

<https://www.shortcutfoo.com/app/dojos/python-strings/cheatsheet>

Cases

```
In [9]: s = 'hello world'
        s.capitalize()
```

Out[9]: 'Hello world'

```
In [11]: s.lower()
```

Out[11]: 'hello world'

```
In [13]: s.swapcase()
```

Out[13]: 'HELLO WORLD'

```
In [15]: s.title()
```

Out[15]: 'Hello World'

```
In [17]: s.upper()
```

Out[17]: 'HELLO WORLD'

Sequence Operations - I

```
In [35]: s = [1, 2, 3, 4, 5]
        s2 = [2, 3, 4]
        print(2 in s)
```

True

```
In [50]: s = [1, 2, 3, 4, 5]
        s2 = [2, 3, 4]
        s2 in s
```

Out[50]: False

- `s2 in s` checks whether the entire list `s2` (`[2, 3, 4]`) is an element inside the list `s` (`[1, 2, 3, 4, 5]`).
- But `s` contains integers, not lists. So Python will look for a list `[2, 3, 4]` inside `s` and won't find it. Thus we are getting answer as false

```
In [37]: s + s2
```

Out[37]: `[1, 2, 3, 4, 5, 2, 3, 4]`

```
In [39]: len(s)
```

Out[39]: 5

In [41]: `min(s)`

Out[41]: 1

In [43]: `max(s)`

Out[43]: 5

Sequence Operations II

In [48]: `s2 not in s`

Out[48]: True

In [53]: `s * 3`

Out[53]: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

In [55]: `s[2]`

Out[55]: 3

In [59]: `s[0:4:2]`

Out[59]: [1, 3]

In [62]: `s.count(s2)`

Out[62]: 0

Whitespace I

`str.center(width)`

- In Python, the `str.center(width)` method centers a string in a field of a given width by padding it with spaces (by default) on both sides.
- `width`: The total length of the resulting string (including the original string and padding).
- If `width` is less than or equal to the length of the string, the original string is returned unchanged.

```
In [67]: s = "Hello"
centered = s.center(11)
print(f'{centered}')
```

```
' Hello '
```

```
In [69]: s.isspace()
```

```
Out[69]: False
```

```
In [71]: s = "Hello"
centered = s.center(11)
print(centered)
```

```
Hello
```

str.ljust(width)

- In Python, the `str.ljust(width)` method left-justifies a string in a field of the given width by padding it with spaces (or a specified character) on the right.

```
In [73]: s = "Hello"
left_justified = s.ljust(10, '*')
print(f'{left_justified}')
```

```
'Hello*****'
```

str.rjust(width[, fillchar])

- The `rjust()` method right-justifies a string in a field of the specified width by padding it on the left. You can optionally specify a character for padding (default is a space).

```
In [77]: s = "Hello"
right_justified = s.rjust(10, '-')
print(f'{right_justified}')
```

```
'-----Hello'
```

str.strip()

- The `str.strip()` method in Python is used to remove leading and trailing whitespace (or other specified characters) from a string.

```
In [1]: s = " Hello World "
stripped = s.strip()
print(f'{stripped}')
```

```
'Hello World'
```

```
In [3]: s = "xyHello Worldyx"
stripped = s.strip("xy")
print(f'{stripped}')
```

```
'Hello World'
```

str.lstrip()

- Returns a copy of the string with leading whitespace (or specified characters) removed.
- It only affects the left side (beginning) of the string.

In [149...

```
s = "  Hello World  "  
print(s.lstrip())
```

Hello World

- Remove leading spaces
- Only the spaces before "Hello" are removed.

In [152...

```
s = "///home/user"  
print(s.lstrip("/"))
```

home/user

It strips all leading / characters from the start.

In [156...

```
s = "123abc123"  
print(s.lstrip("123"))
```

abc123

- Mixed leading characters
- Removes all leading characters that match any in "123", in any order, until a non-matching character is found.

