Algonex

PYTHON

Python String

Python

Introduction to Python Strings

- A Python string is an immutable sequence of characters enclosed in single, double, or triple quotes.
- It is used to store and manipulate textual data.
- Strings support indexing, slicing, and various built-in methods for formatting, searching, and modifying text efficiently in programming and data processing tasks.

Creating Strings

- You can create strings in Python by enclosing text within single or double quotes.
- Both single and double quotes are interchangeable for creating strings, but they must match at the beginning and end of the string.
- Triple quotes are used for multiline strings

```
single_quoted = 'This is a single-quoted string.'
double_quoted = "This is a double-quoted string."

triple_quoted = '''This is a
multiline string.'''
```

Note: in the multiline string result, the line breaks are inserted at the same position as in the code.

Assign String to a Variable

```
my_string = "Intensity Coding"
print(my_string)
```

```
Intensity Coding
```

Triple Quotes

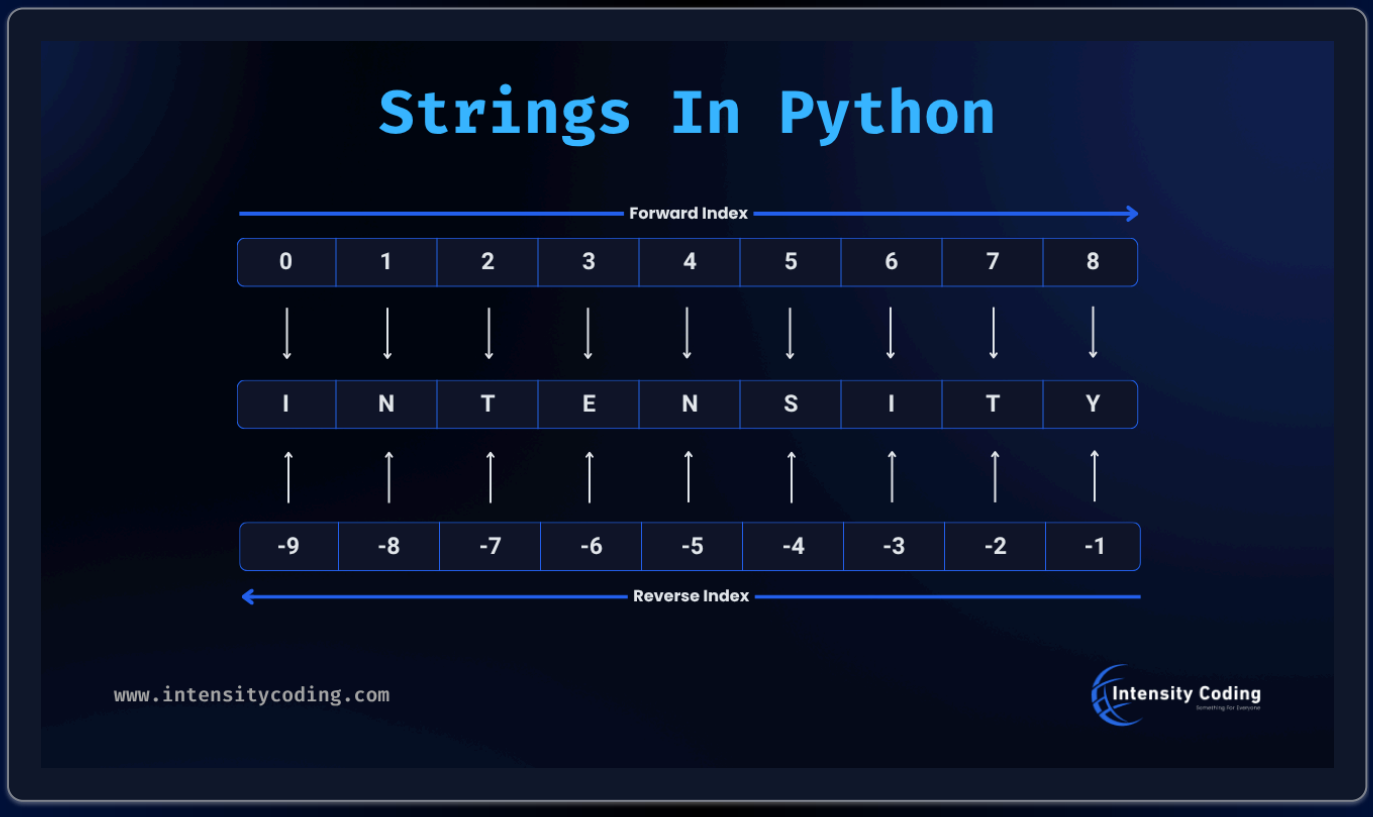
- Python's triple quotes allow defining multi-line strings, preserving all formatting, including newlines (\n), tabs (\t), and other special characters. This makes them useful for handling long texts, documentation, or formatted output.
- It is works with Both Single (''') and Double (""") Quotes.
- The raw string (r") ensures backslashes () are treated literally, making it useful for file paths or regex patterns.

```
# Using triple quotes to define a multi-line string
multi_line_str = """This is a long string that spans
multiple lines and includes special characters like
TAB (\t) and NEWLINEs (\n) which appear as they are."""
print(multi_line_str)

# Normal string with escape characters
print('C:\\nowhere')  # Output: C:\nowhere

# Raw string to ignore escape sequences
print(r'C:\\nowhere')  # Output: C:\\nowhere
```

```
This is a long string that spans
multiple lines and includes special characters like
TAB (	) and NEWLINEs (
) which appear as they are.
C:\nowhere
C:\\nowhere
```

Strings as Arrays

- In Python, strings can be treated like arrays or sequences of characters in many ways. However, it's important to note that Python strings are immutable, meaning you cannot modify them directly.
- Strings in Python are arrays of bytes representing unicode characters and It does not have a character data type, a single character is simply a string with a length of 1.
- You can access individual characters in a string using indexing and perform various operations on them as if they were elements of an array. Here's how you can treat a string as an array in Python.



Accessing Individual Characters

- To access a specific character in a string, you can use square brackets [] with the index of the character you want to access.

```
my_string = "Intensity Coding"
first_char = my_string[0]  # Access the first character (index 0)
print(first_char)  # Output: 'I'

third_char = my_string[2]  # Access the third character (index 2)
print(third_char)  # Output: 't'
```

```
I
t
```

Negative Indexing

- Python also allows you to use negative indexing to access characters from the end of the string. -1 represents the last character, -2 the second-to-last character, and so on.

```
my_string = "Intensity Coding"
last_char = my_string[-1]  # Access the last character
print(last_char)  # Output: 'g'

second_last_char = my_string[-2]  # Access the second-to-last character
print(second_last_char)  # Output: 'n'
```

```
g
n
```

Slicing Strings

- You can also use slicing to extract a portion of a string. Slicing allows you to specify a range of indices and obtain a substring:
- It is done by specifying a start and end index within square brackets. The slice includes the character at the start index but excludes the character at the end index.

```
my_string = "Intensity Coding"
substring = my_string[2:5]  # Get characters from index 2 to 4
(exclusive)
print(substring)  # Output: 'ten'
```

```
ten
```

Start index to slice from the beginning:

- When you omit the start index in slicing, Python automatically starts from the beginning of the string (index 0) and continues up to the specified end index (exclusive).

```
my_string = "Intensity Coding"
print(my_string[:4])
```

```
Inte
```

End index to slice to the end :

- If you omit the end index, Python slices from the given start position to the end of the string.

```
my_string = "Intensity Coding"
```

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

```
tensity Coding
```

Negative Indexing

- Negative indices count from the end of the string, where -1 represents the last character.

```
my_string = "Intensity Coding"
print(my_string[-5:-2])
```

```
odi
```

Concatenation of string

- You can concatenate (combine) strings using the + operator:

```
# Merge variable first_name with variable last_name into variable
full_name:
first_name = "John"
last_name = "Doe"
full_name = first_name + last_name  # 'JohnDoe'
print(full_name)
```

```
JohnDoe
```

```
# #To add a space between them, add a " ":
first_name = "John"
```

```
last_name = "Doe"
full_name = first_name + " " + last_name  # 'John Doe'
print(full_name)
```

```
John Doe
```

Repetition of string

- You can repeat a string using the * operator:

```
str1 = "Python "
result = str1 * 3  # Result: "Python Python Python "
print(result)
```

```
Python Python Python
```

Iterating Over Characters

- You can use a for loop to iterate over each character in a string

```
my_string = "Intensity Coding"
for char in my_string:
    print(char)
```

```
I
n
t
e
n
s
```

```
i
t
y

C
o
d
i
n
g
```

String Length

- The len() function returns the length of a string (the number of characters).

```
a = "Intensity Coding" # it is also include blank space
print(len(a))
```

```
16
```

max() and min() Method in String

- The Python string max() method compares the characters of a given string in a lexicographical manner and displays the maximum valued character among them. That means, the method internally compares the ASCII values of each character within the string and prints the highest valued character.

```
str = "Intensity Coding"
print("Max character: " + max(str))
```

```
Max character: y
```

- The Python string min() method compares the characters of a string lexicographically and displays the minimum valued character. That means, the method internally compares the ASCII values of each character within the string and prints the lowest valued character.

```
str = "IntensityCoding"
print("Min character: " + min(str))
```

```
Min character: C
```

Checking if a Character Exists

- To check if a certain phrase or character is present in a string, we can use the keyword `in`.

```
text = "Intensity Coding"
character_to_check = "o"

if character_to_check in text:
    print(f"The character '{character_to_check}' exists in the string.")
else:
    print(f"The character '{character_to_check}' does not exist in the
string.")
```

```
The character 'o' exists in the string.
```

```
txt = "Python is easy to learn."
print("easy" in txt)
```

```
True
```

Checking if NOT a Character Exists

- To check if a certain phrase or character is NOT present in a string, we can use the keyword not in.

```
text = "Intensity Coding"
character_to_check = "z"

if character_to_check not in text:
    print(f"The character '{character_to_check}' does not exist in the string.")
else:
    print(f"The character '{character_to_check}' exists in the string.")
```

```
The character 'z' does not exist in the string.
```

```
txt = "Python is easy to learn."
print("good" not in txt)
```

```
True
```

Special Operators of String

- Assume string variable a holds 'Hello' and variable b holds 'Python', then

Operator	Description	Example
`+`	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
`*`	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give HelloHello
`[]`	Slice - Gives the character from the given index	a[1] will give e
`[ : ]`	Range Slice - Gives the characters from the given range	a[1:4] will give ell
`in`	Membership - Returns true if a character exists in the given string	H in a will give 1
`not in`	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

Escape Character in String

- In Python, the escape character is used to insert special characters or escape sequences into strings.
- To do this, simply add a backslash (`\`) before the character you want to escape.
- It will give an error if you use double quotes inside a string that is surrounded by double quotes

Escape character	Result
`\'`	Single Quote
`\\`	Backslash
`\n`	New Line
`\r`	Carriage Return
`\t`	Tab
`\b`	Backspace
`\f`	Form Feed

Escape character	Result
`\ooo`	Octal value
`\xhh`	Hex value

```
txt = 'It\'s beautiful.'
print(txt)
```

```
It's beautiful.
```

```
txt = "This is a backslash: \\"
print(txt)
```

```
This is a backslash: \
```

```
txt = "Hello\nWorld!"
print(txt)
```

```
Hello
World!
```

```
txt = "Hello\rWorld!"
print(txt)
```

```
Hello
World!
```

```
txt = "Hello \bWorld!"
print(txt)
```

```
Hello World!
```

```
txt = "\110\145\154\154\157"
print(txt)
```

```
Hello
```

```
txt = "\x48\x65\x6c\x6c\x6f"
print(txt)
```

```
Hello
```

Python Raw Strings (`r''` or `r""`)

- In Python, **raw strings** are string literals prefixed with `r` or `R`. They treat **backslashes** (`\`) **as literal characters**, not as escape characters.

```
# Raw String vs Normal String Comparison
normal_str = "Hello\nWorld"  # Contains a newline
raw_str = r"Hello\nWorld"    # \n is treated literally

print("Normal:", normal_str)  # Output: Hello
                              #         World
print("Raw:   ", raw_str)     # Output: Hello\nWorld
```

```
Normal: Hello
World
Raw:    Hello\nWorld
```

String Formatting in Python

- String formatting in Python is a way to manipulate and format strings to include variables, numbers, or other data within a text. There are several methods and techniques for string formatting in Python, For example :

```
With the help of % Operator.
With the help of format() string method.
With the help of string literals, called f-strings.
With the help of String Template Class
With the help of center() string method.
```

With the help of % Operator:

Point	Description
Use Case	The `%` operator is an **old method** for formatting strings by inserting values into placeholders.
Syntax	`"string with % placeholders" % (values)`
Parameter	**Placeholders** are represented by `%` followed by a type specifier (e.g., `%s`, `%d`, `%f`).

