Python String Methods

Python has a set of built-in methods that you can call on string objects.

Method	Description
capitalize()	Capitalizes first character of the string
casefold()	Returns a casefolded string
center()	Returns center-aligned string
count()	Counts occurrences of a substring in a string
encode()	Return an encoded version of the string as a bytes object
endswith()	Determines whether the string ends with a given suffix
expandtabs()	Replaces tabs with spaces
find()	Searches the string for a given substring
format()	Perform a string formatting operation
format_map()	Perform a string formatting operation
index()	Searches the string for a given substring

Python String capitalize() Method

Capitalizes first character of the string

Usage

The `capitalize()` method returns a copy of the string with its first character capitalized and the rest lowercased.

The method does not change the original string.

Syntax

string.capitalize()

Basic Example

```
# Capitalize the string
S = 'bob is a CEO at ABC.'
x = S.capitalize()
print(x)
# Prints Bob is a ceo at abc.
```

Non-alphabetic First Character

For the string with non-alphabetic first character, the first character is kept unchanged while the rest is changed to lowercase.

```
S = '42 is my FAVOURITE number.'
x = S.capitalize()
print(x)
# Prints 42 is my favourite number.
```

Python String casefold() Method

Returns a casefolded string

Usage

The `casefold()` method returns a casefolded (lowercase but more aggressive) copy of the string. This method does not change the original string.

Casefolded strings are usually used to 'normalize' text for the purposes of caseless comparison (especially when you want to take characters of many different languages into account).

Syntax

string.casefold()

Basic Example

```
# Make a string casefolded
S = 'Hello, World!'
x = S.casefold()
print(x)
# Prints hello, world!
```

casefold() vs lower()

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string.

For example, the German lowercase letter ‘ß’ is equivalent to ‘ss’. Since it is already lowercase, lower() would do nothing to ‘ß’, but casefold() converts it to ‘ss’.

```
S = 'Das straße'
x = S.casefold()
print(x)
# Prints das strasse


S = 'Das straße'
x = S.lower()
print(x)
# Prints das straße
```

If you are working strictly in the English language, lower() and casefold() returns exactly the same results.

However, if you are trying to normalize text from other languages that use more than English 26-letter alphabet, use casefold() to compare your strings for more consistent results.

Python String center() Method

Returns center-aligned string

Usage

The center() method returns center-aligned string of length width.

Padding is done using the specified fillchar (default is an ASCII space).

The original string is returned as it is, if width is less than or equal to string length.

Syntax

string.center(width,fillchar)

Basic Example

```
# Align text center
S = 'Centered'
x = S.center(14)
print(x)

# Prints  Centered
```

Specify a Fill Character

By default the string is padded with whitespace (ASCII space).

You can modify that by specifying a fill character.

```
# center() with '*' as a fill character
S = 'Centered'
x = S.center(14, '*')
print(x)

# Prints ***Centered***
```

Equivalent Method

You can achieve the same result by using format() method.

```
# Align text center with format()
S = 'Centered'
x = '{:*^14}'.format(S)
print(x)
```

```
# Prints ***Centered***
```

Python String count() Method

Counts occurrences of a substring

Usage

The `count()` method returns the number of times the substring `sub` appears in the `string`.

You can limit the search by specifying optional arguments `start` and `end`.

Syntax

```
string.count(sub,start,end)
```

Basic Example

```
# Count occurrences of 'Big' in the string
S = 'Big, Bigger, Biggest'
x = S.count('Big')
print(x)
# Prints 3
```

Limit count() Search to Substring

If you want to search the string from the middle, specify the `start` parameter.

```
# Count occurrences of 'Big' from 5th character
S = 'Big, Bigger, Biggest'
```

```
x = S.count('Big',5)

print(x)
# Prints 2
```

You can specify where to stop the `count()` search with `end` parameter.

```
# Count occurrences of 'Big' between 5th to 13th character

S = 'Big, Bigger, Biggest'

x = S.count('Big',5,13)

print(x)
# Prints 1
```

Optional arguments start and end are interpreted as in slice notation.

Meaning, `S.count('Big',5,13)` is similar to `S[5:13].count('Big')`

Python String encode() Method

Encodes the string to the specified encoding

Usage

The `encode()` function encodes the string to the specified encoding and returns it as a bytes object.

The string is encoded to UTF-8 by default.

Syntax

string.encode(encoding,errors)

Basic Example

```
# Encode the string to UTF-8
```

```
S = 'Das straße'
x = S.encode()
print(x)
# Prints b'Das stra\xc3\x9fe'
```

Unicode Encode Error Handling

Let's try to encode the German words 'Das straße', which translates to 'The street' in english.

```
S = 'Das straße'
```

Following example shows different error handling scheme implementations by using errors parameter.

```
x = S.encode(encoding='ascii',errors='backslashreplace')
print(x)
# Prints b'Das stra\\xdfe'

x = S.encode(encoding='ascii',errors='ignore')
print(x)
# Prints b'Das strae'

x = S.encode(encoding='ascii',errors='namereplace')
print(x)
# Prints b'Das stra\\N{LATIN SMALL LETTER SHARP S}e'

x = S.encode(encoding='ascii',errors='replace')
print(x)
# Prints b'Das stra?e'

x = S.encode(encoding='ascii',errors='xmlcharrefreplace')
print(x)
```

```
# Prints b'Das straße'


x = S.encode(encoding='UTF-8',errors='strict')
print(x)
# Prints b'Das stra\xc3\x9fe'
```

Python String endswith() Method

Determines whether the string ends with a given suffix

Usage

The `endswith()` method returns True if the string ends with the specified suffix, otherwise returns False.

You can limit the search by specifying optional arguments start and end.

endswith() also accepts a tuple of suffixes to look for.

