Python String Methods

Strings are one of the most commonly used data types in Python.

Whenever your program works with names, usernames, emails, passwords, messages, search queries, file names, URLs, or user input, you will often need to **clean, search, modify, validate, or split text**.

Python provides many built-in string methods that make these tasks much easier.

By the end of this guide, you will understand how to use the most important string methods and, more importantly, **when to use each one**.

---

1. What Are String Methods?

A string method is a function that belongs to a string object.

The general syntax is:

```
string.method()
```

For example:

```
name = "hafsa"

print(name.upper())
```

Output:

```
HAFSA
```

Here:

- `name` is the string
- `.upper()` is the string method
- `()` executes the method

Some methods also accept arguments:

```
name = "hafsa waseem"

print(name.replace("hafsa", "Hafsa"))
```

Output:

```
Hafsa waseem
```

---

2. Why String Methods Matter

Imagine a user enters:

```
  Hafsa Waseem
```

Your application may need to:

- remove unnecessary spaces
- convert text to lowercase
- check whether it contains a specific word
- replace part of the text
- split the name into separate parts
- validate whether it contains only letters

- count characters
- check whether it starts or ends with something

Instead of manually processing every character, Python gives you ready-made methods.

A useful mental model is:

> **String methods help you transform, inspect, search, and organize text.**

---

3. Strings Are Immutable

Before learning methods, understand one important rule:

String methods do not usually change the original string.

For example:

```
name = "hafsa"

name.upper()

print(name)
```

Output:

```
hafsa
```

Why?

Because strings are immutable.

The method creates and returns a new string.

To keep the result:

```
name = name.upper()

print(name)
```

Output:

```
HAFSA
```

Or:

```
name = "hafsa"

uppercase_name = name.upper()

print(uppercase_name)
```

---

4. Changing Letter Case

upper()

Converts all letters to uppercase.

```
text = "hello python"

print(text.upper())
```

Output:

```
HELLO PYTHON
```

Useful for:

- headings
- labels
- case-insensitive comparisons
- formatting

---

lower()

Converts all letters to lowercase.

```
text = "HELLO PYTHON"

print(text.lower())
```

Output:

```
hello python
```

A very common use is normalizing user input:

```
answer = input("Continue? ")

if answer.lower() == "yes":
    print("Continuing...")
```

Now these inputs can all match:

```
YES
Yes
yes
yEs
```

---

capitalize()

Makes the first character uppercase and the remaining characters lowercase.

```
text = "python PROGRAMMING"

print(text.capitalize())
```

Output:

```
Python programming
```

---

title()

Capitalizes the first letter of each word.

```
name = "hafsa waseem"

print(name.title())
```

Output:

```
Hafsa Waseem
```

Useful for names and headings.

---

swapcase()

Changes uppercase characters to lowercase and lowercase characters to uppercase.

```
text = "Hello Python"

print(text.swapcase())
```

Output:

```
hELLO pYTHON
```

This is less common in everyday applications but useful to understand.

---

5. Removing Unwanted Spaces

User input often contains accidental spaces.

For example:

```
username = "   hafsa   "
```

strip()

Removes whitespace from both ends.

```
username = "   hafsa   "

print(username.strip())
```

Output:

```
hafsa
```

Important:

`strip()` does **not** remove spaces between words.

```
name = "Hafsa Waseem"

print(name.strip())
```

Output:

```
Hafsa Waseem
```

---

lstrip()

Removes whitespace from the left side.

```
text = "   Python"

print(text.lstrip())
```

Output:

```
Python
```

---

rstrip()

Removes whitespace from the right side.

```
text = "Python   "

print(text.rstrip())
```

Output:

```
Python
```

Real Example: Cleaning User Input

```
name = input("Enter your name: ")

name = name.strip()

print("Hello", name)
```

This prevents accidental leading and trailing spaces from becoming part of the stored value.

---

6. Replacing Text

replace()

Replaces one piece of text with another.

Syntax:

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

Example:

```
text = "I like Java"

new_text = text.replace("Java", "Python")

print(new_text)
```

Output:

```
I like Python
```

You can replace individual characters too:

```
phone = "123-456-789"

print(phone.replace("-", ""))
```

Output:

```
123456789
```

Limiting Replacements

You can provide a third argument.

```
text = "apple apple apple"

print(text.replace("apple", "orange", 1))
```

Output:

```
orange apple apple
```

Only the first occurrence was replaced.