Useful Information

No.	Point	Description
1	Uses `%` for string formatting	The `%` operator replaces placeholders in a string with actual values.

No.	Point	Description
2	Supports multiple data types	Placeholders like `%s` (string), `%d` (integer), and `%f` (float) can be used.
3	Allows precision control	Floating-point numbers can be formatted using `%a.bf`, where `a` is width and `b` is precision.

Explanation

- The `%` operator (also called the **string-formatting operator**) is an **older** way to format strings in Python.
- It works by replacing special placeholders (`%s`, `%d`, `%f`, etc.) inside a string with specified values.
- Although **f-strings** and `.format()` are preferred in modern Python, `%` formatting is still useful for certain cases.

Format operator %.

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e

Format Symbol	Conversion
%G	the shorter of %f and %E

```
# Basic String Formatting using % Operator
print("The %s is a powerful method for %s!" % ("intensity coding",
"Python programming"))
# Output: "The intensity coding is a powerful method for Python
programming!"

# Injecting Multiple Strings
action = "explored"
print("She %s and %s new concepts daily!" % ("learned", action))
# Output: "She learned and explored new concepts daily!"

# Formatting Floating-Point Numbers (Precision Handling)
pi_value = 3.141592
print("The value of pi rounded to 4 decimal places: %5.4f" % pi_value)
# Output: "The value of pi rounded to 4 decimal places: 3.1416"

# Formatting Multiple Data Types
variable = 42
formatted_string = "Variable as integer = %d \nVariable as float = %f" %
(variable, variable)
print(formatted_string)
# Output:
# Variable as integer = 42
# Variable as float = 42.000000

# Combining String and Number Formatting
score = 98.75
print("Your final score in %s is: %6.2f" % ("intensity coding", score))
# Output: "Your final score in intensity coding is:  98.75"
```

```
The intensity coding is a powerful method for Python programming!
She learned and explored new concepts daily!
The value of pi rounded to 4 decimal places: 3.1416
Variable as integer = 42
Variable as float = 42.000000
Your final score in intensity coding is:  98.75
```

With the help of format() string method.

Point	Description
Use Case	The `format()` method is used for flexible and readable string formatting.
Syntax	`"String with {} placeholders".format(values)`
Parameter	Values inside `.format()` replace `{}` placeholders in the string.

Useful Information

No.	Point	Description
1	Uses `{}` as placeholders	Curly braces `{}` define placeholders in a string.
2	Supports index-based and keyword-based insertion	`{0}, {1}` for positional, `{name}` for named placeholders.
3	Allows reusing values	A placeholder can be used multiple times.
4	Supports precision formatting	`{:.2f}` controls floating-point precision.
5	Handles different data types	`d` (integer), `f` (float), `s` (string), `b` (binary), `o` (octal), `x` (hex).

Explanation

- The `format()` method in Python is an **improved** way of formatting strings compared to the `%` operator.
  It allows inserting **values dynamically** into a string by using curly braces `{}` as placeholders.

```
# Basic String Formatting using format()
print("The {} method is essential in {}.".format("format()", "intensity coding"))
# Output: "The format() method is essential in intensity coding."

# Index-based Insertion
print("{2} {1} {0}".format("coding", "intensity", "Mastering"))
```

```python
# Output: "Mastering intensity coding"

# Keyword-based Formatting
print("Topic: {topic}, Difficulty: {level}".format(topic="intensity coding", level="Advanced"))
# Output: "Topic: intensity coding, Difficulty: Advanced"

# Reusing Inserted Objects
print("The first {p} was good, but the {p} {p} was tough.".format(p="lesson"))
# Output: "The first lesson was good, but the lesson lesson was tough."

# Floating-Point Precision
pi_value = 3.141592
print("Pi value rounded to 4 decimal places: {:.4f}".format(pi_value))
# Output: "Pi value rounded to 4 decimal places: 3.1416"

# Formatting Different Data Types
num = 42
binary = "{:b}".format(num)
octal = "{:o}".format(num)
hexadecimal = "{:x}".format(num)

print("Binary: {}, Octal: {}, Hex: {}".format(binary, octal, hexadecimal))
# Output: "Binary: 101010, Octal: 52, Hex: 2a"

# Combining String and Number Formatting
score = 95.6789
print("Your final score in {} is: {:6.2f}".format("intensity coding", score))
# Output: "Your final score in intensity coding is:  95.68"
```

```
The format() method is essential in intensity coding.
Mastering intensity coding
Topic: intensity coding, Difficulty: Advanced
The first lesson was good, but the lesson lesson was tough.
Pi value rounded to 4 decimal places: 3.1416
Binary: 101010, Octal: 52, Hex: 2a
Your final score in intensity coding is:  95.68
```

With the help of string literals, called f-strings.

Point	Description
Use Case	`f-strings` provide a concise way to format strings using expressions directly within `{}`.
Syntax	`f"String with {variables} and {expressions}"`
Parameter	Any valid Python expression inside `{}` gets evaluated and inserted into the string.

Useful Information

No.	Point	Description
1	Uses `f""` or `F""` prefix	Marks the string as an f-string.
2	Supports direct variable interpolation	No need for `.format()` or `%` formatting.
3	Allows expressions inside `{}`	Can include calculations, function calls, and conditions.
4	Supports floating-point precision	`{value:.2f}` controls decimal places.
5	Works with lambda expressions	Allows inline function evaluations.

Explanation

- Python **f-strings** (introduced in **PEP 498**) are the most modern way to format strings.
- They make **string interpolation** simpler by embedding expressions directly into the string using `{}`.
- This removes the need for `.format()` or `%` formatting and improves **readability**.

String Formatting Types

Format Type	Description	Example	Output
`:<`	Left aligns the result within the available space	`"{:<10}".format("text")`	`"text      "`
`:>`	Right aligns the result within the available space	`"{:>10}".format("text")`	`"      text"`
`:^`	Center aligns the result within the available space	`"{:^10}".format("text")`	`"   text   "`
`:=`	Places the sign to the leftmost position	`"{:=+6}".format(-42)`	`"-   42"`
`:+`	Use a plus sign for both positive and negative numbers	`"{:+d}".format(42)`	`"+42"`
`:-`	Use a minus sign for negative values only	`"{:-d}".format(-42)`	`"-42"`
`: `	Inserts an extra space before positive numbers	`"{: d}".format(42)`	`" 42"`
`:,`	Uses a comma as a thousand separator	`"{:,}".format(1000000)`	`"1,000,000"`
`:_`	Uses an underscore as a thousand separator	`"{:_}".format(1000000)`	`"1_000_000"`
`:b`	Binary format	`"{:b}".format(42)`	`"101010"`
`:c`	Converts value to corresponding Unicode character	`"{:c}".format(65)`	`"A"`
`:d`	Decimal format	`"{:d}".format(42)`	`"42"`
`:e`	Scientific format, lowercase `e`	`"{:e}".format(42)`	`"4.200000e+01"`
`:E`	Scientific format, uppercase `E`	`"{:E}".format(42)`	`"4.200000E+01"`
`:f`	Fixed point number format	`"{:f}".format(3.14159)`	`"3.141590"`

Format Type	Description	Example	Output
`:F`	Fixed point number format, uppercase for special cases	`"{:F}".format(float('nan'))`	`"NAN"`
`:g`	General format (compact scientific or decimal)	`"{:g}".format(42.0)`	`"42"`
`:G`	General format (uppercase `E` for scientific notation)	`"{:G}".format(4200000.0)`	`"4.2E+06"`
`:o`	Octal format	`"{:o}".format(42)`	`"52"`
`:x`	Hexadecimal format (lowercase)	`"{:x}".format(42)`	`"2a"`
`:X`	Hexadecimal format (uppercase)	`"{:X}".format(42)`	`"2A"`
`:n`	Number format	`"{:n}".format(1000)`	`"1000"`
`:%`	Percentage format	`"{:.2%}".format(0.75)`	`"75.00%"`

- These formatting options help in controlling how numbers, text, and symbols appear in output strings.

```
# Basic f-string Usage
topic = "intensity coding"
print(f"Learning {topic} improves problem-solving skills.")
# Output: "Learning intensity coding improves problem-solving skills."

# Arithmetic Operations in f-strings
a, b = 5, 10
print(f"The total difficulty score of {a} and {b} is {2 * (a + b)}.")
# Output: "The total difficulty score of 5 and 10 is 30."

# Using Functions inside f-strings
def coding_hours(level):
    return level * 2

level = 5
print(f"To master {topic}, you need {coding_hours(level)} hours of
practice.")
```

```python
# Output: "To master intensity coding, you need 10 hours of practice."

# Lambda Expressions in f-strings
print(f"Complexity level doubled: {(lambda x: x*2)(3)}")
# Output: "Complexity level doubled: 6"

# Floating-Point Precision
pi_value = 3.141592
print(f"Pi value rounded to 4 decimal places: {pi_value:.4f}")
# Output: "Pi value rounded to 4 decimal places: 3.1416"

# String Alignment with f-strings
language = "Python"
print(f"|{language:<10}|{language:^10}|{language:>10}|")
# Output:
# |Python    |  Python   |    Python|

# Mixing Data Types
score = 99
print(f"Your final score in {topic} is: {score}!")
# Output: "Your final score in intensity coding is: 99!"
```

```
Learning intensity coding improves problem-solving skills.
The total difficulty score of 5 and 10 is 30.
To master intensity coding, you need 10 hours of practice.
Complexity level doubled: 6
Pi value rounded to 4 decimal places: 3.1416
|Python    |  Python   |    Python|
Your final score in intensity coding is: 99!
```

With the help of String Template Class

Point	Description
Use Case	`Template` class provides a simplified way to substitute placeholders in strings.
Syntax	`Template("Hello $name!").substitute(name="World")`
Parameter	Uses `$placeholder` syntax for inserting values dynamically.

