# String Methods All Versions

## Examples

### Changing the capitalization of a string

Python's string type provides many functions that act on the capitalization of a string. These include :

- str.casefold
- str.upper
- str.lower
- str.capitalize
- str.title
- str.swapcase

With unicode strings (the default in Python 3), these operations are **not** 1:1 mappings or reversible. Most of these operations are intended for display purposes, rather than normalization.

---

Python 3.x ≥3.3

*str.casefold()*

str.casefold creates a lowercase string that is suitable for case insensitive comparisons. This is more aggressive than str.lower and may modify strings that are already in lowercase or cause strings to grow in length, and is not intended for display purposes.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

The transformations that take place under casefolding are defined by the Unicode Consortium in the CaseFolding.txt file on their website.

---

*str.upper()*

str.upper takes every character in a string and converts it to its uppercase equivalent, for example:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

---

*str.lower()*

str.lower does the opposite; it takes every character in a string and converts it to its lowercase equivalent:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

---

*str.capitalize()*

str.capitalize returns a capitalized version of the string, that is, it makes the first character have upper case and the rest lower:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

---

*str.title()*

str.title returns the title cased version of the string, that is, every letter in the beginning of a word is made upper case and all others are made lower case:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

---

*str.swapcase()*

str.swapcase returns a new string object in which all lower case characters are swapped to upper case and all upper case characters to lower:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

*Usage as `str` class methods*

It is worth noting that these methods may be called either on string objects (as shown above) or as a class method of the `str` class (with an explicit call to `str.upper`, etc.)

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

This is most useful when applying one of these methods to many strings at once in say, a *map* function.

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

### str.translate: Translating characters in a string

Python supports a `translate` method on the `str` type which allows you to specify the translation table (used for replacements) as well as any characters which should be deleted in the process.

```
str.translate(table[, deletechars])
```

| Parameter | Description |
|---|---|
| table | It is a lookup table that defines the mapping from one character to another. |
| deletechars | A list of characters which are to be removed from the string. |

The `maketrans` method ( `str.maketrans` in Python 3 and `string.maketrans` in Python 2) allows you to generate a translation table.

```
translation_table = str.maketrans("aeiou", "12345")
my_string = "This is a string!"
translated = my_string.translate(translation_table)
# out: 'Th3s 3s 1 str3ng!'
```

The `translate` method returns a string which is a translated copy of the original string.

---

You can set the `table` argument to `None` if you only need to delete characters.

```
'this syntax is very useful'.translate(None, 'aeiou')
# Out: 'ths syntx s vry sfl'
```

### str.format: Format values into a string

Python provides string interpolation and formatting functionality through the `str.format` function, introduced in version 2.6.

Given the following variables:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

The following statements are all equivalent

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"

"{} {} {} {} {}".format(i, f, s, l, d)

str.format("{} {} {} {} {}", i, f, s, l, d)

"{0} {1} {2} {3} {4}".format(i, f, s, l, d)

"{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)

"{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
```

For reference, Python also supports C-style qualifiers for string formatting. The examples below are equivalent to those above, but the `str.format` versions are preferred due to benefits in flexibility, consistency

of notation, and extensibility:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)

"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

The braces uses for interpolation in `str.format` can also be numbered to reduce duplication when formatting strings. For example, the following are equivalent:

```
"I am from Australia. I love cupcakes from Australia!"

"I am from {}. I love cupcakes from {}!".format("Australia", "Australia")

"I am from {0}. I love cupcakes from {0}!".format("Australia")
```

While the official python documentation is, as usual, thorough enough, pyformat.info has a great set of examples with detailed explanations.

Additionally, the `{` and `}` characters can be escaped by using double brackets:

```
desired output: "{'a': 5, 'b': 6}"
code:           print "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)
```

See String Formatting for additional information. `str.format()` was proposed in PEP 3101 .

### String module's useful constants

Python's `string` module provides constants for string related operations. To use them, import the `string` module:

```
import string
```

*`string.ascii_letters`:*

Concatenation of `ascii_lowercase` and `ascii_uppercase`:

```
print(string.ascii_letters)
# Output: 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

*`string.ascii_lowercase`:*

Contains all lower case ASCII characters:

```
print(string.ascii_lowercase)
# Output: 'abcdefghijklmnopqrstuvwxyz'
```

---

*`string.ascii_uppercase`:*

Contains all upper case ASCII characters:

```
print(string.ascii_uppercase)
# Output: 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

---

*`string.digits`:*

Contains all decimal digit characters:

```
print(string.digits)
# Output: '0123456789'
```

---

*`string.hexdigits`:*

Contains all hex digit characters:

```
print(string.hexdigits)
# Output: '0123456789abcdefABCDEF'
```

---

*`string.octaldigits`:*

Contains all octal digit characters:

```
print(string.octaldigits)
```

```
print(string.octaldigits)
# Output: '01234567'
```

---

*string.punctuation :*

Contains all characters which are considered punctuation in the C locale:

```
print(string.punctuation)
# Output: '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

---

*string.whitespace :*

Contains all ASCII characters considered whitespace:

```
# wrapped in repr so print won't evaluate escape characters such as tab/newline
print(repr(string.whitespace))
# Output: ' \t\n\r\x0b\x0c'
```

---

*string.printable :*

Contains all characters which are considered printable; a combination of string.digits , string.ascii_letters , string.punctuation , and string.whitespace .

```
print(string.printable)
# Output: '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@
```

---

### Replace all occurrences of one substring with another substring

Python's str type also has a method for replacing occurences of one sub-string with another sub-string in a given string. For more demanding cases, one can use re.sub .

---

*str.replace(old, new[, count]) :*

str.replace takes two arguments old and new containing the old sub-string which is to be replaced by the new sub-string. The optional argument count specifies the number of replacements to be made:

For example, in order to replace 'foo' with 'spam' in the following string, we can call str.replace with old = 'foo' and new = 'spam' :

```
"Make sure to foo your sentence.".replace('foo', 'spam')
# "Make sure to spam your sentence."
```

If the given string contains multiple examples that match the old argument, **all** occurrences are replaced with the value supplied in new :

```
"It can foo multiple examples of foo if you want.".replace('foo', 'spam')
# "It can spam multiple examples of spam if you want."
```

unless, of course, we supply a value for count . In this case count occurrences are going to get replaced:

```
"""It can foo multiple examples of foo if you want,
or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
# """It can spam multiple examples of foo if you want,
# or you can limit the foo with the third argument."""
```

---

### Reversing a string

A string can reversed using the built-in reversed() function, which takes a string and returns an iterator in reverse order.

```
reversed('hello') # = ['o','l','l','e','h']
```

reversed() can be wrapped in a call to ''.join() to make a string from the iterator.

```
''.join(reversed('hello'))   # = 'olleh'
```

While using reversed() might be more readable to uninitiated Python users, using extended slicing with a step of -1 is faster and more concise. Here , try to implement it as function:

```
def reversed_string(main_string):
    return main_string[::-1]

reversed_string('hello')   # = 'olleh'
```

### Split a string based on a delimiter into a list of strings

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

str.split takes a string and returns a list of substrings of the original string. The behavior differs depending on whether the sep argument is provided or omitted.

If sep isn't provided, or is None , then the splitting takes place wherever there is whitespace. However, leading and trailing whitespace is ignored, and multiple consecutive whitespace characters are treated the same as a single whitespace character:

```
"This is a sentence.".split()
# Output: ['This', 'is', 'a', 'sentence.']

" This is    a sentence.  ".split()
# Output: ['This', 'is', 'a', 'sentence.']

"            ".split()
# Output: []
```

The sep parameter can be used to define a delimiter string. The original string is split where the delimiter string occurs, and the delimiter itself is discarded. Multiple consecutive delimiters are *not* treated the same as a single occurrence, but rather cause empty strings to be created.

```
"This is a sentence.".split(' ')
# ['This', 'is', 'a', 'sentence.']

"Earth,Stars,Sun,Moon".split(',')
# ['Earth', 'Stars', 'Sun', 'Moon']

" This is    a sentence.  ".split(' ')
# ['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

"This is a sentence.".split('e')
# ['This is a s', 'nt', 'nc', '.']

"This is a sentence.".split('en')
# ['This is a s', 't', 'ce.']
```

The default is to split on *every* occurrence of the delimiter, however the maxsplit parameter limits the number of splittings that occur. The default value of -1 means no limit:

```
"This is a sentence.".split('e', maxsplit=0)
# ['This is a sentence.']

"This is a sentence.".split('e', maxsplit=1)
# ['This is a s', 'ntence.']

"This is a sentence.".split('e', maxsplit=2)
# ['This is a s', 'nt', 'nce.']

"This is a sentence.".split('e', maxsplit=-1)
# ['This is a s', 'nt', 'nc', '.']
```

---

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

str.rsplit ("right split") differs from str.split ("left split") when maxsplit is specified. The splitting starts at the end of the string rather than at the beginning:

```
"This is a sentence.".rsplit('e', maxsplit=1)
# ['This is a sentenc', '.']

"This is a sentence.".rsplit('e', maxsplit=2)
# ['This is a sent', 'nc', '.']
```

**Note** : Python specifies the maximum number of *splits* performed, while most other programming languages specify the maximum number of *substrings* created. This may create confusion when porting or comparing code.