---

7. Searching Inside Strings

find()

Returns the index of the first occurrence of a substring.

```
text = "Python programming"

position = text.find("program")

print(position)
```

Output:

```
7
```

If the text is not found:

```
text = "Python"

print(text.find("Java"))
```

Output:

```
-1
```

This makes `find()` useful when you want to safely check whether something exists.

---

index()

`index()` also searches for a substring.

```
text = "Python programming"

print(text.index("program"))
```

Output:

```
7
```

But there is an important difference.

If the substring does not exist:

```
text = "Python"

print(text.index("Java"))
```

Python raises:

```
ValueError
```

So:

find()

Returns:

```
-1
```

when not found.

index()

Raises:

`ValueError`

when not found.

For simple existence checks, this is often enough:

```
if "Python" in text:
    print("Found")
```

---

8. Counting Text

count()

Counts how many times a substring appears.

```
text = "banana"

print(text.count("a"))
```

Output:

```
3
```

Another example:

```
sentence = "Python is easy. Python is powerful."

print(sentence.count("Python"))
```

Output:

```
2
```

---

9. Checking the Beginning and Ending

startswith()

Checks whether a string starts with specific text.

```
url = "https://haas.dev"

print(url.startswith("https"))
```

Output:

```
True
```

Example:

```
filename = "notes.py"

if filename.startswith("notes"):
    print("This is a notes file.")
```

---

endswith()

Checks whether a string ends with specific text.

```
filename = "app.py"

print(filename.endswith(".py"))
```

Output:

```
True
```

This is especially useful for file extensions.

```
file = "photo.jpg"

if file.endswith(".jpg"):
    print("JPEG image")
```

---

10. Checking Whether Text Exists

The `in` operator is not technically a string method, but it is one of the most important tools for working with strings.

```
message = "Welcome to haas.dev"

print("haas.dev" in message)
```

Output:

```
True
```

You can use it in conditions:

```
message = "Learn Python with haas.dev"

if "Python" in message:
    print("Python was mentioned.")
```

---

not in

```
message = "Learn Python"

if "Java" not in message:
    print("Java is not mentioned.")
```

---

11. Splitting a String

split()

Breaks a string into a list.

```
text = "Python is easy"

words = text.split()

print(words)
```

Output:

```
['Python', 'is', 'easy']
```

This is extremely useful when processing sentences.

Splitting Using a Specific Separator

```
data = "Hafsa,Python,Developer"

parts = data.split(",")

print(parts)
```

Output:

```
['Hafsa', 'Python', 'Developer']
```

You can split URLs, CSV-like data, paths, commands, and many other formats.

Limiting Splits

```
text = "Python is very easy"

print(text.split(" ", 2))
```

Output:

```
['Python', 'is', 'very easy']
```

The `2` tells Python to perform at most two splits.

---

12. Joining Strings

`join()`

`join()` does the opposite of `split()`.

Suppose you have:

```
words = ["Python", "is", "powerful"]
```

You can combine them:

```
sentence = " ".join(words)

print(sentence)
```

Output:

```
Python is powerful
```

Using another separator:

```
skills = ["Python", "JavaScript", "React"]

result = ", ".join(skills)

print(result)
```

Output:

```
Python, JavaScript, React
```

Understanding `join()`

The separator calls `join()`:

`" ".join(words)`

Not:

`words.join(" ")`

Think:

> “Put this separator between every item.”

---

13. Character Validation Methods

Python provides several methods for checking what a string contains.

These methods return either:

`True`

or:

`False`

---

isalpha()

Checks whether all characters are alphabetic.

```
name = "Hafsa"
```

```
print(name.isalpha())
```

Output:

```
True
```

But:

```
name = "Hafsa123"

print(name.isalpha())
```

Output:

```
False
```

Spaces also cause it to return `False`:

```
print("Hafsa Waseem".isalpha())
```

Output:

```
False
```

---

isdigit()

Checks whether all characters are digits.

```
age = "25"

print(age.isdigit())
```

Output:

```
True
```

But:

```
print("25years".isdigit())
```

Output:

```
False
```

---

isalnum()

Checks whether all characters are letters or numbers.

```
username = "hafsa123"

print(username.isalnum())
```

Output:

```
True
```

But:

```
username = "hafsa_123"

print(username.isalnum())
```

Output:

```
False
```

because _ is not a letter or number.