Useful Information

No.	Point	Description
1	Import from `string` module	`from string import Template` is required.
2	Uses `$variable_name` placeholders	Supports alphanumeric characters and underscores.
3	`$` can be escaped using `$$`	`$$` results in a single `$` in output.
4	`{}` allows additional text after placeholders	`Template("${name}123")` is valid.
5	`.substitute()` requires all placeholders to be provided	Use `.safe_substitute()` to avoid errors.

Explanation

- Python's `Template` **class** from the `string` module provides an alternative way to format strings.
  Instead of `{}` (used in `.format()` and f-strings), it uses `$variable_name` as placeholders.
- This method is particularly useful when working with **user-defined templates** or **external string formatting**.

```
from string import Template

# Using Template Class
template = Template("Welcome to $topic! Let's explore.")
result = template.substitute(topic="intensity coding")
print(result)
# Output: "Welcome to intensity coding! Let's explore."

# Multiple Substitutions
n1 = "intensity"
n2 = "coding"
template = Template("Mastering $n1 $n2 requires consistency.")
print(template.substitute(n1=n1, n2=n2))
# Output: "Mastering intensity coding requires consistency."

# Using Braces to Extend Placeholder
```

```python
template = Template("Learning ${skill}123 helps in problem-solving.")
print(template.substitute(skill="intensity coding"))
# Output: "Learning intensity coding123 helps in problem-solving."

# Escaping `$` Symbol
template = Template("Price of learning: $$99 for $duration months.")
print(template.substitute(duration=6))
# Output: "Price of learning: $99 for 6 months."

# Handling Missing Values Safely
template = Template("Your next step in $topic is $action.")
print(template.safe_substitute(topic="intensity coding"))
# Output: "Your next step in intensity coding is $action."

# Using Dictionary for Substitution
data = {"subject": "intensity coding", "platform": "Python"}
template = Template("Learn $subject using $platform.")
print(template.substitute(data))
# Output: "Learn intensity coding using Python."
```

```
Welcome to intensity coding! Let's explore.
Mastering intensity coding requires consistency.
Learning intensity coding123 helps in problem-solving.
Price of learning: $99 for 6 months.
Your next step in intensity coding is $action.
Learn intensity coding using Python.
```

With the help of center() string method.

Point	Description
Use Case	Aligns a string in the center of a given width by padding with spaces (or a specified character).
Syntax	`string.center(width, fillchar)`
Parameters	`width`(Required) The total width of the final string.
	`fillchar` (Optional) The character used for padding (default is space).

Useful Information

No.	Point	Description
1	Centers the string within a specified width	Adds equal padding on both sides.
2	Default padding character is space	A custom character can be used instead.
3	If `width` is smaller than string length	Returns the original string.
4	If extra space is needed for centering	Left padding is given priority.

Explanation

- The `center()` **method** places a string in the **middle of a specified width**.
- It adds **spaces (default)** or a **custom character** to both sides, ensuring the text remains **centered**.

```
# Centering a string with default space padding
string = "intensity coding"
width = 30

centered = string.center(width)
print(centered)
# Output: "       intensity coding       "

# Using a custom character for padding
centered_with_dash = string.center(width, "-")
print(centered_with_dash)
# Output: "-------intensity coding-------"

# When width is smaller than the string length
small_width = 10
print(string.center(small_width))
# Output: "intensity coding" (No changes)

# Centering an empty string
empty_string = ""
print(empty_string.center(10, "*"))
# Output: "**********" (Only padding)

# Uneven padding (left gets priority)
```

```
odd_width = 29
print(string.center(odd_width, "*"))
# Output: "*******intensity coding******"
```

```
       intensity coding
———————intensity coding———————
intensity coding
**********
*******intensity coding******
```

Python String Methods

Note: All string methods returns new values. They do not change the original string.

No.	Method	Description
1	**capitalize()**	Converts the first letter of the string to uppercase.
2	**casefold()**	Transforms the string into lowercase, ensuring case insensitivity.
3	**center()**	Aligns the string at the center with a specified width.
4	**count()**	Counts occurrences of a particular substring within the string.
5	**encode()**	Returns an encoded version of the string.
6	**endswith()**	Checks if the string concludes with a specific substring.
7	**expandtabs()**	Defines the tab size within the string.
8	**find()**	Locates a substring and returns its position in the string.
9	**format()**	Formats specified values within the string.
10	**format_map()**	Similar to `format()`, but uses a mapping object for formatting.
11	**index()**	Finds a substring and returns its position; raises an error if not found.
12	**isalnum()**	Returns `True` if all characters are alphanumeric.

No.	Method	Description
13	**isalpha()**	Checks if all characters belong to the alphabet.
14	**isascii()**	Returns `True` if all characters are ASCII characters.
15	**isdecimal()**	Returns `True` if the string consists of decimal digits.
16	**isdigit()**	Checks if all characters in the string are digits.
17	**isidentifier()**	Validates whether the string is a proper Python identifier.
18	**islower()**	Returns `True` if all letters in the string are lowercase.
19	**isnumeric()**	Checks if all characters represent numeric values.
20	**isprintable()**	Returns `True` if all characters are printable.
21	**isspace()**	Checks if the string consists solely of whitespace characters.
22	**istitle()**	Returns `True` if the string follows title case conventions.
23	**isupper()**	Checks if all letters are uppercase.
24	**join()**	Concatenates elements of an iterable with the string as a separator.
25	**ljust()**	Left-aligns the string within a specified width.
26	**lower()**	Converts all characters to lowercase.
27	**lstrip()**	Removes leading whitespace from the string.
28	**maketrans()**	Generates a translation table for string replacements.
29	**partition()**	Splits the string into three parts based on a specified separator.
30	**replace()**	Substitutes occurrences of a substring with another.
31	**rfind()**	Finds the last occurrence of a substring in the string.
32	**rindex()**	Returns the last position of a substring; raises an error if not found.
33	**rjust()**	Right-aligns the string within a defined width.
34	**rpartition()**	Similar to `partition()`, but searches from the end of the string.
35	**rsplit()**	Splits the string into a list using a separator, starting from the right.

No.	Method	Description
36	**rstrip()**	Removes trailing whitespace from the string.
37	**split()**	Divides the string into a list based on a separator.
38	**splitlines()**	Splits the string at line breaks and returns a list.
39	**startswith()**	Checks if the string starts with a particular substring.
40	**strip()**	Removes leading and trailing whitespace.
41	**swapcase()**	Swaps lowercase letters to uppercase and vice versa.
42	**title()**	Converts the first letter of each word to uppercase.
43	**translate()**	Returns a modified string based on a translation table.
44	**upper()**	Converts the string to uppercase.
45	**zfill()**	Pads the string with leading zeros to match a specified width.

1. capitalize()

Point	Description
Use Case	The `capitalize()` method returns a new string with the first character converted to uppercase, while the rest of the characters are converted to lowercase.
Syntax	`string.capitalize()`
Parameter Values	The method does not accept any parameters.

Useful Information

No.	Point	Description
1	No Effect on Non-Alphabets	If the first character is not a letter, the original string remains unchanged.
2	Already Capitalized First Letter	If the first letter is already uppercase, the method returns the string as is.

No.	Point	Description
3	No Modification to Original String	The method does not change the original string but returns a modified copy.
4	Converts Other Uppercase Letters to Lowercase	Only the first character is capitalized, and all other uppercase letters are converted to lowercase.

```
# Convert the first letter of the string to uppercase:
text = "intensity coding is awesome."
modified_text = text.capitalize()
print(modified_text)  # Output: Intensity coding is awesome.
```

```
Intensity coding is awesome.
```

2. casefold()

Point	Description
Use Case	The `casefold()` method returns a string where all characters are converted to lowercase. It is more aggressive than the `lower()` method, meaning it converts more characters to lowercase and improves string matching.
Syntax	`string.casefold()`
Parameter Values	The method does not accept any parameters.

```
# Convert the string to lowercase using casefold()

text = "INTENSITY CODING"
result = text.casefold()

print(result)  # Output: intensity coding
```

```
intensity coding
```

3. center()

Point	Description
Use Case	The `center()` method centers a string within a specified width, using a given character (default is space) to fill the extra space on both sides.
Syntax	`string.center(length, character)`
Parameter Values	`length` : Required. Specifies the total width of the returned string.
	`character` : Optional. The character used for padding. Default is a space `" "`.

Useful Information

No.	Point	Description
1	Uses a Specified Fill Character	If a character is provided as `fillchar`, the string is centered with that character as padding.
2	Default Fill Character is Space	If no `fillchar` is specified, spaces are used for padding.
3	Does Not Modify the Original String	If the specified width is less than the string's length, the original string is returned unchanged.
4	Fill Character Must be a Single Character	The method does not accept multi-character strings as `fillchar`; otherwise, a `TypeError` occurs.

```python
# Center align the string with '' as padding
text = "intensity coding"
centered_text = text.center(20)
print(centered_text)  # Output:   intensity coding
```

```
intensity coding
```

```python
# Center align the string with '*' as padding
text = "intensity coding"
centered_text = text.center(30, '*')
print(centered_text)  # Output: *******intensity coding*******
```

```
*******intensity coding*******
```

4. count()

Point	Description
Use Case	The `count()` method returns the number of times a specified substring appears in the given string.
Syntax	`string.count(value, start, end)`
Parameter	`value` : Required. The substring to search for.
	`start` : Optional. The position to start the search (default is `0`).
	`end` : Optional. The position to end the search (default is the end of the string).

Useful Information

No.	Point	Description
1	Substring Parameter is Mandatory	If the substring is not provided, a `TypeError` occurs.
2	Default Search Range	If `start` and `end` parameters are not provided, the entire string is searched.
3	Case-Sensitive Search	The `count()` method performs a case-sensitive search, meaning `"Apple"` and `"apple"` are treated differently.

No.	Point	Description
4	Does Not Modify Original String	The method returns the count but does not alter the original string.

```
# Count occurrences of 'i' in the string
text = "intensity coding in an intense world"
count_i = text.count("i")
print(count_i)  # Output: 5

# Count occurrences of 'in' between index 5 and 20
count_in = text.count("in", 5, 20)
print(count_in)  # Output: 2
```

```
5
2
```

5. encode()

Point	Description
Use Case	The `encode()` method encodes a string using a specified encoding format. If no encoding is provided, it defaults to UTF-8.
Syntax	`string.encode(encoding=encoding, errors=errors)`
Parameter	`encoding` : Optional. The encoding format to use. Default is `UTF-8`.
	`errors` : Optional. Specifies how to handle encoding errors. Possible values:
	`'backslashreplace'` – Replaces unencodable characters with a backslash sequence.
	`'ignore'` – Ignores characters that cannot be encoded.
	`'namereplace'` – Replaces characters with a descriptive text.
	`'strict'` – Default behavior, raises an error for unencodable characters.

Point	Description
	`'replace'` – Replaces unencodable characters with a `?`.
	`'xmlcharrefreplace'` – Replaces characters with their XML entity reference.

