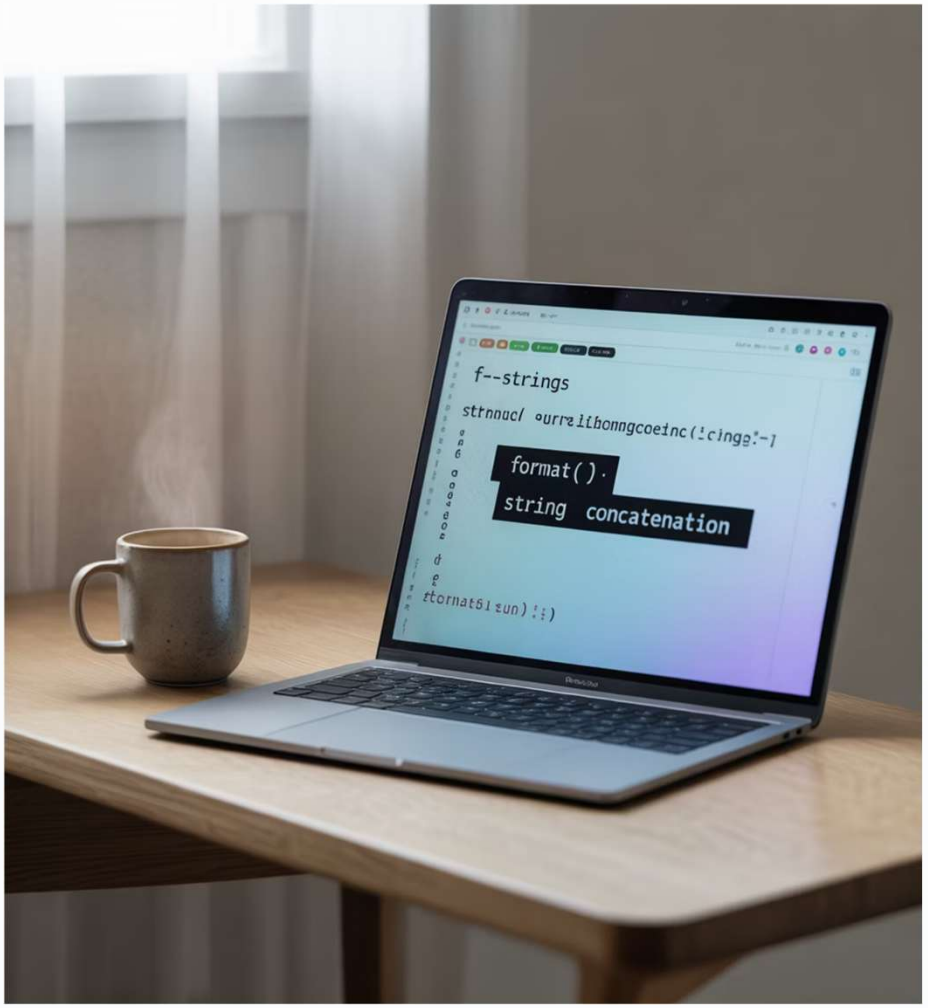
Advanced Formatting Features

```
# Number formatting
amount = 1234567.89
print(f"Amount: ${amount:,.2f}")
# Output: Amount: $1,234,567.89

# Width and alignment
for i in range(1, 11):
 print(f"{i:2d} {i*i:3d} {i*i*i:4d}")
# Output:
# 1 1 1
# 2 4 8
# 3 9 27
# ...

i = 7
print(f"{i:2d}")   # Output: ' 7'  (1 space, then 7 — total width = 2)
print(f"{i:3d}")   # Output: '  7' (2 spaces, then 7 — total width = 3)
```

If the number is **shorter** than the field width, it's **right-aligned** by default, and Python adds **spaces** in front.

Code	Output	Explanation
f"{7:2d}"	' 7'	Width = 2, right-aligned
f"{49:3d}"	' 49'	Width = 3, right-aligned
f"{512:4d}"	' 512'	Width = 4, right-aligned

```
# Date formatting
from datetime import datetime
now = datetime.now()
print(f"Current date: {now:%Y-%m-%d %H:%M}")
# Output: Current date: 2023-09-15 14:30
```



Escape Characters and Raw Strings

Escape characters in Python strings are special sequences that begin with a backslash (\) and represent characters that might be difficult or impossible to type directly. Raw strings, on the other hand, are a way to treat backslashes as literal characters rather than the start of escape sequences.

Common Escape Sequences

- **\n** - Newline
- **\t** - Horizontal tab
- **\r** - Carriage return
- **** - Backslash character
- **\'** - Single quote
- **\"** - Double quote
- **\b** - Backspace
- **\f** - Form feed
- **\uXXXX** - Unicode character with 16-bit hex value XXXX
- **\UXXXXXXXX** - Unicode character with 32-bit hex value XXXXXXXX
- **\ooo** - Character with octal value ooo
- **\xhh** - Character with hex value hh

Using Escape Sequences

```
# Newlines and tabs
print("Line 1\nLine 2\nLine 3")
# Output:
# Line 1
# Line 2
# Line 3
print("Column 1\tColumn 2\tColumn 3")
# Output: Column 1    Column 2    Column 3
# Quotes inside strings
print("He said, \"Hello!\"")
# Output: He said, "Hello!"
print('It\'s a beautiful day.')
# Output: It's a beautiful day.
# Backslash as a literal character
print("C:\\Users\\Documents")
# Output: C:\Users\Documents
```

Summary: String Power in Python

We've explored a comprehensive set of string functions and methods in Python, demonstrating their versatility and power for text processing. Here's a summary of what we've covered:

String Fundamentals

- Strings are immutable sequences of Unicode characters
- Created using single, double, or triple quotes
- Support indexing and slicing for accessing characters
- Can be concatenated with + and repeated with *

String Manipulation

- Case manipulation: upper(), lower(), title()
- Whitespace handling: strip(), lstrip(), rstrip()
- Finding and replacing: find(), rfind(), replace()
- Breaking and joining: split(), join(), partition()

String Analysis

- Content checking: startswith(), endswith()
- Character testing: isalpha(), isdigit(), isalnum()
- Occurrence counting: count()

String Formatting

- Format method: str.format()
- F-strings for direct embedding of expressions
- Escape characters and raw strings

The power of Python's string handling capabilities lies in their expressiveness, flexibility, and ease of use. With these tools, you can tackle a wide range of text processing tasks, from simple data cleaning to complex natural language processing.

Key Takeaways

1. **Immutability is fundamental:** Remember that strings in Python are immutable. Every string operation that appears to modify a string actually creates a new string object. This has implications for performance and memory usage, especially when working with large strings.
2. **String methods are non-destructive:** Methods like `upper()`, `replace()`, and `strip()` return new strings rather than modifying the original. Always capture their return values.
3. **Choose the right tool:** Python offers multiple ways to accomplish the same task. For example, you can split a string using `split()`, `partition()`, or slicing. Choose the method that best fits your specific requirements and makes your code most readable.
4. **String operations can be chained:** You can chain multiple string methods together, such as `text.strip().lower().replace(" ", "_")`, creating concise and powerful operations.
5. **Mastering string manipulation:** is essential for data cleaning, parsing, web development, and almost every area of programming. The time invested in understanding these functions will pay dividends across your Python projects.

As you continue your Python journey, you'll find that these string functions form a core part of your programming toolkit, enabling you to efficiently process and manipulate text data in countless applications.