In [159...

```
s = "Python"  
print(s.lstrip())
```

Python

- No leading characters to remove
- No change since there were no leading whitespaces.

str.rstrip()

- Returns a copy of the string with trailing whitespace (or specified characters) removed.
- It only affects the right side (end) of the string.

In [164...

```
s = "  Hello World  "  
print(s.rstrip())
```

Hello World

- Remove trailing spaces

- Only the spaces after "World" are removed.

```
In [170... s = "file.txt...."  
print(s.rstrip("."))
```

file.txt

- Remove specific trailing characters
- All trailing dots are stripped from the right side.

```
In [174... s = "abc123321"  
print(s.rstrip("123"))
```

abc

- Mixed trailing characters
- Removes characters matching any in "123" from the end, until it hits a non-matching character.

```
In [178... s = "Python"  
print(s.rstrip())
```

Python

- No trailing characters to remove
- No change since there were no trailing whitespaces.

str.index(sub, start, end)

- Purpose: Finds the first occurrence of a substring (sub) within the string str between positions start and end.
- Returns the index of the first match.
- Raises ValueError if the substring is not found.

```
In [6]: s = "Hello world, welcome to Python!"  
index = s.index("o", 5, 20)  
print(index)
```

7

Find / Replace I

str.(find)

- Returns the lowest index where the substring sub is found in the string.
- Returns -1 if the substring is not found.

- Optional start and end arguments define the slice of the string to search

```
In [13]: s = "Hello world"
index = s.find("world")
print(index)
```

6

"world" starts at index 6 in "Hello world".

```
In [10]: s = "Hello world"
index = s.find("Python")
print(index)
```

-1

"Python" is not in "Hello world", so .find() returns -1.

s.index(s2)

- Return lowest index of s2 in s (but raise ValueError if not found)

```
In [18]: s = "abracadabra"
s2 = "cad"

# Find the first occurrence of s2 in s
pos = s.index(s2)
print(pos)
```

4

```
In [20]: s = "hello world"
s2 = "Python"

print(s.index(s2))
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[20], line 4
      1 s = "hello world"
      2 s2 = "Python"
----> 4 print(s.index(s2))

ValueError: substring not found
```

s.rindex(s2)

- str.rindex(sub[, start[, end]])
- Returns the highest index (rightmost) where the substring sub is found in the string.
- Raises a ValueError if the substring is not found.
- Works just like rfind(), but throws an error instead of returning -1.

```
In [57]: s = "apple banana apple"
s2 = "apple"

index = s.rindex(s2)
print(index)
```

13

"apple" occurs at index 0 and again at index 13.

rindex() returns the rightmost (last) occurrence → 13.

```
In [62]: s = "hello world"
s2 = "python"

print(s.rindex(s2)) #If substring is not found
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[62], line 4
      1 s = "hello world"
      2 s2 = "python"
----> 4 print(s.rindex(s2))

ValueError: substring not found
```

✔ Comparison with `rfind()`:

Method	Returns on match	If not found
<code>rfind()</code>	Rightmost index	-1
<code>rindex()</code>	Rightmost index	<code>ValueError</code> raised

str.rfind()

- Returns the highest index where the substring sub is found in the string.
- Returns -1 if the substring is not found.
- Optional start and end arguments can limit the search range.

```
In [44]: s = "bananas are bananas"
s2 = "bananas"

index = s.rfind(s2)
print(index)
```

12

- "bananas" appears twice: at index 0 and again at index 12.
- rfind() returns the last (rightmost) occurrence.

```
In [49]: s = "hello world"
s2 = "python"

index = s.rfind(s2)
print(index) # Substring not found
```

-1

```
In [51]: s = "abc abc abc"
index = s.rfind("abc", 0, 7)
print(index)
```

4

Only the part "abc abc" (index 0 to 6) is searched. The last "abc" in that range starts at index 4.