### Stripping unwanted leading/trailing characters from a string

Three methods are provided that offer the ability to strip leading and trailing characters from a string:

Three methods are provided that offer the ability to strip leading and trailing characters from a string: str.strip , str.rstrip and str.lstrip . All three methods have the same signature and all three return a new string object with unwanted characters removed.

---

*str.strip([chars])*

str.strip acts on a given string and removes (strips) any leading or trailing characters contained in the argument chars ; if chars is not supplied or is None , all white space characters are removed by default. For example:

```
value = "    a line with leading and trailing space     "
new_value = value.strip()
print(new_value) # Outputs: 'a line with leading and trailing space'
```

If chars is supplied, all characters contained in it are removed from the string, which is returned. For example:

```
value = ">>> a Python prompt"
new_value = value.strip('> ')  # strips '>' character and space character
print(new_value) # Outputs: 'a Python prompt'
```

---

*str.rstrip([chars]) and str.lstrip([chars])*

These methods have similar semantics and arguments with str.strip() , their difference lies in the direction from which they start. str.rstrip() starts from the end of the string while str.lstrip() splits from the start of the string.

For example, using str.rstrip :

```
value = "     spacious string      "
right_strip = value.rstrip()
print(right_strip)  # Output: '     spacious string'
```

While, using str.lstrip :

```
value = "     spacious string      "
right_strip = value.rstrip()
print(right_strip)  # Output: 'spacious string      '
```

### Testing what a string is composed of

Python's str type also features a number of methods that can be used to evaluate the contents of a string. These are str.isalpha , str.isdigit , str.isalnum , str.isspace . Capitalization can be tested with str.isupper , str.islower and str.istitle .

---

*str.isalpha*

str.isalpha takes no arguments and returns True if the all characters in a given string are alphabetic, for example:

```
"Hello World".isalpha()  # False (contains spaces)
"Hello2World".isalpha()  # False (contains number)
"HelloWorld!".isalpha()  # False (contains punctuation)
"HelloWorld".isalpha()   # True
```

As an edge case, the empty string evaluates to False when used with "".isalpha() .

---

*str.isupper , str.islower , str.istitle*

These methods test the capitalization in a given string.

str.isupper is a method that returns True if all characters in a given string are uppercase and False otherwise.

```
"HeLLO WORLD".isupper()  # False
"HELLO WORLD".isupper()  # True
"".isupper()             # False
```

Conversely, str.islower is a method that returns True if all characters in a given string are lowercase and False otherwise.

```
"Hello world".islower()  # False
"hello world".islower()  # True
"".islower()             # False
```

`str.istitle` returns `True` if the given string is title cased; that is, every word begins with an uppercase character followed by lowercase characters.

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

---

*`str.isdecimal`, `str.isdigit`, `str.isnumeric`*

`str.isdecimal` returns whether the string is a sequence of decimal digits, suitable for representing a decimal number.

`str.isdigit` includes digits not in a form suitable for representing a decimal number, such as superscript digits.

`str.isnumeric` includes any number values, even if not digits, such as values outside the range 0-9.

```
            isdecimal    isdigit   isnumeric

12345        True        True       True
৲২੩৫        True        True       True
①²³₅       False       True       True
⑩12.         False       False      True
Five         False       False      False
```

Bytestrings ( `bytes` in Python 3, `str` in Python 2), only support `isdigit`, which only checks for basic ASCII digits.

As with `str.isalpha`, the empty string evaluates to `False`.