Syntax

```
string.endswith(suffix,start,end)
```

Basic Examples

```
# Check if the string ends with 'ABC'
S = 'Bob is a CEO at ABC'
x = S.endswith('ABC')
print(x)
# Prints True
```

```
# Check if the string ends with a ' ? '
S = 'Is Bob a CEO?'
x = S.endswith('?')
print(x)
# Prints True
```

Limit endswith() Search to Substring

To limit the search to the substring, specify the start and end parameters.

```
# Check if the substring (4th to 12th character) ends with 'CEO'
S = 'Bob is a CEO at ABC'
x = S.endswith('CEO',4,12)
print(x)
# Prints True
```

Provide Multiple Suffixes to Look for

You can provide multiple suffixes to the method in the form of a tuple. If the string ends with any item of the tuple, the method returns True, otherwise returns False.

```
# Check if the string ends with one of the items in a tuple
S = 'Bob is a CEO'
suffixes = ('CEO','CFO','COO')
x = S.endswith(suffixes)
print(x)
# Prints True


# Check if the string ends with one of the items in a tuple
S = 'Sam is a CFO'
suffixes = ('CEO','CFO','COO')
```

```
x = S.endswith(suffixes)
print(x)
# Prints True
```

Python String expandtabs() Method

Replaces tabs with spaces

Usage

The `expandtabs()` method replaces each tab character '\t' in a string with specified number of spaces (tabsize).

The default tabsize is 8 (tab stop at every eighth column).

Syntax

```
string.exandtabs(tabsize)
```

Basic Example

```
# Expand each tab character with spaces
S1 = 'a\tb\tc'
S2 = 'aaaa\tbbbb\tcccc'
print(S1.expandtabs())
print(S2.expandtabs())
```

```
# Prints a	b	c# Prints aaaa	bbbb	cccc
```

Specify Different Tabsize

The default tabsize is 8. To change the tabsize, specify optional tabsize parameter.

```
# Change the tabsize to 2, 4 and 6
S = 'a\tb\tc'
print(S.expandtabs(2))
print(S.expandtabs(4))
print(S.expandtabs(6))

# Prints a b c# Prints a   b   c# Prints a     b     c
```

Python String find() Method

Searches the string for a given substring

Usage

The find() method searches for the first occurrence of the specified substring sub and returns its index. If specified substring is not found, it returns -1.

The optional arguments start and end are used to limit the search to a particular portion of the string.

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
```

Syntax

```
string.find(sub,start,end)
```

Basic Examples

```
# Find if substring 'Developer' contains in a string
S = 'Bob is a Developer at ABC'
x = S.find('Developer')
print(x)
# Prints 9
```

`find()` method returns -1 if specified substring is not found in the string.

```
# Find if substring 'Manager' contains in a string
S = 'Bob is a Developer at ABC'
x = S.find('Manager')
print(x)
# Prints -1
```

Limit the find() Search

If you want to search the string from the middle, specify the `start` parameter.

```
# Find 'Big' starting a position 7
S = 'Big, Bigger, Biggest'
x = S.find('Big',7)
print(x)
# Prints 13
```

You can also specify where to stop the search with `end` parameter.

```
# Find 'Big' in between 2 & 10
S = 'Big, Bigger, Biggest'
x = S.find('Big',2,10)
print(x)
# Prints 5
```

find() vs index()

The `find()` method is identical to the index() method.

The only difference is that the `index()` method raises a ValueError exception, if the substring is not found.

```
S = 'Bob is a Developer at ABC'
x = S.find('Manager')
print(x)
# Prints -1
```

```
S = 'Bob is a Developer at ABC'
x = S.index('Manager')
# Triggers ValueError: substring not found
```

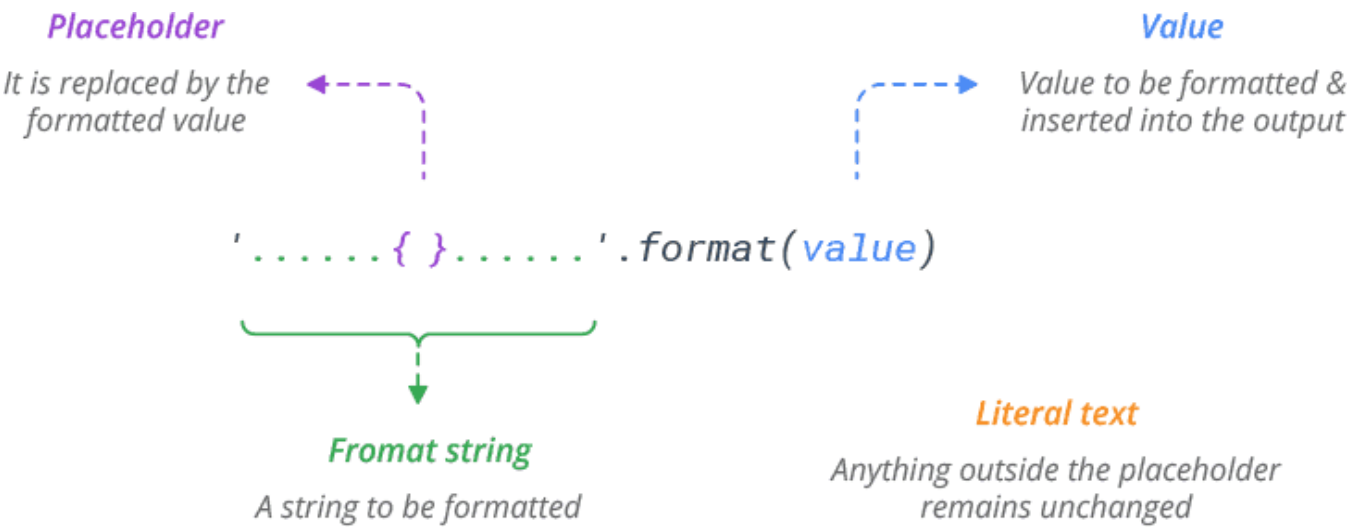
Python String format() Method

Although you can get a lot done with the string methods, Python also provides a more advanced and powerful way to do string processing tasks – string formatting.

It allows you to embed variables inside a string.

format() Method Syntax

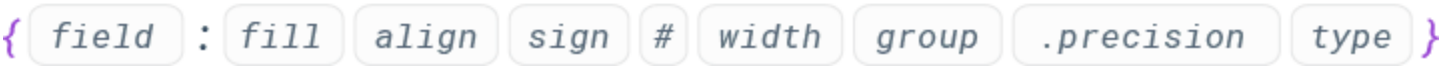
The syntax of format method is:



Python Format Specifier

Each placeholder contains the 'Format specification' to define how the value should be presented.

The general structure of standard format specifier is:



Here are all the formatting options:

Simple Formatting

You can use format() method to do simple positional formatting. Just add a pair of curly braces where you want to substitute the value.

```
# Access arguments by default order (implicit)
S = '{} is {} years old.'.format('Bob', 25)
print(S)
```

```
# Prints Bob is 25 years old.
```

When you leave placeholders empty, the values are replaced in order.

Or, you can give placeholders an explicit positional index and use them in any order you want.