---

isspace()

Checks whether the string contains only whitespace.

```
text = "   "

print(text.isspace())
```

Output:

```
True
```

---

islower()

Checks whether all cased characters are lowercase.

```
text = "python"

print(text.islower())
```

Output:

```
True
```

---

isupper()

Checks whether all cased characters are uppercase.

```
text = "PYTHON"

print(text.isupper())
```

Output:

```
True
```

---

istitle()

Checks whether the string follows title-case formatting.

```
text = "Python Programming"

print(text.istitle())
```

Output:

```
True
```

---

14. Checking Identifiers

isidentifier()

Checks whether a string can be used as a valid Python identifier.

```
name = "user_name"
```

```
print(name.isidentifier())
```

Output:

```
True
```

But:

```
name = "123user"

print(name.isidentifier())
```

Output:

```
False
```

This can be useful when building tools that validate variable names or code-related input.

---

15. Case Insensitive Comparisons

Suppose:

```
username = "HAFSA"
```

and you want to compare it with:

```
hafsa
```

A direct comparison fails:

```
print(username == "hafsa")
```

Output:

```
False
```

Normalize the case:

```
print(username.lower() == "hafsa")
```

Output:

```
True
```

This is a common real-world pattern:

```
command = input("Enter command: ").strip().lower()

if command == "start":
    print("Starting...")
```

Now these can all work:

```
start
START
Start
 sTaRt
```

---

16. `casefold()` for Stronger Case Normalization

Python also provides:

```
casefold()
```

Example:

```
text = "HELLO"

print(text.casefold())
```

Output:

```
hello
```

For ordinary English comparisons, `lower()` is usually sufficient.

`casefold()` is designed for more aggressive Unicode-aware case-insensitive comparisons.

Example:

```
username = input("Username: ")

if username.casefold() == "hafsa":
    print("Welcome")
```

---

17. `partition()`

`partition()` divides a string into three parts:

```
before separator
separator
after separator
```

Example:

```
email = "hafsa@example.com"

result = email.partition("@")

print(result)
```

Output:

```
('hafsa', '@', 'example.com')
```

This is useful when you specifically want to divide text around the first occurrence of a separator.

---

18. rpartition()

`rpartition()` searches from the right side.

```
path = "projects/python/app.py"

print(path.rpartition("/"))
```

Output:

```
('projects/python', '/', 'app.py')
```

This can be useful when working with paths or structured text.

---

19. Removing Specific Characters With strip()

`strip()` can also accept characters.

```
text = "###Python###"

print(text.strip("#"))
```

Output:

```
Python
```

Important:

`strip()` removes matching characters from the **ends**, not from the middle.

```
text = "##Py#thon##"

print(text.strip("#"))
```

Output:

```
Py#thon
```

The `#` in the middle remains.

---

20. Formatting Text With `zfill()`

`zfill()`

Adds zeros to the beginning of a numeric-looking string.

```
number = "42"

print(number.zfill(5))
```

Output:

```
00042
```

Useful for:

- IDs
- invoice numbers
- sequence numbers
- timestamps

Example:

```
order_id = "27"

print("Order #" + order_id.zfill(5))
```

Output:

```
Order #00027
```

---

21. Translating Characters

Python provides:

```
maketrans()
translate()
```

These are useful when you need to replace many individual characters at once.

Example:

```
text = "hello"

table = str.maketrans({
    "h": "H",
    "e": "3",
    "o": "0"
})

print(text.translate(table))
```

Output:

```
H3ll0
```

This is more advanced and less common for beginner projects, but it becomes useful in text processing.

---

22. A Practical String Cleaning Pipeline

Real applications rarely use just one method.

You often combine several methods.

Example:

```
username = "  HAFSA_WASEEM  "

username = username.strip().lower()

print(username)
```

Output:

```
hafsa_waseem
```

This is called **normalization**.