Method	Direction	Returns	Not Found
<code>find()</code>	Left to Right	First match index	-1
<code>rfind()</code>	Right to Left	Last match index	-1

str.replace(old, new)

- old: The substring you want to replace.
- new: The substring to replace it with.
- Returns a new string with all occurrences of old replaced by new.

```
In [33]: s = "I love apples"
s2 = "apples"
s3 = "oranges"

result = s.replace(s2, s3)
print(result)
```

I love oranges

s.replace(s2, s3, count)

- To Replace only a certain number of times - You can also use an optional third argument to limit the number of replacements:

```
In [39]: s = "apple apple apple"
print(s.replace("apple", "orange", 1)) #2 prints orange twice
```

orange apple apple

Cases

str.casefold()

- Converts the string to lowercase in a way that is more aggressive and Unicode-aware than str.lower().
- Useful for case-insensitive comparisons, especially for international text.

```
In [67]: s = "HELLO World"
         print(s.casefold())
```

hello world

```
In [69]: s1 = "straße"      # German for "street"
         s2 = "STRASSE"     # Equivalent in uppercase

         print(s1.casefold() == s2.casefold()) # True
```

True

- s1.lower() would return "straße" (unchanged),
- but s1.casefold() returns "strasse", which matches s2.casefold().

Method	Best for	Unicode-aware?
<code>lower()</code>	Simple lowercase	✗
<code>casefold()</code>	Comparisons, Unicode	✓

str.islower()

- Returns True if all the alphabetic characters in the string are lowercase.
- Returns False if there are any uppercase letters.
- Non-alphabetic characters (like numbers, spaces, punctuation) are ignored in the check.

```
In [76]: s = "hello world"
         print(s.islower())
```

True

```
In [78]: s = "Hello world"
         print(s.islower())
```

False

```
In [82]: s = "hello123!"  
print(s.islower()) #Numbers and symbols ignored
```


True

```
In [84]: s = "1234!"  
print(s.islower()) #No alphabetic characters
```


False

str.isupper()

— Checks if all alphabetic characters are uppercase

```
In [87]: s1 = "HELLO WORLD"  
print(s1.isupper()) #  True
```

True

```
In [89]: s2 = "Hello World"  
print(s2.isupper()) #  False
```

False

```
In [91]: s3 = "HELLO123!"  
print(s3.isupper()) #  True – numbers and punctuation are ignored
```


True

```
In [93]: s4 = "1234!"  
print(s4.isupper()) #  False – no letters present
```


False

str.istitle()


- Checks if the string is in title case
- Each word should start with an uppercase letter followed by lowercase letters.

```
In [96]: s1 = "Hello World"  
print(s1.istitle()) #  True
```

True

```
In [98]: s2 = "Hello world"  
print(s2.istitle()) #  False
```

False

```
In [100]: s3 = "This Is A Title"  
print(s3.istitle()) #  True
```

True

```
In [102]: s4 = "123 Hello!"  
print(s4.istitle()) #  True – non-letter parts don't affect the result
```

True

```
In [104... s5 = "HELLO WORLD"
print(s5.istitle()) # ✗ False – all caps is not title case
```

False

Method	What It Checks	Example	Result
<code>islower()</code>	All letters are lowercase	<code>"hello"</code>	✓ True
<code>isupper()</code>	All letters are uppercase	<code>"HELLO"</code>	✓ True
<code>istitle()</code>	Title case: Each word starts with a capital	<code>"Hello World"</code>	✓ True

Inspection

`str.endswith(suffix[, start[, end]])`

- Checks if the string ends with the specified suffix.
- Returns True if it does, False otherwise.
- Optional start and end let you check within a slice of the string.

```
In [2]: s = "hello world"
s2 = "world"

print(s.endswith(s2))
```

True

Basic usage

```
In [4]: s = "hello world"
print(s.endswith("World")) # Note the capital "W"
```

False

Case-sensitive

```
In [6]: s = "hello world"
# Check if the part from index 0 to 5 ("hello") ends with "lo"
print(s.endswith("lo", 0, 5))
```