---

*`str.isalnum`*

This is a combination of `str.isalpha` and `str.isnumeric`, specifically it evaluates to `True` if all characters in the given string are **alphanumeric**, that is, they consist of alphabetic *or* numeric characters:

```
"Hello2World".isalnum()  # True
"HelloWorld".isalnum()   # True
"2016".isalnum()         # True
"Hello World".isalnum()  # False (contains whitespace)
```

---

*`str.isspace`*

Evaluates to `True` if the string contains only whitespace characters.

```
"\t\r\n".isspace()  # True
" ".isspace()       # True
```

Sometimes a string looks “empty” but we don't know whether it's because it contains just whitespace or no character at all

```
my_str = ""
my_str.isspace()    # False
```

To cover this case we need an additional test

```
my_str = ''
my_str.isspace()                # False
my_str.isspace() or not my_str  # True
```

But the shortest way to test if a string is empty or just contains whitespace characters is to use `strip` (with no arguments it removes all leading and trailing whitespace characters)

```
not my_str.strip()  # True
```

---

### Join a list of strings into one string

A string can be used as a separator to join a list of strings together into a single string using the `join()` method. For example you can create a string where each element in a list is separated by a space.

```
a = ["once","upon","a","time"]
" ".join(a)
```

Which returns `"once upon a time"`

The following example separates the string elements with three hyphens.

```
a = ["once", "upon", "a", "time"]
"---".join(a)
```

Which returns `"once---upon---a---time"`

### Case insensitive string comparisons

Comparing string in a case insensitive way seems like something that's trivial, but it's not. This section only considers unicode strings (the default in Python 3). Note that Python 2 may have subtle weaknesses relative to Python 3 - the later's unicode handling is much more complete.

The first thing to note it that case-removing conversions in unicode aren't trivial. There is text for which `text.lower() != text.upper().lower()` , such as `"ß"` :

```
"ß".lower()
#>>> 'ß'

"ß".upper().lower()
#>>> 'ss'
```

But let's say you wanted to caselessly compare `"BUSSE"` and `"Buße"` . Heck, you probably also want to compare `"BUSSE"` and `"BUßE"` equal - that's the newer capital form. The recommended way is to use `casefold` :

Python 3.x ≥3.3

```
help(str.casefold)
#>>> Help on method_descriptor:
#>>>
#>>> casefold(...)
#>>>     S.casefold() -> str
#>>>
#>>>     Return a version of S suitable for caseless comparisons.
#>>>
```

Do not just use `lower` . If `casefold` is not available, doing `.upper().lower()` helps (but only somewhat).

Then you should consider accents. If your font renderer is good, you probably think `"ê" == "ê"` - but it doesn't:

```
"ê" == "ê"
#>>> False
```

This is because they are actually

```
import unicodedata

[unicodedata.name(char) for char in "ê"]
#>>> ['LATIN SMALL LETTER E WITH CIRCUMFLEX']

[unicodedata.name(char) for char in "ê"]
#>>> ['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

The simplest way to deal with this is `unicodedata.normalize` . You probably want to use **NFKD** normalization, but feel free to check the documentation. Then one does

```
unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")
#>>> True
```

To finish up, here this is expressed in functions:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

### Counting number of times a substring appears in a string

One method is available for counting the number of occurrences of a sub-string in another string, `str.count` .

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

str.count returns an int indicating the number of non-overlapping occurrences of the sub-string sub in another string. The optional arguments start and end indicate the beginning and the end in which the search will take place. By default start = 0 and end = len(str) meaning the whole string will be searched:

```
s = "She sells seashells by the seashore."
s.count("sh")        # 2
s.count("se")        # 3
s.count("sea")       # 2
s.count("seashells") # 1
```

By specifying a different value for start , end we can get a more localized search and count, for example, if start is equal to 13 the call to:

```
s.count("sea", 13)
```

is equivalent to:

```
t = s[start:]
t.count("sea")  # 1
```

### String Contains

Python makes it extremely intuitive to check if a string contains a given substring. Just use the in operator:

```
my_str = "foo.baz.bar"
"foo" in my_str
# Out: True
```

Note: testing an empty string will always result in True :

```
"" in "test"
# Out: True
```

### Justify strings

Python provides functions for justifying strings, enabling text padding to make aligning various strings much easier.

Below is an example of str.ljust and str.rjust :

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4), str(kms).ljust

#  40 -> 2555 mi. (4112 km.)
#  19 -> 63   mi. (102  km.)
#   5 -> 1381 mi. (2222 km.)
#  93 -> 189  mi. (305  km.)
```

ljust and rjust are very similar. Both have a width parameter and an optional fillchar parameter. Any string created by these functions is at least as long as the width parameter that was passed into the function. If the string is longer than width alread, it is not truncated. The fillchar argument, which defaults to the space character ' ' must be a single character, not a multicharacter string.

The ljust function pads the end of the string it is called on with the fillchar until it is width characters long. The rjust function pads the beginning of the string in a similar fashion. Therefore, the l and r in the names of these functions refer to the side that the original string, *not the fillchar* , is positioned in the output string.