A common workflow is:

```
Raw input
   ↓
strip()
   ↓
lower() / casefold()
   ↓
validate
   ↓
search / split / replace
   ↓
store or use
```

---

23. Real Example: Username Validation

Suppose a website allows usernames containing only letters, numbers, and underscores.

```
username = input("Enter username: ").strip()

if username == "":
    print("Username cannot be empty.")
elif not username.replace("_", "").isalnum():
    print("Only letters, numbers, and underscores are allowed.")
else:
    print("Username is valid.")
```

Notice how several string operations can work together.

---

24. Real Example: Email Validation Basics

String methods can help perform basic checks.

```
email = input("Enter your email: ").strip().lower()

if "@" not in email:
    print("Invalid email.")
elif not email.endswith(".com"):
    print("Email must end with .com")
else:
    print("Email looks valid.")
```

This is only a **basic check**, not complete email validation.

Real applications should use stronger validation rules.

---

25. Real Example: File Extension

Suppose a user uploads a file.

```
filename = "profile_photo.png"

if filename.endswith(".png"):
    print("PNG image")
elif filename.endswith(".jpg"):
    print("JPG image")
else:
    print("Unknown format")
```

You can also normalize the filename first:

```
filename = filename.strip().lower()
```

---

26. Real Example: Search Feature

Imagine haas.dev has a resource search feature.

A user enters:

```
 PYTHON
```

You want to find Python resources regardless of capitalization or accidental spaces.

```
search = input("Search: ").strip().lower()

title = "Python String Methods"

if search in title.lower():
    print("Resource found!")
```

The important idea is:

> Normalize both sides before comparing.

---

27. Real Example: Counting Words

```
sentence = input("Enter a sentence: ").strip()

words = sentence.split()

print("Word count:", len(words))
```

Input:

```
Python is easy to learn
```

Output:

```
Word count: 5
```

Here:

`split()`

turns the sentence into a list, and:

`len()`

counts the list items.

---

28. Real Example: Extracting an Email Domain

```
email = "hafsa@example.com"

parts = email.split("@")

username = parts[0]
domain = parts[1]
```

```
print(username)
print(domain)
```

Output:

```
hafsa
example.com
```

A safer approach when you specifically expect one separator is:

```
username, separator, domain = email.partition("@")

print(username)
print(domain)
```

---

29. Real Example: Remove Unwanted Characters

Suppose a phone number is stored as:

```
phone = "0300-123-4567"
```

You can remove hyphens:

```
phone = phone.replace("-", "")

print(phone)
```

Output:

```
03001234567
```

You can extend the same idea:

```
phone = phone.replace("-", "").replace(" ", "")
```

---

30. Method Chaining

You can call multiple methods on the same expression.

```
text = "   HELLO PYTHON   "

result = text.strip().lower()

print(result)
```

Output:

```
hello python
```

You can chain more:

```
text = "   Python Programming   "

result = text.strip().lower().replace(" ", "-")

print(result)
```

Output:

```
python-programming
```

Read chained methods from left to right:

`strip`

`lower`

`replace`

`final result`

Do not create unnecessarily complicated chains when separate steps would be easier to understand.

---

31. Common String Methods at a Glance

Method	Purpose
`upper()`	Convert to uppercase
`lower()`	Convert to lowercase
`capitalize()`	Capitalize first character
`title()`	Capitalize each word
`swapcase()`	Swap letter case
`strip()`	Remove whitespace from both ends

lstrip()	Remove left-side whitespace
rstrip()	Remove right-side whitespace
replace()	Replace text
find()	Find position safely
index()	Find position or raise error
count()	Count occurrences
startswith()	Check beginning
endswith()	Check ending
split()	Convert string into list
join()	Combine list items into string
isalpha()	Check alphabetic characters
isdigit()	Check digits
isalnum()	Check letters/numbers

`isspace()`	Check whitespace
`islower()`	Check lowercase
`isupper()`	Check uppercase
`istitle()`	Check title case
`isidentifier()`	Check valid Python identifier
`partition()`	Split around first separator
`rpartition()`	Split around last separator
`zfill()`	Add zeros to the left
`casefold()`	Normalize case for comparisons

---

32. `find()` vs `index()`

Situation	**`find()`**	**`index()`**

Found	Returns position	Returns position
Not found	Returns `-1`	Raises `ValueError`
Good for safe searching	Yes	Sometimes
Requires error handling when missing	No	Yes

Example:

```
text = "Python"

position = text.find("Java")

if position == -1:
    print("Not found")
```