True

With start and end

```
In [8]: filename = "report.pdf"
print(filename.endswith((".pdf", ".docx")))
```

True

You can check for multiple suffixes using a tuple:

str.startswith(prefix[, start[, end]])

- Checks if a string starts with the specified prefix.
- Returns True if it does, False otherwise.
- You can also specify start and end indexes to check a portion of the string.

```
In [16]: s = "hello world"
print(s.startswith("hello"))
```

True

Basic usage

```
In [18]: s = "hello world"
print(s.startswith("Hello")) # Capital H
```

False

Case-sensitive

```
In [22]: s = "hello world"
# Check if the part starting at index 6 ("world") starts with "wor"
print(s.startswith("wor", 6))
```

True

With start and end

```
In [25]: url = "https://openai.com"
print(url.startswith(("http://", "https://")))
```

True

Tuple of prefixes

Method	Checks...	Returns
<code>startswith()</code>	Beginning of text	<code>True/False</code>
<code>endswith()</code>	End of text	<code>True/False</code>

str.isalnum()

- Returns True if all characters in the string are alphanumeric (letters or numbers).
- Returns False if the string contains spaces, punctuation, or symbols.

- The string must also be non-empty.

```
In [31]: s = "Python3"  
print(s.isalnum())
```

True

Alphanumeric string

```
In [34]: s = "Python 3"  
print(s.isalnum())
```

False

Contains a space

```
In [37]: s = "Python_3"  
print(s.isalnum())
```

False

Contains punctuation

```
In [40]: s = ""  
print(s.isalnum())
```

False

Empty string

```
In [43]: print("12345".isalnum()) # True  
print("OpenAI".isalnum()) # True
```

True

True

Only numbers or only letters

str.isalpha()

- Returns True if all characters in the string are alphabetic letters (A–Z, a–z).
- Returns False if there are spaces, digits, punctuation, or if the string is empty.

```
In [49]: s = "Python"  
print(s.isalpha())
```

True

Only letters

```
In [55]: s = "Hello World"  
print(s.isalpha())
```

False

Contains space

```
In [57]: s = "Python3"  
print(s.isalpha())
```

False

Contains numbers

```
In [60]: s = ""  
print(s.isalpha())
```

False

Empty string

```
In [64]: s = "Café"  
print(s.isalpha())
```

True

Unicode letters (also count) - (Unicode letters like "é" are valid alphabetic characters.)

str.isnumeric()

- Returns True if all characters in the string are numeric.

Includes:

1. Digits (0–9)
2. Unicode numeric characters (like superscripts, fractions, Roman numerals, etc.)
3. Returns False if the string contains letters, spaces, or symbols.
4. An empty string also returns False.

```
In [69]: s = "12345"  
print(s.isnumeric())
```

True

Regular digits

```
In [72]: s = "123abc"  
print(s.isnumeric())
```

False

Contains letters

```
In [75]: s = "123 456"  
print(s.isnumeric())
```

False

Contains spaces

```
In [78]: s = "²³" # superscript 2 and 3  
print(s.isnumeric())
```

True

Unicode numeric characters

```
In [81]: s = ""  
print(s.isnumeric())
```

False

Empty string

str.isdecimal()

- Returns True if all characters in the string are decimal digits (0–9).
- Returns False if the string contains letters, spaces, symbols, or even non-decimal numeric characters (like superscripts or fractions).
- Does not accept Unicode numeric characters like ², ³/₄, or Roman numerals.
- An empty string also returns False.

```
In [85]: s = "123456"  
print(s.isdecimal())
```

True

All decimal digits

```
In [88]: s = "123abc"  
print(s.isdecimal())
```

False

Contains letters

```
In [91]: s = "123 456"  
print(s.isdecimal())
```

False

Contains whitespace

```
In [94]: s = "²" # Superscript 2  
print(s.isdecimal())
```

False

Contains Unicode numeric (e.g., superscript)

```
In [98]: s = "9"  
print(s.isdecimal())
```

True

Single decimal digit

str.isdigit()

- Returns True if all characters in the string are digits.