```
# Access arguments by positional index (explicit)
S = '{1} is {0} years old.'.format(25, 'Bob')
print(S)

# Prints Bob is 25 years old.
```

Or, you can refer to your variable substitutions by name.

```
# Access arguments by name
S = '{name} is {age} years old.'.format(name='Bob', age=25)
print(S)

# Prints Bob is 25 years old.
```

Padding and Aligning Strings

You can pad or create space around a value by increasing field size. You can also force the field to be left, right or center-aligned within the available space.

The various alignment options are as follows:

Here are some examples:

```
# Align text left
S = '{:<12}'.format('Left')
print(S)

# Prints Left
```

```
# Align text right
S = '{:>12}'.format('Right')
print(S)

# Prints      Right

# Align text center
S = '{:^12}'.format('Center')
print(S)

# Prints   Center
```

By default, the values are padded with whitespace. You can modify that by specifying a fill character.

```
# Choose custom fill character
S = '{:*^12}'.format('Center')
print(S)

# Prints ***Center***
```

Truncate Long Strings

You can truncate long strings by specifying a `.precision` option.

```
# Truncate string to two characters
S = '{:.2}'.format('Python')
print(S)

# Prints Py

# Add padding to a truncated string and align it center
S = '{:^10.2}'.format('Python')
print(S)
```

```
# Prints    Py
```

Format Numbers (Integer)

Python provides various type codes to format integers.

Here are some examples:

```
# Convert 42 to hex, octal, binary and unicode character
S = 'int:{0:d}, hex:{0:x}, oct:{0:o}, bin:{0:b}, char:{0:c}'.format(42)
print(S)

# Prints int:42, hex:2a, oct:52, bin:101010, char:*
```

If you want to add the prefix '0b', '0o', or '0x' to the output value, specify `#` format option.

```
# Add a prefix to Hex, Octal and Binary
S = 'hex:{0:#x}; oct:{0:#o}; bin:{0:#b}'.format(42)
print(S)

# Prints hex:0x2a; oct:0o52; bin:0b101010
```

Format Numbers (Floating Points and Decimals)

For floating point numbers, you can achieve a variety of additional formatting effects by specifying left justification, zero padding, numeric signs, total field width, and digits after the decimal point.

Following type codes are used to format floating points and decimal values.

```
# Show floating point number
```

```
S = '{:f}'.format(3.141592653)
print(S)

# Prints 3.141593
```

Floating point numbers are rounded to 6 decimal digits by default.

If you want to limit the number of digits after the decimal point, specify precision option.

```
# Specify digits after the decimal point (Precision)
S = '{:.2f}'.format(3.141592653)
print(S)

# Prints 3.14
```

To display numbers in scientific (exponential) notation, use type code ‘e’ or ‘E’ (for uppercase letter)

```
# Display numbers with exponent notation
S = '{:.2e}'.format(3141592653)
print(S)

# Prints 3.14e+09
```

You can format numbers as percentages using the type code `%`. It multiplies the number by 100 and displays in floating-point `'f'` format, followed by a percent sign.

```
# Format number as percentage
S = '{:.2%}'.format(19.5/22)
print(S)

# Prints 88.64%
```

Format Signed Numbers

Only negative numbers are prefixed with a sign by default. You can change this by specifying the `sign` format option.

The sign format option is only valid for number types, and can be one of the following:

Here are some examples:

```
# Display sign for both positive and negative numbers
S = '{:+.2f}, {:+.2f}'.format(3.14, -3.14)
print(S)

# Prints +3.14, -3.14

# Display sign only for negative numbers
S = '{:-.2f}, {:-.2f}'.format(3.14, -3.14)
print(S)

# Prints 3.14, -3.14
```

By default negative numbers are prefixed with a sign, so {:-f} is same as {:f}

When you use `' '` (space) for sign option, it displays a leading space for positive numbers and a minus sign for negative numbers.

```
# Display a space for positive numbers
S = '{: .2f}, {: .2f}'.format(3.14, -3.14)
print(S)

# Prints  3.14, -3.14
```

Padding Numbers (Intergers & Floats)

Similar to strings, you can pad or create space around a number by increasing field width.

```
# Add padding to a number
S = '{:5d}'.format(42)
print(S)

# Prints    42
```

You can add leading zeros to a number by specifying `'0'` as fill character.

```
# Padding zeros to a number
S = '{:0>3d}'.format(7)
print(S)

# Prints 007
```

For floating points, the padding value represents the length of the complete output (including decimal point & decimal digits).

```
# Padding zeros to a floating point
S = '{:06.2f}'.format(3.141592653589793)
print(S)

# Prints 003.14
```

You can control the position of the sign relative to the padding using `=` alignment option.

```
# Padding zeros to a negative number
S = '{:0=8d}'.format(-120)
print(S)

# Prints -0000120
```

Thousands Separator and Nibble Separator

The group format option ',', '_' can be used as a thousands separator.

```
# Using the comma as a thousands separator
S = '{:,}'.format(1234567890)
print(S)

# Prints 1,234,567,890
```

For Hex, Octal and Binary numbers, underscore can be used to separate nibbles(every 4 digits).

```
# Using underscore as a nibble separator
S = '{:_b}'.format(0b01010101010)
print(S)

# Prints 10_1010_1010
```

Datetime Formatting

Python allows datetime objects to be formatted inline.

```
# Using datetime formatting
import datetime
D = datetime.datetime(2010, 7, 4, 12, 15, 58)
S = '{:%Y-%m-%d %H:%M:%S}'.format(D)
print(S)

# Prints 2010-07-04 12:15:58
```

Parametrized Formats

Python allows all of the format options to be specified dynamically using parametrization.

```
# Parametrized fill, alignment and width
S = '{:{fill}{align}{width}}'.format('center', fill='*', align='^', width='12')
print(S)

# Prints ***center***
```

Python String index() Method

Searches the string for a given substring

Usage

The `index()` method searches for the first occurrence of the specified substring `sub` and returns its index. If specified substring is not found, it raises ValueError exception.

The optional arguments `start` and `end` are used to limit the search to a particular portion of the string.

Syntax

```
string.index(sub,start,end)
```

Basic Examples

```
# Find index of the substring 'Developer'
S = 'Bob is a Developer at ABC'
x = S.index('Developer')
```

```
print(x)
# Prints 9
```

index() method raises ValueError exception, if specified substring is not found in the string.