---

33. `split()` vs `join()`

These methods often work as opposites.

```
text = "Python is powerful"

words = text.split()

print(words)
```

Result:

```
['Python', 'is', 'powerful']
```

Then:

```
sentence = " ".join(words)

print(sentence)
```

Result:

```
Python is powerful
```

Mental model:

```
String
  split()
List
  join()
String
```

---

34. Common Mistakes

Mistake 1: Forgetting that strings are immutable

Wrong assumption:

```
name = "hafsa"

name.upper()
```

```
print(name)
```

Output:

```
hafsa
```

Correct:

```
name = name.upper()
```

Mistake 2: Forgetting parentheses

Wrong:

```
name.upper
```

Correct:

```
name.upper()
```

`name.upper` refers to the method.

`name.upper()` calls it.

Mistake 3: Expecting `strip()` to remove middle spaces

```
text = "Python   Programming"

print(text.strip())
```

The spaces between the words remain.

Mistake 4: Using `index()` when the text might not exist

```
text.index("Java")
```

can raise an error.

For simple searching:

```
text.find("Java")
```

or:

```
"Java" in text
```

may be more appropriate.

Mistake 5: Forgetting case sensitivity

This:

```
"Python" == "python"
```

returns:

```
False
```

Normalize when appropriate:

```
"Python".lower() == "python".lower()
```

Mistake 6: Expecting `split()` to return a string

```
result = "Python is easy".split()

print(type(result))
```

Output:

```
<class 'list'>
```

`split()` returns a list.

---

35. Choosing the Right Method

When working with text, ask what you actually need.

Need to change case?

Use:

`upper()`
`lower()`
`capitalize()`
`title()`

Need to clean spaces?

Use:

`strip()`
`lstrip()`
`rstrip()`

Need to replace text?

Use:

`replace()`

Need to search?

Use:

`in`
`find()`
`index()`

Need to count?

Use:

`count()`

Need to check beginning or ending?

Use:

`startswith()`
`endswith()`

Need to break text apart?

Use:

`split()`
`partition()`

Need to combine text?

Use:

`join()`

Need to validate characters?

Use:

`isalpha()`
`isdigit()`
`isalnum()`
`isspace()`

This is more useful than memorizing methods randomly.

---

36. A General Text Processing Framework

When processing user input, think in this order:

1. Receive

2. Clean

3. Normalize

4. Validate

5. Transform
    ↓
6. Search / Extract
    ↓
7. Store or Display

Example:

```
email = input("Enter email: ")

# 1. Clean
email = email.strip()

# 2. Normalize
email = email.lower()

# 3. Validate
if "@" not in email:
    print("Invalid email")
else:
    # 4. Extract
    username, separator, domain = email.partition("@")

    print("Username:", username)
    print("Domain:", domain)
```

This way of thinking becomes extremely useful when you start building real applications.

---

37. Mini Project: Text Analyzer

Let's combine several string methods into one small project.

The program will:

- clean the sentence
- count words
- count characters
- search for Python
- count Python occurrences
- display the sentence in uppercase
- display the sentence in lowercase

```
text = input("Enter a sentence: ")

text = text.strip()

if text == "":
    print("Please enter some text.")
else:
    words = text.split()

    print("\n--- Text Analysis ---")
    print("Original:", text)
    print("Uppercase:", text.upper())
    print("Lowercase:", text.lower())
    print("Characters:", len(text))
    print("Words:", len(words))
    print("Python occurrences:", text.lower().count("python"))

    if "python" in text.lower():
        print("The text contains Python.")
    else:
        print("The text does not contain Python.")
```

Example input:

```
Python is easy. I am learning Python.
```

Possible output:

```
--- Text Analysis ---
Original: Python is easy. I am learning Python.
Uppercase: PYTHON IS EASY. I AM LEARNING PYTHON.
Lowercase: python is easy. i am learning python.
Characters: 38
Words: 7
Python occurrences: 2
The text contains Python.
```

This small project demonstrates how individual methods become useful when combined into a complete workflow.

---

38. Practice Exercises

Exercise 1: Name Formatter

Ask the user for their name and:

- remove unnecessary spaces
- convert it to title case
- print the formatted name

Exercise 2: Word Counter

Ask for a sentence and display:

- total characters

- total words

Exercise 3: Search

Ask the user for:

```
sentence
search word
```

Check whether the word exists in the sentence.