Useful Information

No.	Point	Description
1	Default Encoding is UTF-8	If no encoding is specified, the string is encoded using UTF-8.
2	Handles Encoding Errors	The `errors` parameter provides flexibility in handling encoding failures.
3	Converts String to Bytes	The `encode()` method returns a byte representation of the string.
4	Works with `decode()`	Encoded byte strings can be decoded back using the `decode()` method.

```
# Encode the string using UTF-8 (default)

text = "intensity cødïng"
encoded_text = text.encode()
print(encoded_text)  # Output: b'intensity c\xc3\xb8d\xc3\xafng'
```

```
b'intensity c\xc3\xb8d\xc3\xafng'
```

```
# Encode using ASCII with different error handling methods
text_special = "intensity cødïng"

print(text_special.encode(encoding="ascii", errors="backslashreplace"))
print(text_special.encode(encoding="ascii", errors="ignore"))
print(text_special.encode(encoding="ascii", errors="namereplace"))
print(text_special.encode(encoding="ascii", errors="replace"))
print(text_special.encode(encoding="ascii", errors="xmlcharrefreplace"))
```

```
b'intensity c\\xf8d\\xefng'
b'intensity cdng'
b'intensity c\\N{LATIN SMALL LETTER O WITH STROKE}d\\N{LATIN SMALL
LETTER I WITH DIAERESIS}ng'
b'intensity c?d?ng'
b'intensity c&#248;d&#239;ng'
```

6. endswith()

Point	Description
Use Case	The `endswith()` method checks whether a string ends with a specified substring. It returns `True` if the string ends with the given value; otherwise, it returns `False`.
Syntax	`string.endswith(value, start, end)`
Parameter	`value` : Required. The substring to check if the string ends with.
	`start` : Optional. The position where the search starts (default is `0`).
	`end` : Optional. The position where the search ends (default is the length of the string).

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` or `False` based on whether the string ends with the specified value.
2	Allows Search Within a Range	The `start` and `end` parameters define a specific portion of the string to check.
3	Case-Sensitive	The method is case-sensitive, meaning `"Coding."` and `"coding."` are treated differently.
4	Does Not Modify the Original String	The method only checks for a match and does not alter the string.

```python
# Check if the string ends with 'coding'

text = "intensity coding"
result = text.endswith("coding")
print(result)  # Output: True

# Check if the string ends with 'Intensity'
result = text.endswith("Intensity")
print(result)  # Output: False

# Check if a substring from position 5 to 15 ends with 'sity'
result = text.endswith("sity", 5, 15)
print(result)  # Output: False
```

```
True
False
False
```

7. expandtabs()

Point	Description
Use Case	The `expandtabs()` method replaces tab characters (`\t`) in a string with spaces, using a specified tab size. The default tab size is 8 spaces.
Syntax	`string.expandtabs(tabsize)`
Parameter	`tabsize` (Optional) : An integer defining the number of spaces for each tab. Default is `8`.

Useful Information

No.	Point	Description
1	Default Tab Size	If no `tabsize` is provided, the default size is `8` spaces.
2	Modifies Tabs Only	This method only replaces tab characters (`\t`) with spaces and does not affect other characters.

No.	Point	Description
3	Accepts Any Integer	The `tabsize` can be any integer, including `0`, which removes tab spacing entirely.
4	Does Not Modify Original String	The method returns a new string and does not change the original string.

```
# Original string with tab characters
text = "i\tn\tt\te\tn\ts\ti\tt\ty"

print(text) # Output:
print(text.expandtabs())
print(text.expandtabs(2))
print(text.expandtabs(4))
print(text.expandtabs(10))
```

```
i       n       t       e       n       s       i       t       y
i       n       t       e       n       s       i       t       y
i n t e n s i t y
i   n   t   e   n   s   i   t   y
i         n         t         e         n         s         i
t         y
```

8. find()

Point	Description
Use Case	The `find()` method searches for the first occurrence of a specified substring within a string. It returns the index of the first match or `-1` if the substring is not found.
Syntax	`string.find(value, start, end)`
Parameter	`value` (Required) : The substring to search for.
	`start` (Optional) : The starting position for the search. Default is `0`.

Point	Description
	`end` (Optional) : The ending position for the search. Default is the length of the string.

Useful Information

No.	Point	Description
1	Returns Index or `-1`	If the substring is found, it returns the index of the first occurrence; otherwise, it returns `-1`.
2	Similar to `index()` but Safe	Unlike `index()`, `find()` does not raise an exception if the substring is not found.
3	Can Search Within a Range	The `start` and `end` parameters allow searching within a specific portion of the string.
4	Case-Sensitive	The method is case-sensitive, meaning `"Coding"` and `"coding"` are treated differently.

```python
# Original string
text = "intensity coding"

# Find the first occurrence of "coding"
result = text.find("coding")
print(result)  # Output: 10

# Find the first occurrence of "i"
result = text.find("i")
print(result)  # Output: 0

# Find "i" within a range (start at index 5, end at 10)
result = text.find("i", 5, 10)
print(result)  # Output: 6

# If the substring is not found, it returns -1
result = text.find("python")
print(result)  # Output: -1
```

```
10
0
```

```
6
-1
```

9. format()

Point	Description
Use Case	The `format()` method allows inserting values into a string at designated placeholders `{}`. It provides flexibility in formatting numbers, strings, and other data types.
Syntax	`string.format(value1, value2, ...)`
Parameter	`value1, value2, ...` : Required. One or more values that should be formatted and inserted into the string.

Useful Information

No.	Point	Description
1	Supports Named, Numbered, and Empty Placeholders	You can use `{name}`, `{0}`, or `{}` as placeholders.
2	Allows Advanced Formatting	You can specify alignment, number formatting, and padding.
3	Works with Different Data Types	Supports integers, floats, strings, and even objects.
4	Does Not Modify Original String	Returns a new formatted string instead of modifying the original one.

```python
# Original string
text = "Welcome to {}!"

# Using format() with an empty placeholder
formatted_text = text.format("intensity coding")
print(formatted_text)  # Output: Welcome to intensity coding!

# Using numbered placeholders
formatted_text = "This is {0}, and it is {1}.".format("intensity coding",
```

```
"awesome")
print(formatted_text)  # Output: This is intensity coding, and it is
awesome.

# Using named placeholders
formatted_text = "Welcome to {topic}!".format(topic="intensity coding")
print(formatted_text)  # Output: Welcome to intensity coding!

# Formatting numbers with two decimal places
price = 49
formatted_text = "The price is {:.2f} dollars.".format(price)
print(formatted_text)  # Output: The price is 49.00 dollars.

# Right-align text within a width of 20
formatted_text = "Title: {:>20}".format("intensity coding")
print(formatted_text)  # Output: Title:     intensity coding

# Binary representation of a number
formatted_text = "The binary version of {0} is {0:b}".format(5)
print(formatted_text)  # Output: The binary version of 5 is 101

# Percentage format
formatted_text = "Completion: {:.0%}".format(0.75)
print(formatted_text)  # Output: Completion: 75%
```

```
Welcome to intensity coding!
This is intensity coding, and it is awesome.
Welcome to intensity coding!
The price is 49.00 dollars.
Title:     intensity coding
The binary version of 5 is 101
Completion: 75%
```

10. format_map()

Point	Description
Use Case	The `format_map()` method is similar to `format()`, but it takes a dictionary as an argument and substitutes values based on keys.

Point	Description
Syntax	`string.format_map(dictionary)`
Parameter	`dictionary` : Required. A dictionary containing key-value pairs to be replaced in the string.

Useful Information

No.	Point	Description
1	Uses Dictionary Mapping	Unlike `format()`, it takes a dictionary directly instead of separate arguments.
2	Similar to `format()`	Works like `format()`, but provides a more direct way to use dictionaries.
3	No Key Checking	If a key is missing in the dictionary, it raises a `KeyError`.
4	Does Not Modify Original String	Returns a new formatted string.

```
# Using format_map() with a dictionary
data = {"topic": "intensity coding", "type": "learning"}
formatted_text = "Welcome to {topic}, a {type}
platform!".format_map(data)
print(formatted_text)  # Output: Welcome to intensity coding, a learning
platform!

# Using format_map() with a number
data = {"price": 49.99}
formatted_text = "The price is {price:.3f} dollars.".format_map(data)
print(formatted_text)  # Output: The price is 49.990 dollars.

# Handling missing keys (raises KeyError)
data = {"topic": "intensity coding"}
#formatted_text = "Welcome to {topic}, a {type}
platform!".format_map(data)
#print(formatted_text)  # Raises KeyError: 'type'
```

```
Welcome to intensity coding, a learning platform!
```

```
The price is 49.990 dollars.
```

11. index()

Point	Description
Use Case	The `index()` method finds the first occurrence of a substring within a string. It raises a `ValueError` if the substring is not found.
Syntax	`string.index(value, start, end)`
Parameter	`value` : Required. The substring to search for.
	`start` : Optional. The position where the search starts (default is `0`).
	`end` : Optional. The position where the search ends (default is the length of the string).

Useful Information

No.	Point	Description
1	Raises an Exception if Not Found	Unlike `find()`, it raises a `ValueError` instead of returning `-1`.
2	Case-Sensitive	The search is case-sensitive. `"Code"` and `"code"` are treated differently.
3	Allows Search Within a Range	The `start` and `end` parameters allow searching within a specific section.
4	Does Not Modify the Original String	It only searches for the substring and does not alter the string.

```python
# Finding the index of a substring
text = "intensity coding"
result = text.index("coding")
print(result)  # Output: 10

# Finding the first occurrence of a character
result = text.index("i")
print(result)  # Output: 0
```

```python
# Searching within a specific range
result = text.index("i", 2, 12)
print(result)  # Output: 6

# Handling a missing substring (raises ValueError)
# result = text.index("python")  # Raises ValueError: substring not found
```

```
10
0
6
```

12. isalnum()

Point	Description
Use Case	The `isalnum()` method checks if all characters in a string are alphanumeric (letters and numbers). It returns `True` if the string consists only of letters (`a-z`, `A-Z`) and digits (`0-9`), and `False` otherwise.
Syntax	`string.isalnum()`
Parameter	None

Useful Information

No.	Point	Description
1	Returns a Boolean Output	The method returns `True` if all characters are alphanumeric; otherwise, it returns `False`.
2	No Whitespace or Special Characters	If the string contains spaces, punctuation, or symbols, it returns `False`.
3	Works on Empty Strings	An empty string returns `False` because it has no characters to check.
4	Does Not Modify the Original String	The method only checks the condition and does not alter the string.

```python
# String with only letters and numbers
text = "intensity123"
result = text.isalnum()
print(result)  # Output: True

# String with spaces (not alphanumeric)
text = "intensity 123"
result = text.isalnum()
print(result)  # Output: False

# String with special characters (not alphanumeric)
text = "intensity@123"
result = text.isalnum()
print(result)  # Output: False

# Empty string check
text = ""
result = text.isalnum()
print(result)  # Output: False
```

```
True
False
False
False
```

13. isalpha()

Point	Description
Use Case	The `isalpha()` method checks if all characters in a string are alphabetic (letters only). It returns `True` if the string consists only of letters (`a-z`, `A-Z`), and `False` otherwise.
Syntax	`string.isalpha()`
Parameter	None

Useful Information

No.	Point	Description
1	Returns a Boolean Output	The method returns `True` if all characters are alphabetic; otherwise, it returns `False`.
2	No Digits or Special Characters	If the string contains numbers, spaces, or symbols, it returns `False`.
3	Works on Empty Strings	An empty string returns `False` because it has no characters to check.
4	Does Not Modify the Original String	The method only checks the condition and does not alter the string.

```
# String with only letters
text = "intensity"
result = text.isalpha()
print(result)  # Output: True

# String with numbers (not alphabetic)
text = "intensity123"
result = text.isalpha()
print(result)  # Output: False

# String with spaces (not alphabetic)
text = "intensity coding"
result = text.isalpha()
print(result)  # Output: False

# String with special characters (not alphabetic)
text = "intensity!"
result = text.isalpha()
print(result)  # Output: False

# Empty string check
text = ""
result = text.isalpha()
print(result)  # Output: False
```

```
True
False
False
```

```
False
False
```