```
# Find index of the substring 'Manager'
S = 'Bob is a Developer at ABC'
x = S.index('Manager')
print(x)
# Triggers ValueError: substring not found
```

Limit the index() Search

If you want to search the string from the middle, specify the start parameter.

```
# Find 'Big' starting a position 7
S = 'Big, Bigger, Biggest'
x = S.index('Big',7)
print(x)
# Prints 13
```

You can also specify where to stop the index() search with end parameter.

```
# Find 'Big' in between 2 & 10
S = 'Big, Bigger, Biggest'
x = S.index('Big',2,10)
print(x)
# Prints 5
```

index() vs find()

The index() method is identical to the find() method.

The only difference is that the find() method returns -1 (instead of raising a ValueError), if the substring is not found.

```
S = 'Bob is a Developer at ABC'
x = S.index ('Manager')
# Triggers ValueError: substring not found


S = 'Bob is a Developer at ABC'
x = S.find('Manager')
print(x)
# Prints -1
```

Python String isalnum() Method

Determines whether the string contains alphanumeric characters

Usage

The isalnum() method returns TRUE if the string is nonempty and all characters in it are alphanumeric. Otherwise, it returns FALSE.

A character is alphanumeric if it is either a letter [a-z],[A-Z] or a number[0-9].

Syntax

string.isalnum()

Basic Example

```
# Check if all characters in the string are alphanumeric
S = 'abc123'
x = S.isalnum()
print(x)
# Prints True
```

isalnum() on String with Special Character

The `isalnum()` method returns FALSE if at least one character is not alphanumeric.

```
S = 'abc-123'
x = S.isalnum()
print(x)
# Prints False

S = '*abc123?'
x = S.isalnum()
print(x)
# Prints False

# even a space
S = 'abc 123'
x = S.isalnum()
print(x)
# Prints False
```

isalnum() on Empty String

The `isalnum()` method returns FALSE if the string is empty.

```
S = ''
x = S.isalnum()
```

```
print(x)
# Prints False
```

Python String isalpha() Method

Determines whether the string contains alphabetic characters

Usage

The `isalpha()` method returns TRUE if the string is nonempty and all characters in it are alphabetic (a-z or A-Z). Otherwise, it returns FALSE.

Syntax

string.isalpha()

Basic Example

```
# Check if all characters in the string are alphabetic
S = 'abc'
x = S.isalpha()
print(x)
# Prints True
```

String with Number/Special Character

The `isalpha()` method returns FALSE if at least one character is not alphabetic.

```
S = '123'
```

```
x = S.isalpha()
print(x)
# Prints False

S = 'abc123'
x = S.isalpha()
print(x)
# Prints False

# even a space
S = 'abc xyz'
x = S.isalpha()
print(x)
# Prints False
```

isalpha() on Empty String

The `isalpha()` method returns FALSE if the string is empty.

```
S = ''
x = S.isalpha()
print(x)
# Prints False
```

Python String isdecimal() Method

Determines whether the string contains decimal characters

Usage

The `isdecimal()` method returns TRUE if the string is nonempty and all characters in it are decimal characters. Otherwise, it returns FALSE.

Decimal characters are those that can be used to form numbers in base 10 (0-9).

Unicode decimal character such as U+0660 (Arabic-Indic Digit Zero) is also considered as a decimal.

Syntax

string.isdecimal()

Basic Examples

```
# Check if all characters in the string are decimal characters
S = '123'
x = S.isdecimal()
print(x)
# Prints True
```

Below are a few examples where `isdecimal()` method returns false.

```
# floating point number
S = '123.456'
x = S.isdecimal()
print(x)
# Prints False


# number with thousands separator
S = '1,234,567'
x = S.isdecimal()
print(x)
# Prints False
```

```
# empty string
S = ''
x = S.isdecimal()
print(x)
# Prints False
```

isdecimal() on Unicode Decimal Characters

Unicode character such as U+0660 (Arabic-Indic Digit Zero) is also considered as a decimal.

```
S = '\u0660'
x = S.isdigit()
print(x)
# Prints True
```

isdecimal() vs isdigit() vs isnumeric()

Following examples explain the difference between the three methods.

```
# Is 42 a decimal or digit or numeric number?
print('42'.isdecimal())       # Prints True
print('42'.isdigit())         # Prints True
print('42'.isnumeric())       # Prints True
# Is ² (Superscript Two) a decimal or digit or numeric number?
print('\u00b2'.isdecimal())   # Prints False
print('\u00b2'.isdigit())     # Prints True
print('\u00b2'.isnumeric())   # Prints True
# Is ⅓ (Vulgar Fraction One Third) a decimal or digit or numeric number?
print('\u2153'.isdecimal())   # Prints False
print('\u2153'.isdigit())     # Prints False
print('\u2153'.isnumeric())   # Prints True
```

As you can see, the main difference between the three functions is:

- isdecimal() method supports only Decimal Numbers.
- isdigit() method supports Decimals, Subscripts, Superscripts.
- isnumeric() method supports Digits, Vulgar Fractions, Subscripts, Superscripts, Roman Numerals, Currency Numerators.

Python String isdigit() Method

Determines whether the string contains digits

Usage

The `isdigit()` method returns TRUE if the string is nonempty and all characters in it are digits. Otherwise, it returns FALSE.

Unicode characters such as superscript digits ¹ , ² and ³ are also considered as digits.

Syntax

string.isdigit()

Basic Examples

```
# Check if all characters in the string are digits
S = '123'
x = S.isdigit()
print(x)
# Prints True
```

Below are a few examples where `isdigit()` method returns false.

```
# floating point number
S = '123.456'
x = S.isdigit()
print(x)
# Prints False

# number with thousands separator
S = '1,234,567'
x = S.isdigit()
print(x)
# Prints False

# empty string
S = ''
x = S.isdigit()
print(x)
# Prints False
```

isdigit() on Unicode Digit Characters

Unicode character such as superscript digit ² is considered as a digit.

```
S = '10²'
x = S.isdigit()
print(x)
# Prints True
```

Special Unicode characters like circled digits ⑥ are also considered as digits.

```
S = '\u2465' # Special Unicode ⑥
x = S.isdigit()
print(x)
# Prints True
```

isdecimal() vs isdigit() vs isnumeric()

Following examples explain the difference between the three methods.