### Test the starting and ending characters of a string

In order to test the beginning and ending of a given string in Python, one can use the methods str.startswith() and str.endswith() .

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

As it's name implies, `str.startswith` is used to test whether a given string starts with the given characters in `prefix`.

```
s = "This is a test string"
s.startswith("T")           # True
s.startswith("Thi")         # True
s.startswith("thi")         # False
```

The optional arguments `start` and `end` specify the start and end points from which the testing will start and finish. In the following example, by specifying a start value of `2` our string will be searched from position 2 and afterwards:

```
s.startswith("is", 2)        # True
```

This yields `True` since `s[2] == 'i'` and `s[3] == 's'`.

You can also use a `tuple` to check if it starts with any of a set of strings

```
s.startswith(('This', 'That'))       # True, starts with 'This'
s.startswith(('ab', 'bc'))      # False, not starts with 'ab' or 'bc'
```

---

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

`str.endswith` is exactly similar to `str.startswith` with the only difference being that it searches for ending characters and not starting characters. For example, to test if a string ends in a full stop, one could write:

```
s = "this ends in a full stop."
s.endswith('.')   # True
s.endswith('!')   # False
```

as with `startswith` more than one characters can used as the ending sequence:

```
s.endswith('stop.')  # True
s.endswith('Stop.')  # False
```

You can also use a `tuple` to check if it ends with any of a set of strings

```
s.endswith(('.', 'something'))  # True, ends with '.'
s.endswith(('ab', 'bc'))  # False, not ends with 'ab' or 'bc'
```

---

### Conversion between str or bytes data and unicode characters

The contents of files and network messages may represent encoded characters. They often need to be converted to unicode for proper display.

In Python 2, you may need to convert str data to Unicode characters. The default ( `''`, `""`, etc.) is an ASCII string, with any values outside of ASCII range displayed as escaped values. Unicode strings are `u''` (or `u""`, etc.).

Python 2.x ≥2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printe
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a U
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII char
```

In Python 3 you may need to convert arrays of bytes (referred to as a 'byte literal') to strings of Unicode characters. The default is now a Unicode string, and bytestring literals must now be entered as `b''`, `b""`, etc. A byte literal will return `True` to `isinstance(some_val, byte)`, assuming `some_val` to be a string that might be encoded as bytes.

Python 3.x ≥3.0

```
# You get from file or network "© abc" encoded in UTF-8

s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Pytho
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescape
```

## Syntax

```
str.capitalize() -> str
str.casefold() -> str [only for Python > 3.3]
str.center(width[, fillchar]) -> str
str.count(sub[, start[, end]]) -> int
str.decode(encoding="utf-8"[, errors]) -> unicode [only in Python 2.x]
str.encode(encoding="utf-8", errors="strict") -> bytes
str.endswith(suffix[, start[, end]]) -> bool
str.expandtabs(tabsize=8) -> str
str.find(sub[, start[, end]]) -> int
str.format(*args, **kwargs) -> str
str.format_map(mapping) -> str
str.index(sub[, start[, end]]) -> int
str.isalnum() -> bool
str.isalpha() -> bool
str.isdecimal() -> bool
str.isdigit() -> bool
str.isidentifier() -> bool
str.islower() -> bool
str.isnumeric() -> bool
str.isprintable() -> bool
str.isspace() -> bool
str.istitle() -> bool
str.isupper() -> bool
str.join(iterable) -> str
str.ljust(width[, fillchar]) -> str
str.lower() -> str
str.lstrip([chars]) -> str
```

```
static str.maketrans(x[, y[, z]])
str.partition(sep) -> (head, sep, tail)
str.replace(old, new[, count]) -> str
str.rfind(sub[, start[, end]]) -> int
str.rindex(sub[, start[, end]]) -> int
str.rjust(width[, fillchar]) -> str
str.rpartition(sep) -> (head, sep, tail)
str.rsplit(sep=None, maxsplit=-1) -> list of strings
str.rstrip([chars]) -> str
str.split(sep=None, maxsplit=-1) -> list of strings
str.splitlines([keepends]) -> list of strings
str.startswith(prefix[, start[, end]]) -> book
str.strip([chars]) -> str
str.swapcase() -> str
str.title() -> str
str.translate(table) -> str
str.upper() -> str
str.zfill(width) -> str
```

### Parameters

## Remarks

String objects are immutable, meaning that they can't be modified in place the way a list can. Because of this, methods on the built-in type `str` always return a **new** `str` object, which contains the result of the method call.