14. isascii()

Point	Description
Use Case	The `isascii()` method checks if all characters in a string belong to the ASCII character set (0-127). It returns `True` if all characters are ASCII, otherwise `False`.
Syntax	`string.isascii()`
Parameter	None

Useful Information

No.	Point	Description
1	Returns a Boolean Output	Returns `True` if all characters are within the ASCII range (0-127); otherwise, it returns `False`.
2	Checks for Non-ASCII Characters	Unicode characters, emojis, and special symbols outside ASCII will return `False`.
3	Works on Empty Strings	An empty string returns `True` since it contains no characters.
4	Does Not Modify the Original String	The method only performs a check and does not alter the string.

```python
# String with only ASCII characters
text = "intensity123"
result = text.isascii()
print(result)  # Output: True

# String with spaces (ASCII characters are allowed)
text = "intensity coding"
result = text.isascii()
print(result)  # Output: True
```

```python
# String with an emoji (non-ASCII)
text = "intensity🔥"
result = text.isascii()
print(result)  # Output: False

# String with Unicode characters (non-ASCII)
text = "intensityΩ"
result = text.isascii()
print(result)  # Output: False

# Empty string check
text = ""
result = text.isascii()
print(result)  # Output: True
```

```
True
True
False
False
True
```

15. isdecimal()

Point	Description
Use Case	The `isdecimal()` method checks whether all characters in a string are decimal digits (0-9). It returns `True` if all characters are decimal digits, otherwise `False`.
Syntax	`string.isdecimal()`
Parameter	None

Useful Information

No.	Point	Description
1	Returns a Boolean Output	Returns `True` if all characters in the string are decimal numbers (0-9); otherwise, `False`.

No.	Point	Description
2	Unicode Decimal Digits Are Supported	Works with Unicode decimal characters (e.g., Arabic-Indic digits).
3	Does Not Accept Fractions or Subscripts	Characters like `²` (superscript 2) or `½` (fraction) return `False`.
4	Does Not Allow Negative Numbers	Strings containing `-` or `.` are not considered decimal numbers.
5	Works on Empty Strings	An empty string returns `False`.

```python
# String with only decimal digits
text = "123456"
result = text.isdecimal()
print(result)  # Output: True

# String with a non-decimal character
text = "1234a"
result = text.isdecimal()
print(result)  # Output: False

# Unicode decimal characters
a = "\u0030"  # Unicode for '0
result = a.isdecimal()
print(result)  # Output: True
```

```
True
False
True
```

16. isdigit()

Point	Description
Use Case	The `isdigit()` method checks whether all characters in a string are digits (0-9). It returns `True` if all characters are numeric digits, otherwise `False`.

Point	Description
Syntax	`string.isdigit()`
Parameter	None

Useful Information

No.	Point	Description
1	Returns a Boolean Output	Returns `True` if all characters in the string are digits (0-9); otherwise, `False`.
2	Recognizes Unicode Digits	Accepts Unicode digit characters like `²` (superscript 2).
3	Does Not Allow Decimal Points	Numbers containing `.` (e.g., `"10.5"`) return `False`.
4	Negative Numbers Are Not Considered Digits	Strings with `-` are not valid digit-only strings.
5	Empty Strings Return `False`	An empty string will return `False`.

```python
# String with only digits
text = "50800"
result = text.isdigit()
print(result)  # Output: True

# Unicode digit characters
a = "\u0030"  # Unicode for '0'
b = "\u00B2"  # Unicode for '²' (superscript 2)
print(a.isdigit())  # Output: True
print(b.isdigit())  # Output: True

# String with a non-digit character
text = "123a"
print(text.isdigit())  # Output: False
```

```
True
True
```

```
True
False
```

17 isidentifier()

Point	Description
Use Case	The `isidentifier()` method checks whether a string is a valid Python identifier (i.e., a valid variable name). It returns `True` if the string is a valid identifier; otherwise, `False`.
Syntax	`string.isidentifier()`
Parameter	None

Useful Information

No.	Point	Description
1	Valid Identifier Rules	Identifiers must only contain letters (`a-z`, `A-Z`), digits (`0-9`), and underscores (`_`).
2	Cannot Start with a Number	Identifiers cannot begin with a digit.
3	No Spaces Allowed	Identifiers cannot contain spaces.
4	Reserved Keywords Not Checked	It does not check whether a word is a reserved Python keyword.
5	Case-Sensitive	Python identifiers are case-sensitive.

```python
# Example 1: Checking valid identifiers
print("variable_name".isidentifier())  # Output: True
print("Data123".isidentifier())        # Output: True
print("_hiddenValue".isidentifier())   # Output: True

# Example 2: Invalid identifiers
print("123value".isidentifier())   # Output: False  (starts with digit)
print("data value".isidentifier()) # Output: False  (contains space)
print("value-name".isidentifier()) # Output: False  (contains hyphen)
```

```python
# Example 3: Checking Python keywords
print("class".isidentifier())  # Output: True
print("for".isidentifier())    # Output: True
```

```
True
True
True
False
False
False
True
True
```

18. islower()

Point	Description
Use Case	The `islower()` method checks whether all alphabetic characters in a string are lowercase. It returns `True` if all letters are lowercase; otherwise, it returns `False`.
Syntax	`string.islower()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all alphabetic characters are lowercase; otherwise, it returns `False`.
2	Ignores Non-Alphabet Characters	Numbers, symbols, and spaces do not affect the result. Only alphabetic characters are checked.
3	Case-Sensitive	If there is at least one uppercase letter, the method returns `False`.

No.	Point	Description
4	Does Not Modify the Original String	The method only checks the case of the string; it does not alter it.

```
# Check if all characters are lowercase
text = "intensity coding"
result = text.islower()
print(result)  # Output: True

# String contains an uppercase letter
text = "Intensity coding"
result = text.islower()
print(result)  # Output: False

# String with numbers and symbols
text = "hello 123!"
result = text.islower()
print(result)  # Output: True

# String with mixed case letters
text = "mynameisPeter"
result = text.islower()
print(result)  # Output: False
```

```
True
False
True
False
```

19. isnumeric()

Point	Description
Use Case	The `isnumeric()` method checks whether all characters in a string are numeric (0-9, Unicode numeric characters, and fractions). It returns `True` if all characters are numeric; otherwise, it returns `False`.
Syntax	`string.isnumeric()`

Point	Description
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all characters in the string are numeric; otherwise, it returns `False`.
2	Supports Unicode Numerics	Exponents (e.g., `²`, `¾`) and other numeric Unicode characters are considered valid.
3	Does Not Recognize Negative or Decimal Numbers	Strings like `"-1"` and `"1.5"` return `False` because `-` and `.` are not numeric characters.
4	Does Not Modify the Original String	The method only checks if the characters are numeric; it does not alter the string.

```
# Check if a string contains only numeric characters
text = "565543"
result = text.isnumeric()
print(result)  # Output: True

# Using Unicode numeric characters
a = "\u0030"  # Unicode for 0
b = "\u00B2"  # Unicode for ² (superscript 2)
print(a.isnumeric())  # Output: True
print(b.isnumeric())  # Output: True

# String with non-numeric characters
c = "10km2"
d = "-1"
e = "1.5"
print(c.isnumeric())  # Output: False (contains 'km')
print(d.isnumeric())  # Output: False (contains '-')
print(e.isnumeric())  # Output: False (contains '.')
```

```
True
True
True
```

```
False
False
False
```

20. isprintable()

Point	Description
Use Case	The `isprintable()` method checks whether all characters in a string are printable. It returns `True` if all characters are printable; otherwise, it returns `False`.
Syntax	`string.isprintable()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all characters in the string are printable; otherwise, it returns `False`.
2	Non-Printable Characters Cause `False`	Characters like newlines (`\n`), carriage returns (`\r`), and other control characters are considered non-printable.
3	Space and Special Characters Are Printable	Spaces, punctuation, and symbols (e.g., `#`, `!`, `?`) are considered printable characters.
4	Does Not Modify the Original String	The method only checks if the characters are printable; it does not alter the string.

```python
# String with only printable characters
text = "Intensity Coding"
result = text.isprintable()
print(result)  # Output: True

# String containing a newline character
text = "Intensity\nCoding"
```

```
result = text.isprintable()
print(result)  # Output: False
```

```
True
False
```

21. isspace()

Point	Description
Use Case	The `isspace()` method checks whether all characters in a string are whitespace characters. It returns `True` if all characters are whitespace; otherwise, it returns `False`.
Syntax	`string.isspace()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all characters in the string are whitespace; otherwise, it returns `False`.
2	Recognizes Various Whitespace Characters	Spaces (`' '`), tabs (`'\t'`), newlines (`'\n'`), vertical tabs (`'\v'`), form feeds (`'\f'`), and carriage returns (`'\r'`) are considered whitespace.
3	Returns `False` for Mixed Characters	If the string contains any non-whitespace character, it returns `False`.
4	Does Not Modify the Original String	The method only checks for whitespace characters; it does not alter the string.

```
# String with only spaces
text = "     "
result = text.isspace()
print(result)  # Output: True
```

```python
# String containing non-whitespace characters
text = "Intensity Coding"
result = text.isspace()
print(result)  # Output: False

# String with spaces and a tab
text = "    \t   "
result = text.isspace()
print(result)  # Output: True

# String with spaces and a newline
text = "   \n   "
result = text.isspace()
print(result)  # Output: True

# String containing "Intensity Coding" with spaces
text = "   Intensity Coding   "
result = text.isspace()
print(result)  # Output: False
```

```
True
False
True
True
False
```

22. istitle()

Point	Description
Use Case	The `istitle()` method checks whether a string follows title case, meaning each word starts with an uppercase letter and the remaining letters are lowercase. It returns `True` if all words follow this rule; otherwise, it returns `False`.
Syntax	`string.istitle()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all words in the string follow title case; otherwise, it returns `False`.
2	Ignores Symbols and Numbers	Non-alphabetic characters (e.g., `22`, `!`, `%`) do not affect the result.
3	Case-Sensitive	A word like `"HELLO"` is not considered title case, so `istitle()` will return `False`.
4	Does Not Modify the Original String	The method only checks the case format of the string; it does not alter it.

```
# Title case string
text = "Intensity Coding"
result = text.istitle()
print(result)  # Output: True

# Fully uppercase string
text = "INTENSITY CODING"
result = text.istitle()
print(result)  # Output: False

# Single correctly formatted word
text = "Intensity"
result = text.istitle()
print(result)  # Output: True

# String with a number
text = "22 Intensity Coding"
result = text.istitle()
print(result)  # Output: True

# String with special characters
text = "This Is %'!?"
result = text.istitle()
print(result)  # Output: True

# Incorrect title case (one lowercase letter in a word)
text = "Intensity coding"
result = text.istitle()
print(result)  # Output: False
```

```
True
False
True
True
True
False
```

23. isupper()

Point	Description
Use Case	The `isupper()` method checks whether all alphabetic characters in a string are uppercase. It returns `True` if all letters are uppercase; otherwise, it returns `False`.
Syntax	`string.isupper()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Returns Boolean Output	The method returns `True` if all alphabetic characters in the string are uppercase; otherwise, it returns `False`.
2	Ignores Non-Alphabetic Characters	Numbers, spaces, and symbols do not affect the result.
3	At Least One Letter Required	The method returns `False` if there are no alphabetic characters in the string.
4	Case-Sensitive	The method strictly checks for uppercase letters; mixed-case words return `False`.
5	Does Not Modify the Original String	The method only checks letter casing; it does not change the string.

```python
# Fully uppercase string
text = "INTENSITY CODING"
result = text.isupper()
print(result)  # Output: True

# String with mixed case
text = "Intensity Coding"
result = text.isupper()
print(result)  # Output: False

# String with numbers and symbols (still uppercase)
text = "INTENSITY 123!@#"
result = text.isupper()
print(result)  # Output: True

# String with lowercase letters
text = "INTENSITY coding"
result = text.isupper()
print(result)  # Output: False

# String with no letters
text = "1234!@#"
result = text.isupper()
print(result)  # Output: False
```

```
True
False
True
False
False
```

24. join()

Point	Description
Use Case	The `join()` method takes an iterable (e.g., list, tuple, dictionary) and joins its elements into a single string, using the specified string as a separator.
Syntax	`string.join(iterable)`

Point	Description
Parameter	**iterable**: Required. Any iterable object (e.g., list, tuple, dictionary) where all elements are strings.