```
# Is 42 a decimal or digit or numeric number?
print('42'.isdecimal())       # Prints True
print('42'.isdigit())         # Prints True
print('42'.isnumeric())       # Prints True

# Is ² (Superscript Two) a decimal or digit or numeric number?
print('\u00b2'.isdecimal())   # Prints False
print('\u00b2'.isdigit())     # Prints True
print('\u00b2'.isnumeric())   # Prints True

# Is ⅓ (Vulgar Fraction One Third) a decimal or digit or numeric number?
print('\u2153'.isdecimal())   # Prints False
print('\u2153'.isdigit())     # Prints False
print('\u2153'.isnumeric())   # Prints True
```

As you can see, the main difference between the three functions is:

- isdecimal() method supports only Decimal Numbers.
- isdigit() method supports Decimals, Subscripts, Superscripts.
- isnumeric() method supports Digits, Vulgar Fractions, Subscripts, Superscripts, Roman Numerals, Currency Numerators.

Python String isidentifier() Method

Determines whether the string is a valid Python identifier

Usage

The isidentifier() method returns TRUE if the string is a valid identifier according to the language definition, and FALSE otherwise.

A valid identifier can only have alphanumeric characters a-z, A-Z, 0-9 and underscore _ . The first character of an identifier cannot be a digit. Also, identifier should not match a Python keyword (reserved identifier).

Syntax

string.isidentifier()

Examples

```
# Check if string 'totalCount' is a valid identifier
S = 'totalCount'
x = S.isidentifier()
print(x)
# Prints True
```

An identifier can contain an underscore but not a special character.

```
print('total_Count'.isidentifier())
# Prints True

print('total Count'.isidentifier())
# Prints False

print('total-Count'.isidentifier())
# Prints False
```

An identifier can contain a digit, except for the first character.

```
print('123totalCount'.isidentifier())
```

```
# Prints False


print('totalCount123'.isidentifier())
# Prints True
```

What If The String Is a Python Keyword?

Surprisingly, `isidentifier()` returns True for a string that matches a Python keyword, even though it is not a valid identifier.

```
print('class'.isidentifier())
# Prints True
```

To test whether a string matches a Python keyword, use `keyword.iskeyword()`

```
from keyword import iskeyword

print(iskeyword('class'))
# Prints True
```

So, a string is considered a valid identifier
if `.isidentifier()` returns True and `iskeyword()` returns False.

Python String islower() Method

Determines whether string contains lowercase characters

Usage

The `islower()` method return TRUE if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Syntax

string.islower()

Examples

```
# Check if all characters in the string are lowercase
S = 'abcd'
x = S.islower()
print(x)
# Prints True
```

The method returns FALSE, if the string doesn't contain at least one cased character.

```
S = '123$@%'
x = S.islower()
print(x)
# Prints False

S = 'a123$@%'
x = S.islower()
print(x)
# Prints True
```

The method also returns FALSE, if the string contains at least one uppercase alphabet.

```
S = 'abcdE'
x = S.islower()
print(x)
# Prints False
```

Python String isnumeric() Method

Determines whether the string contains numeric characters

Usage

The `isnumeric()` method returns TRUE if the string is nonempty and all characters in it are numeric characters. Otherwise, it returns FALSE.

Numeric characters include digit characters, and all characters that have the Unicode numeric value property.

e.g. ² (U+00b2, Superscript Two), ⅕ (U+2155, Vulgar Fraction One Fifth)

Syntax

string.isnumeric()

Basic Examples

```
# Check if all characters in the string are numeric characters
S = '123'
x = S.isnumeric()
print(x)
# Prints True
```

Below are a few examples where `isnumeric()` method returns false.

```
# floating point number
S = '123.456'
x = S.isnumeric()
print(x)
# Prints False
```

```
# number with thousands separator
S = '1,234,567'
x = S.isnumeric()
print(x)
# Prints False

# empty string
S = ''
x = S.isnumeric()
print(x)
# Prints False
```

isnumeric() on Unicode Numeric Characters

Unicode character such as superscript digit ² is considered as a numeric character.

```
S = '\u00b2'
x = S.isnumeric()
print(x)
# Prints True
```

Unicode character like Vulgar Fraction One Third ⅓ is also considered as a numeric.

```
S = '\u2153'
x = S.isnumeric()
print(x)
# Prints True
```

isdecimal() vs isdigit() vs isnumeric()

Following examples explain the difference between the three methods.

```
# Is 42 a decimal or digit or numeric number?
print('42'.isdecimal())       # Prints True
print('42'.isdigit())         # Prints True
print('42'.isnumeric())       # Prints True

# Is ² (Superscript Two) a decimal or digit or numeric number?
print('\u00b2'.isdecimal())   # Prints False
print('\u00b2'.isdigit())     # Prints True
print('\u00b2'.isnumeric())   # Prints True

# Is ⅓ (Vulgar Fraction One Third) a decimal or digit or numeric number?
print('\u2153'.isdecimal())   # Prints False
print('\u2153'.isdigit())     # Prints False
print('\u2153'.isnumeric())   # Prints True
```

As you can see, the main difference between the three functions is:

- isdecimal() method supports only Decimal Numbers.
- isdigit() method supports Decimals, Subscripts, Superscripts.
- isnumeric() method supports Digits, Vulgar Fractions, Subscripts, Superscripts, Roman Numerals, Currency Numerators.

Python String isprintable() Method

Determines whether string contains printable characters

Usage

The `isprintable()` method returns TRUE if the string is empty or all characters in it are printable. It returns FALSE if the string contains at least one non-printable character.

Carriage return `\r` , line feed `\n` and tab `\t` are examples of nonprintable characters.

A simple space character ' ' (0x20, ASCII space) is considered printable.

Syntax

string.isprintable()

Examples

```
# Check if all characters in the string are printable
S = 'Hello, World!'
x = S.isprintable()
print(x)
# Prints True

# Line feed \n and tab \t are nonprintable characters
S = '\tHello,\nWorld!'
x = S.isprintable()
print(x)
# Prints False

# Empty string is considered printable
S = ''
x = S.isprintable()
print(x)
# Prints True
```

Python String isspace() Method

Determines whether the string contains only whitespace characters

Usage

The `isspace()` method returns TRUE if the string is nonempty and all characters in it are whitespace characters. Otherwise, it returns FALSE.

Syntax

```
string.isspace()
```

Basic Example

```
# Check if the string contains only whitespace characters


S = '    '
x = S.isspace()
print(x)
# Prints True


S = '   a'
x = S.isspace()
print(x)
# Prints False
```

ASCII Whitespace Characters

The most common whitespace characters are space `' '` , tab `'\t'` , and newline `'\n'`. Carriage Return `'\r'` and ASCII Form Feed `'\f'` are also considered as whitespace characters.

