

4.7.1. String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

`str.capitalize()`

Return a copy of the string with its first character capitalized and the rest lowercased.

`str.casefold()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

`str.center(width[, fillchar])`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring *sub* in the range `[start, end]`. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.encode(encoding="utf-8", errors="strict")`

Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. *errors* may be given to set a different error handling scheme. The default for *errors* is 'strict', meaning that encoding errors raise a [UnicodeError](#). Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via [codecs.register_error\(\)](#), see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

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

Return True if the string ends with the specified *suffix*, otherwise return False. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

`str.expandtabs(tabsize=8)`

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

str.**find**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice *s*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> 'Py' in 'Python'
True
```

```
>>>
```

str.**format**(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

```
>>>
```

str.**format_map**(*mapping*)

Similar to `str.format(**mapping)`, except that *mapping* is used directly and not copied to a `dict`. This is useful if for example *mapping* is a `dict` subclass:

str.**index**(*sub*[, *start*[, *end*]])

Like `find()`, but raise `ValueError` when the substring is not found.

str.**isalnum**()

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise. A character *c* is alphanumeric if one of the following returns True: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

str.**isalpha**()

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “LI”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard.

str.**isdecimal**()

Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those from general category “Nd”. This category includes digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

str.**isdigit**()

Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. Formally, a digit is a character that has the property value `Numeric_Type=Digit` or `Numeric_Type=Decimal`.

`str.isidentifier()`

Return true if the string is a valid identifier according to the language definition, section *Identifiers and keywords*.

Use `keyword.iskeyword()` to test for reserved identifiers such as `def` and `class`.

`str.islower()`

Return true if all cased characters [4] in the string are lowercase and there is at least one cased character, false otherwise.

`str.isnumeric()`

Return true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value `Numeric_Type=Digit`, `Numeric_Type=Decimal` or `Numeric_Type=Numeric`.

`str.isprintable()`

Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

`str.isspace()`

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as “Other” or “Separator” and those with bidirectional property being one of “WS”, “B”, or “S”.

`str.istitle()`

Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

`str.isupper()`

Return true if all cased characters [4] in the string are uppercase and there is at least one cased character, false otherwise.

`str.join(iterable)`

Return a string which is the concatenation of the strings in the *iterable* iterable. A

`TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

`str.ljust(width[, fillchar])`

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.lower()`

Return a copy of the string with all the cased characters [4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.lstrip([chars])`

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

`static str.maketrans(x[, y[, z]])`

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *x* will be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

`str.partition(sep)`

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

`str.replace(old, new[, count])`

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.rfind(sub[, start[, end]])`

Return the highest index in the string where substring *sub* is found, such that *sub* is

contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.rjust(width[, fillchar])`

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.rpartition(sep)`

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

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

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`str.rstrip([chars])`

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

>>>

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

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
```

>>>

```
[ '1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,.'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is *None*, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a *None* separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

`str.splitlines([keepends])`

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of [universal newlines](#).

Representation	Description
<code>\n</code>	Line Feed
<code>\r</code>	Carriage Return
<code>\r\n</code>	Carriage Return + Line Feed
<code>\v</code> or <code>\x0b</code>	Line Tabulation
<code>\f</code> or <code>\x0c</code>	Form Feed
<code>\x1c</code>	File Separator
<code>\x1d</code>	Group Separator
<code>\x1e</code>	Record Separator
<code>\x85</code>	Next Line (C1 Control Code)
<code>\u2028</code>	Line Separator
<code>\u2029</code>	Paragraph Separator

Changed in version 3.2: `\v` and `\f` added to list of line boundaries.

For example:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
```

```
['ab c', '', 'de fg', 'kl']  
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)  
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike `split()` when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()  
[]  
>>> "One line\n".splitlines()  
['One line']
```

For comparison, `split('\n')` gives:

```
>>> ''.split('\n')  
['']  
>>> 'Two lines\n'.split('\n')  
['Two lines', '']
```

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

Return True if string starts with the *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

str.strip([*chars*])

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()  
'spacious'  
>>> 'www.example.com'.strip('cmowz.')  
'example'
```

The outermost leading and trailing *chars* argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in *chars*. A similar action takes place on the trailing end. For example:

```
>>> comment_string = '#..... Section 3.2.1 Issue #32 .....'  
>>> comment_string.strip('#! ')  
'Section 3.2.1 Issue #32'
```

str.swapcase()

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

str.title()

Return a titlecased version of the string where words start with an uppercase

character and the remaining characters are lowercase.

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

```
>>>
```

`str.translate(table)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a [mapping](#) or [sequence](#). When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a [LookupError](#) exception, to map the character to itself.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

See also the [codecs](#) module for a more flexible approach to custom character mappings.

`str.upper()`

Return a copy of the string with all the cased characters [\[4\]](#) converted to uppercase. Note that `str.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.zfill(width)`

Return a copy of the string left filled with ASCII ‘0’ digits to make a string of length *width*. A leading sign prefix (‘+’/‘-’) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

```
>>>
```

4.7.2. printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer `str.format()` interface helps avoid these errors, and also