Useful Information

No.	Point	Description
1	Separator is Required	The string before `.join()` acts as a separator between the elements of the iterable.
2	Works with Any Iterable	Lists, tuples, sets, and dictionaries can be joined, as long as all elements are strings.
3	Dictionaries Return Keys	When joining a dictionary, only the keys (not values) are included in the final string.
4	Does Not Modify the Original Iterable	The method creates a new string; the original iterable remains unchanged.

```
# Joining a list with a space separator
words = ["Intensity", "Coding"]
result = " ".join(words)
print(result)  # Output: Intensity Coding

# Joining a tuple with a hyphen separator
names = ("John", "Peter", "Vicky")
result = "-".join(names)
print(result)  # Output: John-Peter-Vicky

# Joining dictionary keys with a custom separator
info = {"name": "John", "country": "Norway"}
separator = "TEST"
result = separator.join(info)
print(result)  # Output: nameTESTcountry

# Joining characters with no separator
text = ["I", "n", "t", "e", "n", "s", "i", "t", "y"]
result = "".join(text)
print(result)  # Output: Intensity

# Attempting to join a list with non-string elements (causes error)
numbers = [1, 2, 3]
```

```
# result = "-".join(numbers)  # TypeError: sequence item 0: expected str
instance, int found
```

```
Intensity Coding
John-Peter-Vicky
nameTESTcountry
Intensity
```

25. ljust()

Point	Description
Use Case	The `ljust()` method left-aligns a string within a specified width, filling the remaining space on the right with a specified character (default is a space).
Syntax	`string.ljust(length, character)`
Parameter	**length**: Required. The total width of the returned string.
	character: Optional. A character to fill the extra space (default is space `" "`).

Useful Information

No.	Point	Description
1	Left Aligns the String	The original string is positioned to the left within the given width.
2	Fills Extra Space	The method adds the specified fill character (default is a space) to the right.
3	Returns a New String	The method does not modify the original string.
4	Truncates if Length is Less	If the specified length is shorter than the string length, the original string is returned unchanged.

```python
# Left align "Intensity" within 20 spaces (default fill character is space)
text = "Intensity"
result = text.ljust(20)
print(result, "Coding")
# Output: "Intensity           Coding"

# Left align using a custom fill character 'O'
text = "Intensity"
result = text.ljust(20, "O")
print(result)
# Output: "IntensityOOOOOOOOOO"

# When the specified width is smaller than the string length
text = "Intensity"
result = text.ljust(5)
print(result)
# Output: "Intensity" (unchanged)

# Left align with a different fill character
text = "Coding"
result = text.ljust(15, "-")
print(result)
# Output: "Coding--------"
```

```
Intensity           Coding
IntensityOOOOOOOOOO
Intensity
Coding--------
```

26. lower()

Point	Description
Use Case	The `lower()` method converts all uppercase letters in a string to lowercase. Symbols and numbers remain unchanged.
Syntax	`string.lower()`

Point	Description
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Converts to Lowercase	Only alphabetic characters are affected; symbols and numbers remain unchanged.
2	Returns a New String	The original string remains unchanged; the method returns a new string.
3	Useful for Case-Insensitive Comparisons	Often used to standardize text for case-insensitive operations.

```
# Convert text to lowercase
text = "Intensity CODING"
result = text.lower()
print(result)
# Output: intensity coding
```

```
intensity coding
```

27. lstrip()

Point	Description
Use Case	The `lstrip()` method removes leading (left-side) characters from a string. By default, it removes spaces.
Syntax	`string.lstrip(characters)`
Parameter	**characters**: Optional. A set of characters to remove from the beginning of the string.

Useful Information

No.	Point	Description
1	Removes Leading Characters	By default, it removes spaces from the left.
2	Custom Character Removal	You can specify a set of characters to remove.
3	Does Not Modify Original String	The method returns a new string.
4	Order of Characters in `characters` Does Not Matter	The method removes all occurrences of the specified characters, regardless of their order.

```
# Removing leading spaces
text = "     Intensity Coding     "
result = text.lstrip()
print(f"'{result}'")
# Output: 'Intensity Coding     '

# Removing specified leading characters
text = ",,,,,ssaaww.....Intensity Coding"
result = text.lstrip(",.asw")
print(result)
# Output: 'Intensity Coding'

# Removing specific letters
text = "aaaaaAAAaaathis is string example"
result = text.lstrip("aAth")
print(result)
# Output: 'is is string example
```

```
'Intensity Coding     '
Intensity Coding
is is string example
```

28. maketrans()

Point	Description
Use Case	The `maketrans()` method creates a translation table for use with `translate()`, allowing character replacements and deletions.
Syntax	`str.maketrans(x, y, z)`
Parameter	**x**: Required. A dictionary specifying mappings or a string of characters to be replaced.
	y: Optional. A string of replacement characters (same length as `x`).
	z: Optional. A string containing characters to remove from the string.

Useful Information

No.	Point	Description
1	Used With `translate()`	The table created by `maketrans()` is passed to `translate()` to modify strings.
2	Supports Character Mapping	You can map specific characters to new ones.
3	Can Remove Characters	The third argument (`z`) defines characters to be deleted.
4	Returns a Dictionary	`maketrans()` itself returns a dictionary of Unicode mappings.

```
# Replacing a single character
text = "Intensity Coding!"
table = str.maketrans("I", "Y")
print(text.translate(table))
# Output: 'Yntensity Coding!'

# Replacing multiple characters
text = "Intensity Coding!"
x = "ICd"
y = "XYz"
table = str.maketrans(x, y)
print(text.translate(table))
# Output: 'Xntensity Yozing!'

# Replacing and removing characters
```

```
text = "Intensity Coding!"
x = "ICd"
y = "XYz"
z = "!"
table = str.maketrans(x, y, z)
print(text.translate(table))
# Output: 'Xntensity Yozing'

# Viewing the translation dictionary
print(str.maketrans(x, y, z))
# Output: {73: 88, 67: 89, 100: 122, 33: None}
```

```
Yntensity Coding!
Xntensity Yozing!
Xntensity Yozing
{73: 88, 67: 89, 100: 122, 33: None}
```

29. partition()

Point	Description
Use Case	The `partition()` method searches for a specified substring and splits the string into a tuple of three parts.
Syntax	`string.partition(value)`
Parameter	**value**: Required. The substring to search for and split the string around.

Useful Information

No.	Point	Description
1	Returns a Tuple	The method returns a 3-element tuple: `(before_match, match, after_match)`.
2	Only Splits at First Match	It partitions at the first occurrence of the specified value.
3	Returns the Whole String if Not Found	If the value is not found, it returns `("original_string", "", "")`.

No.	Point	Description
4	Case-Sensitive	The search is case-sensitive, meaning "Coding" and "coding" are treated differently.

```
# Partitioning a string using a keyword
text = "Intensity Coding is fun!"
result = text.partition("Coding")
print(result)
# Output: ('Intensity ', 'Coding', ' is fun!')

# Partitioning when the keyword is absent
text = "Intensity Coding is fun!"
result = text.partition("Python")
print(result)
# Output: ('Intensity Coding is fun!', '', '')

# Splitting at a space
text = "Intensity Coding is fun!"
result = text.partition(" ")
print(result)
# Output: ('Intensity', ' ', 'Coding is fun!')

# Case-sensitive search
text = "Intensity Coding is fun!"
result = text.partition("coding")
print(result)
# Output: ('Intensity Coding is fun!', '', '')  # "coding" is not the
same as "Coding"
```

```
('Intensity ', 'Coding', ' is fun!')
('Intensity Coding is fun!', '', '')
('Intensity', ' ', 'Coding is fun!')
('Intensity Coding is fun!', '', '')
```

30. replace()

Point	Description
Use Case	The `replace()` method replaces a specified substring with another substring.
Syntax	`string.replace(oldvalue, newvalue, count)`
Parameter	**oldvalue**: Required. The substring to search for and replace.
	newvalue: Required. The substring to replace `oldvalue` with.
	count: Optional. The number of occurrences to replace (default: all occurrences).

Useful Information

No.	Point	Description
1	Replaces All by Default	If `count` is not specified, all occurrences of `oldvalue` are replaced.
2	Case-Sensitive	The search is case-sensitive, meaning "Coding" and "coding" are treated differently.
3	Returns a New String	The method does not modify the original string; it returns a modified copy.
4	Partial Replacement Possible	Using `count`, you can limit the number of replacements.

```
# Replace a word in a sentence
text = "Intensity Coding is fun!"
result = text.replace("Coding", "Learning")
print(result)
# Output: "Intensity Learning is fun!"

# Replace all occurrences of a character
text = "Intensity Coding Coding"
result = text.replace("Coding", "AI")
print(result)
# Output: "Intensity AI AI"

# Replace only the first occurrence
text = "Intensity Coding Coding"
result = text.replace("Coding", "AI", 1)
```

```
print(result)
# Output: "Intensity AI Coding"

# Case-sensitive replacement
text = "Intensity Coding is different from coding"
result = text.replace("coding", "AI")
print(result)
# Output: "Intensity Coding is different from AI"  # "Coding" remains
unchanged
```

```
Intensity Learning is fun!
Intensity AI AI
Intensity AI Coding
Intensity Coding is different from AI
```

31. rfind()

Point	Description
Use Case	The `rfind()` method finds the last occurrence of a substring in a string.
Syntax	`string.rfind(value, start, end)`
Parameter	**value**: Required. The substring to search for.
	start: Optional. The position to start searching (default: `0`).
	end: Optional. The position to stop searching (default: end of string).

Useful Information

No.	Point	Description
1	Returns `-1` if Not Found	If the substring is not found, it returns `-1`.
2	Searches from Right	Unlike `find()`, `rfind()` starts searching from the right but still returns the **index from the left**.