Accepts:

1. Standard digits: 0–9
2. Unicode digit characters (e.g., superscripts like ², ³)
3. Returns False for:
4. Letters
5. Spaces
6. Decimal points
7. Fractions
8. Empty strings

```
In [101... s = "2025"  
print(s.isdigit())
```

True

Regular digits

```
In [104... s = "²³"  
print(s.isdigit())
```

True

Unicode digits (like superscripts)

```
In [107... s = "123abc"  
print(s.isdigit())
```

False

Contains letters

```
In [110... s = "123 456"  
print(s.isdigit())
```

False

Contains a space

In [113...

```
s = ""
print(s.isdigit())
```

False

Empty string

Quick Comparison:

Character/String	.isdigit()	.isdecimal()	.isnumeric()
"123"	✓	✓	✓
" 2 3 "	✓	✗	✓
"½" (fraction)	✗	✗	✓
"XII" (Roman)	✗	✗	✓

Splitting

str.join()

separator.join(iterable)

- separator is the string that will appear between the elements.
- iterable is typically a list, tuple, or string.
- Each element in the iterable must be a string.

In [119...

```
s = "-"
result = s.join('123')
print(result)
```

1-2-3

'123' is a string, which is also an iterable of individual characters: '1', '2', '3'

s is '-', so it inserts '-' between each character

In [122...

```
s = " "
words = ["Python", "is", "fun"]
print(s.join(words))
```

Python is fun

Join a list of words

In [126...

```
s = ","  
digits = ["1", "2", "3"]  
print(s.join(digits))
```

1,2,3

Join with a comma

str.partition(sep)

- Splits the string into three parts:
 1. The part before the separator
 2. The separator itself
 3. The part after the separator
- Always returns a tuple of 3 strings
- If the separator is not found, it returns:

```
( "", "" )
```

In [130...

```
s = "hello:world"  
result = s.partition(":")  
print(result)
```

```
('hello', ':', 'world')
```

Separator found

In [133...

```
s = "helloworld"  
result = s.partition(":")  
print(result)
```

```
('helloworld', '', '')
```

Separator not found

In [137...

```
s = "2025-05-27"  
result = s.partition("-")  
print(result)
```

```
('2025', '-', '05-27')
```

Partition at first match only

(Only the first - is used to split.)

In [140...

```
s = "username=kirti"  
key, sep, value = s.partition("=")  
print(key) # 'username'  
print(value) # 'kirti'
```

username
kirti

Splitting key-value pairs

str.rpartition(sep)

- Works like `.partition(sep)` but splits from the rightmost occurrence of the separator.
- Always returns a 3-element tuple:
 1. Part before the last occurrence of sep
 2. The separator itself
 3. Part after the separator

If the separator is not found, it returns: ('', '',)

```
In [144... s = "2025-05-27"
result = s.rpartition("-")
print(result)
```

('2025-05', '-', '27')

- Separator found
- Note: It splits at the last dash -.

```
In [149... s = "a=b=c"
result = s.rpartition("=")
print(result)
```

('a=b', '=', 'c')

Multiple separators

```
In [152... s = "hello"
result = s.rpartition(":")
print(result)
```

('', '', 'hello')

Separator not found

```
In [155... filename = "report.final.v2.pdf"
name, sep, ext = filename.rpartition(".")
print(ext) # Output: 'pdf'
```

pdf

- Real-world use (extract file extension)

s.split(sep, maxsplit)

- `str.split(sep=None, maxsplit=-1)`
- `sep`: The delimiter to split the string on (default is any whitespace).
- `maxsplit` (optional): Maximum number of splits. If not specified or -1, splits all possible times.
- Returns a list of substrings.

```
In [161... s = "Python is fun"
print(s.split())
```

```
['Python', 'is', 'fun']
```

Default split (on whitespace)

```
In [164... s = "apple,banana,cherry"
print(s.split(","))
```

```
['apple', 'banana', 'cherry']
```

Split by custom separator

```
In [169... s = "one-two-three-four"
print(s.split("-", 2)) #Using maxsplit
```

```
['one', 'two', 'three-four']
```

Only the first 2 - are used to split.

```
In [172... s = "hello world"
print(s.split(","))
```

```
['hello world']
```

Separator not found

```
In [180... s = "Python    is    awesome"
print(s.split())
```

```
['Python', 'is', 'awesome']
```

Splitting with multiple spaces

Default behavior treats any amount of whitespace as a single separator.