```
S = ' \t \n \r \f '
```

```
x = S.isspace()

print(x)
# Prints True
```

Unicode Whitespace Characters

Some Unicode characters qualify as whitespace.

```
S = '\u2005 \u2007'

x = S.isspace()

print(x)
# Prints True
```

Python String istitle() Method

Determines whether the string is a titlecased string

Usage

The `istitle()` method returns TRUE if the string is nonempty and a titlecased string. Otherwise, it returns FALSE.

Numbers and special characters are ignored.

In titlecased string each word starts with an uppercase character and the remaining characters are lowercase.

Syntax

string.istitle()

Examples

```
# Check if the string is a titlecased string


# titlecase
S = 'Hello World'
print(S.istitle())
# Prints True


# numbers and characters are ignored
S = '*** Hello, World! 123'
print(S.istitle())
# Prints True
```

Below are a few examples where `istitle()` method returns false.

```
# uppercase
S = 'HELLO, WORLD!'
print(S.istitle())
# Prints False


# lowercase
S = 'hello, world!'
print(S.istitle())
# Prints False
```

Python String isupper() Method

Determines whether string contains uppercase characters

Usage

The `isupper()` method return TRUE if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Syntax

```
string.isupper()
```

Examples

```
# Check if all characters in the string are uppercase
S = 'ABCD'
x = S.isupper()
print(x)
# Prints True
```

The method returns FALSE, if the string doesn't contain at least one cased character.

```
S = '123$@%'
x = S.isupper()
print(x)
# Prints False


S = 'A123$@%'
x = S.isupper()
print(x)
# Prints True
```

The method also returns FALSE, if the string contains at least one lowercase alphabet.

```
S = 'ABCDe'
```

```
x = S.isupper()

print(x)
# Prints False
```

Python String join() Method

Joins all items in an iterable into a single string

Usage

The `join()` method joins all items in an iterable into a single string. Call this method on a string you want to use as a delimiter like comma, space etc.

If there are any non-string values in iterable, a TypeError will be raised.

Syntax

```
string.join(iterable)
```

Parameter	Condition	Description
iterable	Required	Any iterable (like list, tuple, dictionary etc.) whose items are strings

Return Value

The method returns the string obtained by concatenating the items of an iterable.

Basic Examples

```
# Join all items in a list with comma
L = ['red', 'green', 'blue']
x = ','.join(L)
print(x)
# Prints red,green,blue
# Join list items with space
L = ['The', 'World', 'is', 'Beautiful']
x = ' '.join(L)
print(x)
# Prints The World is Beautiful
# Join list items with newline
L = ['First Line', 'Second Line']
x = '\n'.join(L)
print(x)
# First Line
# Second Line
```

A delimiter can contain multiple characters.

```
L = ['the beginning', 'the end', 'the beginning']
x = ' is '.join(L)
print(x)
# Prints the beginning is the end is the beginning
```

join() on Iterable of Size 1

`join()` method is smart enough to insert the delimiter in between the strings rather than just adding at the end of every string. So, if you pass an iterable of size 1, you won't see the delimiter.

```
L = ['red']
x = ','.join(L)
print(x)
# Prints red
```

Join a List of Integers

If there are any non-string values in iterable, a `TypeError` will be raised.

```
L = [1, 2, 3, 4, 5, 6]
x = ','.join(L)
print(x)
# Triggers TypeError: sequence item 0: expected string, int found
```

To avoid such exception, you need to convert each item in a list to string. The list comprehension makes this especially convenient.

```
L = [1, 2, 3, 4, 5, 6]
x = ','.join(str(val) for val in L)
print(x)
# Prints 1,2,3,4,5,6
```

join() on Dictionary

When you use a dictionary as an iterable, all dictionary keys are joined by default.

```
L = {'name':'Bob', 'city':'seattle'}
x = ','.join(L)
print(x)
# Prints city,name
```

To join all values, call values() method on dictionary and pass it as an iterable.

```
L = {'name':'Bob', 'city':'seattle'}
x = ','.join(L.values())
```

```
print(x)
# Prints seattle,Bob
```

To join all keys and values, use `join()` method with list comprehension.

```
L = {'name':'Bob', 'city':'seattle'}
x = ','.join('='.join((key,val)) for (key,val) in L.items())
print(x)
# Prints city=seattle,name=Bob
```

join() vs Concatenation operator +

Concatenation operator `+` is perfectly fine solution to join two strings. But if you need to join more strings, it is convenient to use `join()` method.

```
# concatenation operator
x = 'aaa' + 'bbb'
print(x)
# Prints aaabbb


# join() method
x = ''.join(['aaa','bbb'])
print(x)
# Prints aaabbb
```

Python String ljust() Method

Returns left justified string

Usage

The `ljust()` method returns left-justified string of length width. Padding is done using the specified fillchar (default is an ASCII space).

The original string is returned as it is, if width is less than or equal to string length.

Syntax

string.ljust(width,fillchar)

Basic Example

```
# Align text left
S = 'Left'
x = S.ljust(12)
print(x)
# Prints Left
```

Specify a Fill Character

By default the string is padded with whitespace (ASCII space). You can modify that by specifying a fill character.

```
# * as a fill character
S = 'Left'
x = S.ljust(12, '*')
print(x)
# Prints Left********
```

Equivalent Method

You can achieve the same result by using format() method.

```
S = 'Left'
```

```
x = '{:<12}'.format(S)

print(x)
# Prints Left
```

Python String lower() Method

Converts all characters in a string to lowercase

Usage

The `lower()` method returns a copy of the string with all the characters converted to lowercase. This method does not change the original string.

Syntax

```
string.lower()
```

Examples

```
# Convert all characters to lowercase
S = 'Hello, World!'
x = S.lower()
print(x)
# Prints hello, world!
```

`lower()` method ignores numbers and special characters in a string.