No.	Point	Description
3	Does Not Raise Error	Unlike `rindex()`, if the substring is not found, `rfind()` does not raise an error.

```
# Find the last occurrence of "Coding"
text = "Intensity Coding is part of Coding."
result = text.rfind("Coding")
print(result)
# Output: 28 (last occurrence of "Coding")

# Find the last occurrence of "e"
text = "Hello, welcome to Intensity Coding!"
result = text.rfind("e")
print(result)
# Output: 21 (last occurrence of "e")

# Find "e" within a specific range
text = "Hello, welcome to Intensity Coding!"
result = text.rfind("e", 5, 15)
print(result)
# Output: 13 (last "e" between index 5 and 15)

# If the substring is not found
text = "Intensity Coding"
result = text.rfind("Z")
print(result)
# Output: -1 (not found)
```

```
28
21
13
-1
```

32. rindex()

Point	Description
Use Case	The `rindex()` method finds the last occurrence of a substring in a string.
Syntax	`string.rindex(value, start, end)`
Parameter	**value**: Required. The substring to search for.
	start: Optional. The position to start searching (default: `0`).
	end: Optional. The position to stop searching (default: end of string).

Useful Information

No.	Point	Description
1	Raises an Exception	Unlike `rfind()`, if the substring is **not found**, `rindex()` raises a `ValueError`.
2	Searches from Right	Similar to `rfind()`, but still returns the **index from the left**.
3	Returns Last Occurrence	It finds the last occurrence of the substring in the string.

```
# Find the last occurrence of "Coding"
text = "Intensity Coding is part of Coding."
result = text.rindex("Coding")
print(result)
# Output: 28 (last occurrence of "Coding")

# Find the last occurrence of "e"
text = "Hello, welcome to Intensity Coding!"
result = text.rindex("e")
print(result)
# Output: 21 (last occurrence of "e")

# Find "e" within a specific range
text = "Hello, welcome to Intensity Coding!"
result = text.rindex("e", 5, 15)
print(result)
# Output: 13 (last "e" between index 5 and 15)

# If the substring is not found (Raises an Exception)
text = "Intensity Coding"
```

```
#result = text.rindex("Z")  # Raises ValueError
#print(result)
# Output: ValueError: substring not found
```

```
28
21
13
```

33. rjust()

Point	Description
Use Case	The `rjust()` method right-aligns a string by padding it with a specified character.
Syntax	`string.rjust(length, character)`
Parameter	**length**: Required. The total width of the resulting string.
	character: Optional. The character used for padding (default: `" "` space).

Useful Information

No.	Point	Description
1	Right Aligns String	Moves the original string to the **right** and fills the **left** with the specified character.
2	Default Padding	Uses **spaces** if no padding character is specified.
3	Does Not Modify	Returns a new string; the original string remains unchanged.

```
# Right align with spaces (default)
text = "Intensity"
result = text.rjust(20)
print(result, "Coding")
# Output: "           Intensity Coding"  (11 spaces on the left)
```

```python
# Right align with a custom character "O"
text = "Intensity"
result = text.rjust(20, "O")
print(result)
# Output: "OOOOOOOOOOIntensity"  (11 'O's on the left)

# When the length is smaller than or equal to the original string
text = "IntensityCoding"
result = text.rjust(10)
print(result)
# Output: "IntensityCoding"  (No change since it's already longer than
10)

# Right align with a symbol
text = "Coding"
result = text.rjust(15, "*")
print(result)
# Output: "*********Coding"  (9 '*' on the left)
```

```
           Intensity Coding
OOOOOOOOOOIntensity
IntensityCoding
*********Coding
```

34. rpartition()

Point	Description
Use Case	The `rpartition()` method searches for the **last occurrence** of a specified substring and splits the string into three parts.
Syntax	`string.rpartition(value)`
Parameter	**value**: Required. The substring to search for.

Useful Information

No.	Point	Description
1	Last Occurrence	Unlike `partition()`, which splits at the **first** occurrence, `rpartition()` splits at the **last** occurrence of the substring.
2	Returns a Tuple	The method always returns a **3-element tuple**: `('before', 'match', 'after')`.
3	Handles Missing Substrings	If the substring is **not found**, it returns `('', '', original_string)`.

```
# Search for the last occurrence of "Coding"
text = "Intensity Coding is part of Coding."
result = text.rpartition("Coding")
print(result)
# Output: ('Intensity Coding is part of ', 'Coding', '.')

# When the searched word is at the beginning
text = "Coding is essential, but Coding takes practice."
result = text.rpartition("Coding")
print(result)
# Output: ('Coding is essential, but ', 'Coding', ' takes practice.')

# If the substring is not found
text = "Intensity Coding"
result = text.rpartition("Python")
print(result)
# Output: ('', '', 'Intensity Coding')

# Using rpartition() to extract filename from a path
path = "/home/user/documents/Intensity_Coding.txt"
result = path.rpartition("/")
print(result)
# Output: ('/home/user/documents', '/', 'Intensity_Coding.txt')
```

```
('Intensity Coding is part of ', 'Coding', '.')
('Coding is essential, but ', 'Coding', ' takes practice.')
('', '', 'Intensity Coding')
('/home/user/documents', '/', 'Intensity_Coding.txt')
```

35. rsplit()

Point	Description
Use Case	The `rsplit()` method splits a string into a list, **starting from the right**.
Syntax	`string.rsplit(separator, maxsplit)`
Parameter	**separator** (Optional): The delimiter string to split on. Default is whitespace.
	maxsplit (Optional): The maximum number of splits to perform, starting from the **right**. Default is `-1` (all occurrences).

Useful Information

No.	Point	Description
1	Similar to `split()`	Works like `split()`, but performs the **splits from the right** instead of the left.
2	Useful for File Paths	Helps extract specific elements from structured data like file paths, logs, or version numbers.
3	`maxsplit` Control	Limits the number of splits, ensuring a controlled list size.

```
# Basic rsplit() without maxsplit (same as split())
txt = "Intensity, Coding, AI"
result = txt.rsplit(", ")
print(result)
# Output: ['Intensity', 'Coding', 'AI']

# Using maxsplit = 1, splits only once from the right
txt = "Intensity, Coding, AI"
result = txt.rsplit(", ", 1)
print(result)
# Output: ['Intensity, Coding', 'AI']

# Splitting a file path to extract filename
path = "/home/user/documents/report.pdf"
result = path.rsplit("/", 1)
print(result)
```

```
# Output: ['/home/user/documents', 'report.pdf']

# Extracting domain name from an email
email = "Intensity.coding@example.com"
result = email.rsplit("@", 1)
print(result)
# Output: ['Intensity.coding', 'example.com']

# Handling multiple spaces
txt = "  Hello   World   Python  "
result = txt.rsplit()
print(result)
# Output: ['Hello', 'World', 'Python']  (default separator is whitespace)
```

```
['Intensity', 'Coding', 'AI']
['Intensity, Coding', 'AI']
['/home/user/documents', 'report.pdf']
['Intensity.coding', 'example.com']
['Hello', 'World', 'Python']
```

36. rstrip()

Point	Description
Use Case	The `rstrip()` method removes **trailing characters** (characters at the end of a string). By default, it removes spaces.
Syntax	`string.rstrip(characters)`
Parameter	**characters** (Optional): A set of characters to remove from the **end** of the string. Default is whitespace.

Useful Information

No.	Point	Description
1	Removes only trailing characters	It **does not** remove characters from the beginning of the string.

No.	Point	Description
2	Default removes spaces	If no argument is given, it trims only spaces from the **right** end.
3	Used for cleanup	Useful when dealing with unwanted trailing symbols, such as in filenames, logs, or formatting.

```
# Removing trailing spaces
txt = "     intensity coding     "
result = txt.rstrip()
print("of all topics,", result, "is my favorite")
# Output: "of all topics,      intensity coding is my favorite"

# Removing specific trailing characters
txt = "intensity coding,,,,,ssqqqww....."
result = txt.rstrip(",.qsw")
print(result)
# Output: "intensity coding"

# Using rstrip() to clean up log data
log = "intensity coding!!!"
cleaned_log = log.rstrip("!")
print(cleaned_log)
# Output: "intensity coding"

# Handling newline characters
txt = "intensity coding\n\n\n"
result = txt.rstrip("\n")
print(result)
# Output: "intensity coding"

# Removing trailing dots from a filename
filename = "intensity_coding.md ... "
cleaned_filename = filename.rstrip(".")
print(cleaned_filename)
# Output: "intensity_coding.md"
```

```
of all topics,      intensity coding is my favorite
intensity coding
intensity coding
```

```
intensity coding
intensity_coding.md
```

37. split()

Point	Description
Use Case	The `split()` method splits a string into a **list of substrings** based on a specified separator. By default, it splits on whitespace.
Syntax	`string.split(separator, maxsplit)`
Parameter	**separator** (Optional): Specifies the separator to use. Default is any whitespace.
	maxsplit (Optional): The maximum number of splits. Default is -1 (all occurrences).

Useful Information

No.	Point	Description
1	Splitting behavior	If no separator is given, `split()` will divide based on **whitespace** (multiple spaces are treated as one).
2	Default vs maxsplit	If `maxsplit` is provided, the string will be split **only that many times**, returning at most `maxsplit + 1` elements.
3	Separator is mandatory for certain cases	When splitting on a specific character, it must be explicitly provided as the separator.

```python
# Default behavior (splitting on whitespace)
txt = "Intensity Coding is powerful"
result = txt.split()
print(result)
# Output: ['Intensity', 'Coding', 'is', 'powerful']

# Splitting using a comma as the separator
txt = "apple, banana, cherry"
result = txt.split(", ")
print(result)
```

```
# Output: ['apple', 'banana', 'cherry']

# Using maxsplit to limit the number of splits
txt = "apple#banana#cherry#orange"
result = txt.split("#", 2)
print(result)
# Output: ['apple', 'banana', 'cherry#orange']

# Handling multiple spaces
txt = "  Intensity    Coding   is  fun  "
result = txt.split()
print(result)
# Output: ['Intensity', 'Coding', 'is', 'fun']
```

```
['Intensity', 'Coding', 'is', 'powerful']
['apple', 'banana', 'cherry']
['apple', 'banana', 'cherry#orange']
['Intensity', 'Coding', 'is', 'fun']
```

38. splitlines()

Point	Description
Use Case	The `splitlines()` method splits a string into a **list of lines**, using different types of line breaks as delimiters.
Syntax	`string.splitlines(keeplinebreaks)`
Parameter	**keeplinebreaks** (Optional): If `True`, keeps line break characters in the result. Default is `False` (removes line breaks).

Useful Information

No.	Point	Description
1	Works with various line breaks	Recognizes `\n`, `\r`, `\r\n`, `\v`, `\f`, `\x1c`, `\x1d`, `\x1e`, `\x85`, `\u2028`, and `\u2029`.
2	Default removes line breaks	If `keeplinebreaks=False`, line breaks are not included in the output.

No.	Point	Description
3	Useful for text processing	Often used when reading multi-line strings, logs, or processing files.

```python
# Basic usage (default behavior)
txt = "Thank you for the music\nWelcome to the jungle"
result = txt.splitlines()
print(result)
# Output: ['Thank you for the music', 'Welcome to the jungle']

# Keeping line breaks
txt = "Thank you for the music\nWelcome to the jungle"
result = txt.splitlines(True)
print(result)
# Output: ['Thank you for the music\n', 'Welcome to the jungle']

# Handling different line breaks
txt = "Line1\rLine2\nLine3\r\nLine4"
result = txt.splitlines()
print(result)
# Output: ['Line1', 'Line2', 'Line3', 'Line4']

# Using splitlines() on a multiline string
txt = """Intensity Coding
is powerful
and useful"""
result = txt.splitlines()
print(result)
# Output: ['Intensity Coding', 'is powerful', 'and useful']
```

```
['Thank you for the music', 'Welcome to the jungle']
['Thank you for the music\n', 'Welcome to the jungle']
['Line1', 'Line2', 'Line3', 'Line4']
['Intensity Coding', 'is powerful', 'and useful']
```

39. startswith()

Point	Description
Use Case	The `startswith()` method checks if a string **begins** with a specified substring and returns `True` or `False`.
Syntax	`string.startswith(value, start, end)`
Parameter	**value** (Required): The substring to check. **start** (Optional): Position to start checking from. **end** (Optional): Position to stop checking.