`str.rsplit(sep=None, maxsplit=-1)`

- `sep`: The delimiter to split the string from the right.
- `maxsplit`: Maximum number of splits from the right.
- Returns a list of substrings.
- Default separator: whitespace (if `sep` is not provided).

- maxsplit = -1 means split as many times as possible.

```
In [185... s = "Python is very fun"
print(s.rsplit())
```

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

Default behavior (splits on whitespace from right)

```
In [188... s = "a-b-c-d"
print(s.rsplit("-"))
```

```
['a', 'b', 'c', 'd']
```

Split by custom separator

```
In [193... s = "a-b-c-d"
print(s.rsplit("-", 2))
```

```
['a-b', 'c', 'd']
```

- Using maxsplit
- Split starts from the right, so first split gives d, then c, the rest is a-b.

```
In [197... s = "hello world"
print(s.rsplit(","))
```

```
['hello world']
```

Separator not found

```
In [200... filename = "archive.tar.gz"
print(filename.rsplit(".", 1))
```

```
['archive.tar', 'gz']
```

Splitting filenames

str.splitlines(keepends=False)

- Splits a string into a list of lines, breaking at line boundaries (\n, \r, \r\n).
- If keepends=True, the line breaks (\n, \r, etc.) are kept in the output.
- Default behavior is keepends=False, so line endings are removed.

```
In [204... s = "Hello\nWorld\nPython"
print(s.splitlines())
```

```
['Hello', 'World', 'Python']
```

Basic usage

```
In [209... s = "Line1\nLine2\rLine3\r\nLine4"
print(s.splitlines(keepends=True))
```

```
['Line1\n', 'Line2\r', 'Line3\r\n', 'Line4']
```

Keep line endings

In [212...

```
s = """This is line 1
This is line 2
This is line 3"""
print(s.splitlines())
```

```
['This is line 1', 'This is line 2', 'This is line 3']
```

Multi-line string from triple quotes

In [215...

```
s = ""
print(s.splitlines())
```

```
[]
```

Empty string

Inspection II

s[i:j] — String Slicing

- Returns a substring starting from index i up to but not including index j.
- i is the start index (inclusive).
- j is the end index (exclusive).
- If i is omitted, it defaults to 0.
- If j is omitted, it goes until the end of the string.

In [221...

```
s = "Python"
print(s[0:3])
```

Pyt

From index 0 to index 2 (j=3 is excluded)

In [224...

```
s = "Python"
print(s[:4])
```

Pyth

Omitting the start index

In [227...

```
s = "Python"
print(s[2:])
```

thon

Omitting the end index

```
In [230]: s = "Python"
          print(s[-4:-1])
```

tho

Starts 4 from the end (t), stops at 1 from the end (n is excluded)

```
In [233]: s = "Python"
          print(s[:])
```

Python

Entire string

s[i:j:k] — Slicing with a Step

- i: Start index (inclusive)
- j: End index (exclusive)
- k: Step – how many characters to skip
- If k is:

Positive → slice moves left to right

Negative → slice moves right to left

```
In [4]: s = "abcdef"
        print(s[0:6:2])
```

ace

Slice with step 2

Starts at index 0, takes every 2nd character until index 6 (excluded)

```
In [7]: s = "Python"
        print(s[::-1])
```

nohtyP

No start/end specified, step = -1 → reverses the string

```
In [10]: s = "abcdefgh"
         print(s[2:7:2])
```

ceg

Slice every 2nd letter from middle

```
In [13]: s = "abcdefgh"
         print(s[6:2:-1])
```

gfed

Reverse part of the string

Starts at index 6 (g), moves backward, stops before index 2 (c is excluded)

```
In [17]: s = "hello"  
print(s[::-10])
```

h

Step larger than string length

Only the first character fits into this step range.