```
S = '123 ABC $@%'
x = S.lower()
print(x)
# Prints 123 abc $@%
```

Python String lstrip() Method

Strips characters from the left end of a string

Usage

The `lstrip()` method removes whitespace from the beginning (leading) of the string by default.

By adding chars parameter, you can also specify the characters you want to strip.

Syntax

```
string.lstrip(chars)
```

Return Value

The method return a copy of the string with the specified characters removed from the beginning of the string.

Strip Whitespace

By default, the method removes leading whitespace.

```
S = '   Hello, World!   '
x = S.lstrip()
print(x)

# Prints Hello, World!
```

Newline '\n', tab '\t' and carriage return '\r' are also considered as whitespace characters.

```
S = ' \t\n\r Hello, World!    '
x = S.lstrip()
print(x)

# Prints Hello, World!
```

Strip Characters

By adding chars parameter, you can also specify the character you want to strip.

```
# Strip single character 'a'
S = 'aaaaab'
x = S.lstrip('a')
print(x)

# Prints b
```

Strip Multiple Characters

The chars parameter is not a prefix; rather, all combinations of its values are stripped.

In below example, strip() would strip all the characters provided in the argument i.e. 'h', 'w', 't', 'p', ':', '/' and '.'

```
S = 'http://www.example.com'
x = S.lstrip('hwtp:/.')
print(x)

# Prints example.com
```

More About lstrip() Method

Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in chars.

```
S = 'xxxxSxxxxSxxxx'
x = S.lstrip('x')
print(x)

# Prints SxxxxSxxxx
```

Here is another example:

```
S = '... - Version 3.2 Model-32'
x = S.lstrip('.- ')
print(x)

# Prints Version 3.2 Model-32
```

Python String partition() Method

Splits the string into a three-part tuple

Usage

The `partition()` method splits the string at the first occurrence of separator, and returns a tuple containing three items.

- The part before the separator
- The separator itself
- The part after the separator

partition() Vs rpartition()

Unlike partition(), The rpartition() method splits the string at the last occurrence of separator.

Otherwise, both methods work exactly the same.

Syntax

string.partition(separator)

Basic Example

```
# Split the string on 'and'
S = 'Do it now and keep it simple'
x = S.partition('and')
print(x)
# Prints ('Do it now ', 'and', ' keep it simple')
```

No Match Found

If the separator is not found, the method returns a tuple containing the string itself, followed by two empty strings.

```
S = 'Do it now and keep it simple'
x = S.partition('or')
print(x)
# Prints ('Do it now and keep it simple', '', '')
```

Multiple Matches

If the separator is present multiple times, the method splits the string at the first occurrence.

```
S = 'Do it now and keep it simple'
x = S.partition('it')
print(x)
# Prints ('Do ', 'it', ' now and keep it simple')
```

Python String replace() Method

Replaces occurrences of a substring within a string

Usage

The `replace()` method returns a copy of string with all occurrences of old substring replaced by new.

By default, all occurrences of the substring are removed. However, you can limit the number of replacements by specifying optional parameter count.

Syntax

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

Examples

```
# Replace substring 'World' with 'Universe'
S = 'Hello, World!'
x = S.replace('World','Universe')
print(x)
# Prints Hello, Universe!
```

By default, the method replaces all occurrences of the specified substring.

```
# Replace all occurrence of the substring 'Long'
S = 'Long, Longer, Longest'
x = S.replace('Long','Small')
print(x)
# Prints Small, Smaller, Smallest
```

If the optional argument count is specified, only the first count occurrences are replaced.

```
# Replace first two occurrence of the substring 'Long'
S = 'Long, Longer, Longest'
x = S.replace('Long','Small', 2)
print(x)
# Prints Small, Smaller, Longest
```

Python String rfind() Method

Searches the string for a given substring, starting from the right

Usage

The `rfind()` method searches for the last occurrence of the specified substring sub and returns its index. If specified substring is not found, it returns -1.

The optional arguments start and end are used to limit the search to a particular portion of the string.

Syntax

string.rfind(sub,start,end)

Basic Examples

```
# Find last occurrence of the substring 'Big'
S = 'Big, Bigger, Biggest'
x = S.rfind('Big')
print(x)
# Prints 13
```

`rfind()` method returns -1 if the specified substring doesn't exist in the string.

```
S = 'Big, Bigger, Biggest'
x = S.rfind('Small')
print(x)
# Prints -1
```

Limit the rfind() Search

If you want to search the string from the middle, specify the start and end parameters.

```
# Search the string from position 2 to 10
S = 'Big, Bigger, Biggest'
x = S.rfind('Big',2,10)
print(x)
# Prints 5
```

rfind() vs rindex()

The `rfind()` method is identical to the rindex() method. The only difference is that the `rindex()` method raises a ValueError exception, if the substring is not found.

```
S = 'Big, Bigger, Biggest'
x = S.rfind('Small')
print(x)
# Prints -1
```

```
S = 'Big, Bigger, Biggest'
x = S.rindex('Small')
print(x)
# Triggers ValueError: substring not found
```

Python String rindex() Method

Searches the string for a given substring, starting from the right

Usage

The `rindex()` method searches for the last occurrence of the specified substring `sub` and returns its index. If specified substring is not found, it raises ValueError exception.

The optional arguments `start` and `end` are used to limit the search to a particular portion of the string.

Syntax

```
string.rindex(sub,start,end)
```

Basic Examples

```
# Find the index of last occurrence of the substring 'Big'
```

```
S = 'Big, Bigger, Biggest'
x = S.rindex('Big')
print(x)
# Prints 13
```

`rindex()` method raises ValueError exception, if specified substring is not found in the string.

```
S = 'Big, Bigger, Biggest'
x = S.rindex('Small')
print(x)
# Triggers ValueError: substring not found
```

Limit the rindex() Search

If you want to search the string from the middle, specify the start and end parameter.

```
# Search the string from position 2 to 10
S = 'Big, Bigger, Biggest'
x = S.rindex('Big',2,10)
print(x)
# Prints 5
```

rindex() vs rfind()

The `rindex()` method is identical to the rfind() method. The only difference is that the `rfind()` method returns -1 (instead of raising a ValueError), if the substring is not found.