Useful Information

No.	Point	Description
1	Returns a boolean	`True` if the string starts with the given substring, otherwise `False`.
2	Supports optional range	You can specify `start` and `end` to check within a specific portion of the string.
3	Case-sensitive	`"hello".startswith("H")` returns `False` since case matters.

```
# Basic usage
txt = "Intensity Coding is powerful"
result = txt.startswith("Intensity")
print(result)
# Output: True

# Case-sensitive check
txt = "Intensity Coding"
result = txt.startswith("intensity")
print(result)
# Output: False

# Checking with a specific range
txt = "Hello, welcome to Intensity Coding!"
result = txt.startswith("welcome", 7, 20)
print(result)
# Output: True

# Checking multiple prefixes using tuple
txt = "Machine Learning is fascinating"
```

```python
result = txt.startswith(("Machine", "Deep", "AI"))
print(result)
# Output: True

# Using startswith() to filter strings in a list
sentences = ["AI is evolving", "Machine Learning is key", "Deep Learning is powerful"]
filtered = [s for s in sentences if s.startswith("Machine")]
print(filtered)
# Output: ['Machine Learning is key']
```

```
True
False
True
True
['Machine Learning is key']
```

40. strip()

Point	Description
Use Case	The `strip()` method removes **leading (left)** and **trailing (right)** characters from a string. By default, it removes spaces.
Syntax	`string.strip(characters)`
Parameter	**characters** (Optional): A set of characters to remove from both ends of the string. Default is whitespace.

Useful Information

No.	Point	Description
1	Removes characters from both ends	Unlike `lstrip()` and `rstrip()`, it trims from both the **left** and **right** sides.
2	Default removes spaces	If no argument is given, it trims only **whitespaces** from both ends.

No.	Point	Description
3	Can remove specific characters	Provide a string containing characters you want to remove from both sides.
4	Handles newline characters	It also removes `\n`, `\t`, and other whitespace characters if specified.

```python
# Removing leading and trailing spaces
txt = "     Intensity Coding     "
result = txt.strip()
print("of all topics", result, "is my favorite")
# Output: "of all topics Intensity Coding is my favorite"

# Removing specific characters from both ends
txt = ",,,,,rrttgg.....Intensity Coding....rrr"
result = txt.strip(",.grt")
print(result)
# Output: "Intensity Codin"

# Using strip() to clean up filenames
filename = "###report.pdf###"
cleaned_filename = filename.strip("#")
print(cleaned_filename)
# Output: "report.pdf"

# Removing newline characters
txt = "\n\nIntensity Coding is awesome!\n\n"
result = txt.strip()
print(result)
# Output: "Intensity Coding is awesome!"

# Handling user input cleanup
user_input = "   Hello, World!   "
cleaned_input = user_input.strip()
print(cleaned_input)
# Output: "Hello, World!"
```

```
of all topics Intensity Coding is my favorite
Intensity Codin
report.pdf
```

```
Intensity Coding is awesome!
Hello, World!
```

41. swapcase()

Point	Description
Use Case	The `swapcase()` method **swaps uppercase and lowercase letters** in a string.
Syntax	`string.swapcase()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Swaps case for each character	Uppercase letters become **lowercase**, and lowercase letters become **uppercase**.
2	Works on mixed-case strings	Useful for toggling case in text transformations.
3	No effect on non-alphabetic characters	Numbers, punctuation, and spaces remain unchanged.

```python
# Basic usage of swapcase()
txt = "Hello My Name Is PETER"
result = txt.swapcase()
print(result)
# Output: "hELLO mY nAME iS peter"

# Swapping case in a mixed string
txt = "Intensity Coding 123!"
result = txt.swapcase()
print(result)
# Output: "iNTENSITY cODING 123!"

# Handling special characters and spaces
txt = "Python@123 IS Fun!"
```

```python
result = txt.swapcase()
print(result)
# Output: "pYTHON@123 is fUN!"

# Swapping case in a sentence
txt = "Learning with ChatGPT is Amazing!"
result = txt.swapcase()
print(result)
# Output: "lEARNING WITH cHATgpt IS aMAZING!"
```

```
hELLO mY nAME iS peter
iNTENSITY cODING 123!
pYTHON@123 is fUN!
lEARNING WITH cHATgpt IS aMAZING!
```

42. title()

Point	Description
Use Case	The `title()` method **capitalizes the first letter** of every word in a string, making it look like a title.
Syntax	`string.title()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Capitalizes each word	Converts the **first letter** of every word to **uppercase**.
2	Affects words with numbers/symbols	If a word has a **number or symbol**, the first letter after it is capitalized.
3	Not ideal for proper names	Some words like `"McDonald's"` or `"iPhone"` may not be formatted correctly.

```python
# Basic usage of title()
txt = "welcome to my world"
result = txt.title()
print(result)
# Output: "Welcome To My World"

# Title case with numbers
txt = "welcome to my 2nd world"
result = txt.title()
print(result)
# Output: "Welcome To My 2Nd World"

# Handling words with numbers and symbols
txt = "hello b2b2b2 and 3g3g3g"
result = txt.title()
print(result)
# Output: "Hello B2B2B2 And 3G3G3G"

# Title case with mixed characters
txt = "the quick brown fox's jump"
result = txt.title()
print(result)
# Output: "The Quick Brown Fox'S Jump"
```

```
Welcome To My World
Welcome To My 2Nd World
Hello B2B2B2 And 3G3G3G
The Quick Brown Fox'S Jump
```

43. translate()

Point	Description
Use Case	The `translate()` method replaces characters in a string using a **mapping table** (created with `maketrans()`) or a **dictionary**.
Syntax	`string.translate(table)`

Point	Description
Parameter	**table** (Required): A dictionary or mapping table describing how to replace characters.

Useful Information

No.	Point	Description
1	Uses ASCII codes	When using a **dictionary**, replacements are specified using ASCII values.
2	Works with maketrans()	`maketrans()` helps create a **translation table** for `translate()`.
3	Supports deletion	Characters **can be removed** from the string using an additional argument in `maketrans()`.
4	Raises LookupError	If a character is mapped to itself, `translate()` raises a `LookupError`.

```
# Using a dictionary to replace characters
mydict = {105: 111}  # ASCII: 'i' → 'o'
txt = "intensity coding"
print(txt.translate(mydict))
# Output: "ontensoty codong"

# Using maketrans() to create a mapping table
txt = "intensity coding"
mytable = str.maketrans("ic", "oz")  # 'i' → 'o', 'c' → 'z'
print(txt.translate(mytable))
# Output: "ontensoty zodong"

# Replacing multiple characters
txt = "intensity coding"
x = "int"
y = "xyz"
mytable = str.maketrans(x, y)  # 'i'→'x', 'n'→'y', 't'→'z'
print(txt.translate(mytable))
# Output: "xyzeysxzy codxyg"

# Deleting specific characters while replacing
txt = "intensity coding"
x = "ic"
y = "oz"
```

```python
z = "tyd"  # Characters to delete
mytable = str.maketrans(x, y, z)
print(txt.translate(mytable))
# Output: "onenso zoong"

# Using a dictionary for replacement and deletion
txt = "intensity coding"
mydict = {105: 111,  # 'i' → 'o'
          110: 120,  # 'n' → 'x'
          116: None, # Remove 't'
          115: None} # Remove 's'
print(txt.translate(mydict))
# Output: "oxexoy codoxg"
```

```
ontensoty codong
ontensoty zodong
xyzeysxzy codxyg
onenso zoong
oxexoy codoxg
```

44. upper()

Point	Description
Use Case	The `upper()` method converts all characters in a string to **uppercase**. **Symbols and numbers remain unchanged**.
Syntax	`string.upper()`
Parameter	No parameters.

Useful Information

No.	Point	Description
1	Converts only letters	Numbers and symbols remain **unchanged**.
2	Useful for normalization	Ensures **case consistency** when comparing strings.

No.	Point	Description
3	Works with multilingual text	Supports **Unicode characters** like `à, é, ñ`.

```
# Converting lowercase to uppercase
txt = "intensity coding"
result = txt.upper()
print(result)
# Output: "INTENSITY CODING"

# Uppercase with numbers and symbols
txt = "intensity123! coding?"
result = txt.upper()
print(result)
# Output: "INTENSITY123! CODING?"

# Using upper() in a case-insensitive comparison
user_input = "Intensity Coding"
stored_value = "INTENSITY CODING"

if user_input.upper() == stored_value:
    print("Match found!")
else:
    print("No match.")
# Output: "Match found!"

# Working with multilingual text
txt = "élève naïve café"
result = txt.upper()
print(result)
# Output: "ÉLÈVE NAÏVE CAFÉ"
```

```
INTENSITY CODING
INTENSITY123! CODING?
Match found!
ÉLÈVE NAÏVE CAFÉ
```

45. zfill()

Point	Description
Use Case	The `zfill()` method **pads a string with leading zeros** until it reaches the specified length.
Syntax	`string.zfill(length)`
Parameter	**length** (Required): The total length of the output string (if smaller, no padding is added).

Useful Information

No.	Point	Description
1	Adds leading zeros	Fills the string **from the left** with `"0"` until the specified length is met.
2	Works with numbers in strings	Can be useful for **formatting numbers**, such as IDs or dates.
3	Does not affect signs	If the string starts with `+` or `-`, the zeros are added **after the sign**.

```
# Padding a number stored as a string
txt = "50"
result = txt.zfill(10)
print(result)
# Output: "0000000050"

# Using zfill() with text
txt = "intensity"
result = txt.zfill(15)
print(result)
# Output: "000000intensity"

# zfill() does not trim longer strings
txt = "intensity coding"
result = txt.zfill(10)
print(result)
# Output: "intensity coding" (unchanged, since it's already longer than
10)

# zfill() with negative numbers
txt = "-42"
```

```python
result = txt.zfill(6)
print(result)
# Output: "-00042" (zeros added after the minus sign)

# Formatting a date
def perfect_date(day, month, year):
    day = day.zfill(2)
    month = month.zfill(2)
    year = year.zfill(4)
    date = day + '/' + month + '/' + year
    print("The perfect date (DD/MM/YYYY) is: " + date)

perfect_date("1", "2", "2025")
# Output: "The perfect date (DD/MM/YYYY) is: 02/06/2022"
```

```
0000000050
000000intensity
intensity coding
-00042
The perfect date (DD/MM/YYYY) is: 01/02/2025
```

Found this helpful ?

Follow on LinkedIn
Master AI/ML with Intensity Coding

@bhavdippatel2020

Like | Comment | Share | Save

Each Article Includes Everything You Need

THEORY MADE SIMPLE
Complex ideas explained in an easy way

PYTHON CODE
Hands-on coding for practice

VISUAL LEARNING
Visual diagrams for clarity

MATH BEHIND AI/ML
Step-by-step explanation of core concepts

Explore more tutorials at Intensity Coding

www.intensitycoding.com

Intensity Coding
Something For Everyone