s.endswith((s1, s2, s3))

str.endswith(tuple_of_suffixes)

- Checks if the string **ends with any of the strings in the tuple.
- Returns True if at least one matches.
- Returns False if none match.

```
In [22]: s = "report_final.pdf"  
result = s.endswith((".pdf", ".docx", ".txt"))  
print(result)
```

True

Match found

Because the string ends with .pdf, which is in the tuple.

```
In [26]: s = "image.jpeg"  
result = s.endswith((".png", ".gif", ".bmp"))  
print(result)
```

False

No match

None of the suffixes in the tuple match .jpeg.

```
In [29]: s = "README.TXT"  
result = s.endswith((".txt", ".doc"))  
print(result)
```

False

Case-sensitive check

Ends with .TXT, but .txt is lowercase — Python's string methods are case-sensitive.

```
In [32]: extensions = (".mp3", ".wav", ".flac")  
filename = "track01.mp3"  
print(filename.endswith(extensions))
```


True

Handling dynamic suffix list

.startswith()

str.startswith(tuple_of_prefixes)

- Checks if the string starts with any of the values in the tuple.
- Returns True if a match is found.
- Returns False otherwise.
- This is case-sensitive (just like .endswith()).

```
In [38]: s = "welcome_home.txt"
print(s.startswith(("welcome", "home", "report")))
```

True

Match found

s starts with "welcome", which is in the tuple.

```
In [42]: s = "greeting_message.txt"
print(s.startswith(("hello", "hi", "welcome")))
```

False

No match

None of the prefixes match the start of the string.

```
In [48]: s = "Invoice_2025.pdf"
print(s.startswith(("invoice",)))
```

False

"Invoice" has an uppercase "I", so it does not match "invoice" (case-sensitive).

```
In [51]: prefixes = ("img_", "vid_", "doc_")
filename = "img_12345.png"
print(filename.startswith(prefixes))
```

True

Using a tuple dynamically

str.isidentifier()

- Returns True if the string is a valid Python identifier.
- A valid identifier:

1. Starts with a letter (a–z, A–Z) or underscore _
2. Followed by letters, digits (0–9), or underscores
3. Cannot be a keyword (e.g., if, class)
4. Cannot start with a digit

```
In [56]: s = "variable_name1"  
print(s.isidentifier())
```

True

Valid identifier

```
In [59]: s = "1variable"  
print(s.isidentifier())
```

False

Starts with a digit

```
In [62]: s = "user-name"  
print(s.isidentifier())
```

False

Contains special character

```
In [65]: s = "class"  
print(s.isidentifier())
```

True

Python keyword (still returns True)

Even though "class" is a Python keyword, it is a valid identifier — but you still shouldn't use it as a variable name. To check for keywords, use:

```
In [69]: import keyword  
print(keyword.iskeyword("class")) # True
```

True

```
In [72]: s = "_"   
print(s.isidentifier())
```

True

Underscore-only (valid)

str.isprintable()

- Returns True if all characters in the string are printable.
- Printable characters include:

Letters, digits, punctuation, whitespace (like spaces)

- It returns False if the string contains non-printable characters like:

1. Newlines (\n)
2. Tabs (\t)
3. Escape characters

```
In [103]: s = "Hello! 123"
print(s.isprintable())
print(s)
```

True

Hello! 123

All printable characters

```
In [101]: s = "Hello\nWorld"
print(s.isprintable())
print(s)
```

False

Hello

World

Contains newline (\n)

\n is a non-printable character.

```
In [99]: s = ""
print(s.isprintable())
print(s)
```

True

Empty string

An empty string is considered printable.

```
In [97]: s = "Name\tAge"
print(s.isprintable())
print(s)
```

False

Name Age

Contains tab (\t)

```
In [95]: s = "C:\\Users\\Kirti"
print(s.isprintable())
print(s)
```

True

C:\Users\Kirti

Escape sequences like \ are printable if they resolve to a printable character (here, a backslash).