Exercise 4: Email Extractor

Ask for an email and extract:

```
username
domain
```

Exercise 5: File Checker

Ask for a filename and determine whether it is:

```
.py
.jpg
.png
.pdf
```

Exercise 6: Username Validator

Accept usernames containing only:

- letters
- numbers
- _

Reject everything else.

Exercise 7: Sentence Cleaner

Given:

```
"   PYTHON is AMAZING   "
```

produce:

```
python is amazing
```

using string methods.

Exercise 8: Character Counter

Ask for a word and a character.

Example:

```
Word: programming
```

```
Character: g
```

Output:

```
g appears 2 times.
```

---

39. 5 Minute Challenge

Build a program that asks:

```
Enter your full name:
Enter your favorite programming language:
```

Then produce a message like:

```
Hello Hafsa Waseem!

Your favorite language is Python.

Name length: 12
Language length: 6

Python is a great language for building things.
```

Requirements:

- clean input with `strip()`
- format the name with `title()`
- normalize the language using `lower()`
- use `len()`
- use `in` for a simple check

- use at least three different string methods

---

40. Mini Quiz

Question 1

What does this return?

```
"Python".lower()
```

A. PYTHON
B. Python
C. python
D. Error

Answer: C

Question 2

What does `split()` return?

A. String
B. Integer
C. List
D. Boolean

Answer: C

Question 3

What happens here?

```
text = "Python"
print(text.find("Java"))
```

A. 0
B. -1
C. None
D. ValueError

Answer: B

---

41. Interview Questions

Beginner

1. What is a string method in Python?
2. What is the difference between `upper()` and `lower()`?
3. What does `strip()` do?
4. What does `replace()` do?
5. What does `split()` return?
6. What does `join()` do?
7. How can you check whether a string contains a word?
8. What does `startswith()` do?
9. What does `endswith()` do?
10. What is the difference between `find()` and `index()`?

Intermediate

11. Why don't string methods modify the original string?
12. How would you normalize user input before comparing it?
13. How can `split()` and `join()` be used together?
14. How would you count occurrences of a word?
15. How would you validate that a string contains only digits?
16. How would you remove leading and trailing whitespace?
17. What is the difference between `strip()` and `replace()`?
18. When would `casefold()` be preferable to `lower()`?
19. How would you extract the domain from an email?
20. How would you build a basic text normalization pipeline?

---

42. String Methods Cheat Sheet

```
text.upper()
text.lower()
text.capitalize()
text.title()
text.swapcase()

text.strip()
text.lstrip()
text.rstrip()

text.replace("old", "new")

text.find("word")
text.index("word")
text.count("word")

text.startswith("https")
text.endswith(".py")
```

```
"Python" in text
"Python" not in text

text.split()
text.split(",")

" ".join(words)

text.isalpha()
text.isdigit()
text.isalnum()
text.isspace()
text.islower()
text.isupper()
text.istitle()
text.isidentifier()

text.partition("@")
text.rpartition("/")

text.zfill(5)

text.casefold()
```

---

43. Key Takeaways

- String methods are built-in tools for working with text.
- Strings are immutable, so methods generally return new strings.
- Use `strip()` to clean surrounding whitespace.
- Use `lower()` or `casefold()` when normalizing text for comparison.
- Use `replace()` to substitute text.

- Use `find()`, `index()`, and `in` for searching.
- Use `count()` to count occurrences.
- Use `startswith()` and `endswith()` for prefix and suffix checks.
- Use `split()` to turn text into a list.
- Use `join()` to combine list items into a string.
- Use `isalpha()`, `isdigit()`, and `isalnum()` for validation.
- Multiple methods can be combined into a practical text-processing pipeline.
- The goal is not to memorize every method. Learn **what problem each method solves**.

Final Mental Model

When you receive text in a Python program, think:

```
INPUT
  
CLEAN
strip()
  
NORMALIZE
lower() / casefold()
  
VALIDATE
isalpha() / isdigit() / isalnum()
  
TRANSFORM
replace() / capitalize() / title()
  
SEARCH
in / find() / count()
  
SPLIT OR EXTRACT
```

split() / partition()

USE THE DATA

Once you understand this workflow, string methods stop being a list of functions to memorize and become a practical toolkit for processing real application data.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app/resources