Whitespace - II

str.center(width, pad_char)

- Returns a centered string in a field of given width.
- Pads the string with the specified pad_char (a single character).
- If pad_char is not provided, it defaults to a space ' '.

In [110...

```
s = "Python"
print(s.center(12))
```

Python

- Center with spaces (default)
- "Python" is centered within 12 characters, padded with spaces.

In [114...

```
s = "Python"
print(s.center(12, '*'))
```

Python

Center with * as padding

In [117...

```
s = "Python"
print(s.center(4, '-'))
```

Python

Width less than string length

No padding is applied — original string is returned if width is less than or equal to its length.

In [121...

```
s = "Hi"
print(s.center(7, '.'))
```

...Hi..

Uneven padding (Python handles extra on the right)

str.expandtabs(tabsize)

- Replaces each tab character (\t) in the string with spaces.
- The number of spaces depends on the tabsize argument.

- Default tabsize is 8 if not specified.

In [125...

```
s = "Name\tAge\tCity"
print(s.expandtabs())
```

```
Name      Age      City
```

Default behavior (tabsize = 8)

Each `\t` is replaced with spaces so that the next item starts at the next multiple of 8 characters.

In [137...

```
s = "Name\tAge\tCity"
print(s.expandtabs(4))
```

```
Name      Age City
```


The original string is:

- "Name\tAge\tCity"
- `\t` represents a tab character, and `expandtabs(4)` means each tab should align the next character to the next multiple of 4 positions.

Step-by-step expansion (tab size = 4):

- "Name" is 4 characters, so the first `\t` is at position 4.
- Next multiple of 4 = 4, but since we're already at 4, we move to 8.
- So the tab adds 4 spaces → now at position 8.
- "Age" is 3 characters.
- Current position = 8, plus 3 characters = 11
- Next `\t` is at position 11
- Next multiple of 4 after 11 is 12, so `\t` adds 1 space to reach 12.
- "City" begins at position 12.
- Final Output (with visible spaces):

```
Name Age City
```

- [4 chars]"Name"
- [4 spaces]" "
- [3 chars]"Age"
- [1 space]" "
- [4 chars]"City"
-  So `expandtabs(4)` replaced each `\t` with just enough spaces to align to the next multiple of 4, creating neatly aligned columns.

```
In [139... s = "Name\tAge\tCity"
print(s.expandtabs(10))
```

Name Age City

Custom tab size (tabsize = 4)

```
In [132... s = "A\tB\tC"
print(s.expandtabs(2))
```

A B C

- Custom tab size (tabsize = 2)
- Each tab (\t) turns into 1–2 spaces depending on the current position in the string.

```
In [141... s = "Item\tPrice\tQty"
print(s.expandtabs(6))
```

Item Price Qty

Mixed tab size formatting

str.zfill(width)

- Returns a copy of the string left-padded with zeros (0) until it reaches the specified width.
- Works best with numeric strings, but can be used with any string.
- If the string has a sign (+ or -), the zeros are added after the sign.

```
In [184... s = "42"
print(s.zfill(5))
```

00042

Pad a number string

It adds 3 leading zeros to make the total width 5.

```
In [188... s = "12345"
print(s.zfill(4))
```

12345

- No padding needed (already long enough)
- No change because the string length is already \geq width.

```
In [195... s = "-89"
print(s.zfill(5))
```

-0089

- Negative number string
- The - sign stays at the front, and padding comes after it.

```
In [198... s = "cat"  
print(s.zfill(6))
```

000cat

- Non-numeric string
- It still pads with zeros even though it's not a number.

```
In [ ]:
```