```
# rfind()
S = 'Big, Bigger, Biggest'
x = S.rfind('Small')
print(x)
```

```
# Prints -1

# rindex()

S = 'Big, Bigger, Biggest'

x = S.rindex('Small')

print(x)
# Triggers ValueError: substring not found
```

Python String rjust() Method

Returns right justified string

Usage

The `rjust()` method returns right-justified string of length width. Padding is done using the specified fillchar (default is an ASCII space).

The original string is returned as it is, if width is less than or equal to string length.

Syntax

```
string.rjust(width,fillchar)
```

Basic Example

```
# Align text right

S = 'Right'

x = S.rjust(12)

print(x)
# Prints      Right
```

Specify a Fill Character

By default the string is padded with whitespace (ASCII space). You can modify that by specifying a fill character.

```
# * as a fill character
S = 'Right'
x = S.rjust(12, '*')
print(x)
# Prints *******Right
```

Equivalent Method

You can achieve the same result by using format() method.

```
S = 'Right'
x = '{:>12}'.format(S)
print(x)
# Prints       Right
```

Python String rpartition() Method

Splits the string into a three-part tuple

Usage

The `rpartition()` method splits the string at the last occurrence of separator, and returns a tuple containing three items.

- The part before the separator
- The separator itself
- The part after the separator

rpartition() Vs partition()

Unlike rpartition(), The partition() method splits the string at the first occurrence of separator. Otherwise, both methods work exactly the same.

Syntax

string.rpartition(separator)

Basic Example

```
# Split the string on 'and'
S = 'Do it now and keep it simple'
x = S.rpartition('and')
print(x)
# Prints ('Do it now ', 'and', ' keep it simple')
```

No Match Found

If the separator is not found, the method returns a tuple containing two empty strings, followed by the string itself.

```
S = 'Do it now and keep it simple'
x = S.rpartition('or')
print(x)
# Prints ('', '', 'Do it now and keep it simple')
```

Multiple Matches

If the separator is present multiple times, the method splits the string at the last occurrence.

```
S = 'Do it now and keep it simple'
x = S.rpartition('it')
print(x)
# Prints ('Do it now and keep ', 'it', ' simple')
```

Python String rsplit() Method

Splits a string into a list of substrings, starting from the right

Usage

The `rsplit()` method splits the string on a specified delimiter and returns the list of substrings.

When you specify maxsplit, only the given number of splits will be made.

Syntax

```
string.rsplit(delimiter,maxsplit)
```

Split on Whitespace

When delimiter is not specified, the string is split on whitespace.

```
S = 'The World is Beautiful'
x = S.rsplit()
print(x)
# Prints ['The', 'World', 'is', 'Beautiful']
```

Split on a Delimiter

You can split a string by specifying a delimiter.

```
# Split on comma
S = 'red,green,blue'
x = S.rsplit(',')
print(x)
# Prints ['red', 'green', 'blue']
# Delimiter with multiple characters
S = 'the beginning is the end is the beginning'
x = S.rsplit(' is ')
print(x)
# Prints ['the beginning', 'the end', 'the beginning']
```

Limit Splits With Maxsplit

When you specify maxsplit, only the given number of splits will be made, starting from the right. The resulting list will have the specified number of elements plus one.

```
S = 'The World is Beautiful'
x = S.rsplit(None,1)
print(x)
# Prints ['The World is', 'Beautiful']


S = 'The World is Beautiful'
x = S.rsplit(None,2)
print(x)
# Prints ['The World', 'is', 'Beautiful']
```

rsplit() vs split()

If maxsplit is specified, rsplit() counts splits from the right end, whereas split() counts them from left. Otherwise, they both behave exactly the same.

```
# rsplit()
S = 'The World is Beautiful'
x = S.rsplit(None,1)
print(x)
# Prints ['The World is', 'Beautiful']


# split()
S = 'The World is Beautiful'
x = S.split(None,1)
print(x)
# Prints ['The', 'World is Beautiful']
```

Python String rstrip() Method

Strips characters from the right end of a string

Usage

The strip() method removes whitespace from the right end (trailing) of the string by default.

By adding chars parameter, you can also specify the characters you want to strip.

Syntax

string.rstrip(chars)

Return Value

The method return a copy of the string with the specified characters removed from the right end of a string.

Strip Whitespace

By default, the method removes trailing whitespace.

```
S = '   Hello, World!   '
x = S.rstrip()
print(x)

# Prints    Hello, World!
```

Newline '\n', tab '\t' and carriage return '\r' are also considered as whitespace characters.

```
S = '    Hello, World! \t\n\r '
x = S.rstrip()
print(x)

# Prints    Hello, World!
```

Strip Characters

By adding chars parameter, you can also specify the character you want to strip.

```
# Strip single character 'a'
S = 'baaaaa'
x = S.rstrip('a')
print(x)

# Prints b
```

Strip Multiple Characters

The chars parameter is not a suffix; rather, all combinations of its values are stripped.

In below example, strip() would strip all the characters provided in the argument i.e. 'w', 'o' and '/'

```
S = 'example.com/wow'
x = S.rstrip('wo/')
print(x)

# Prints example.com
```

More About rstrip() Method

Characters are removed from the trailing end until reaching a string character that is not contained in the set of characters in chars.

```
S = 'xxxxSxxxxSxxxx'
x = S.rstrip('x')
print(x)

# Prints xxxxSxxxxS
```

Here is another example:

```
S = 'Version 3.2 Model-32 - ...'
x = S.rstrip('.- ')
print(x)

# Prints Version 3.2 Model-32
```

Python String split() Method

Splits a string into a list of substrings

Usage

The `split()` method splits the string on a specified delimiter and returns the list of substrings. When delimiter is not specified, the string is split on whitespace.

By default, `split()` will make all possible splits (there is no limit on the number of splits). When you specify maxsplit, however, only the given number of splits will be made.

Syntax

```
string.split(delimiter,maxsplit)
```

Split on Whitespace

When delimiter is not specified, the string is split on whitespace.

```
S = 'The World is Beautiful'
x = S.split()
print(x)
# Prints ['The', 'World', 'is', 'Beautiful']
```

Another feature of the bare call to `split()` is that it automatically combines consecutive whitespace into single delimiter, and splits the string.

```
S = '  The    World is  Beautiful'
x = S.split()
print(x)
# Prints ['The', 'World', 'is', 'Beautiful']
```

Newline '\n', tab '\t' and carriage return '\r' are also considered as whitespace characters.

```
S = 'The\n\rWorld\tis Beautiful'

x = S.split()

print(x)
# Prints ['The', 'World', 'is', 'Beautiful']
```

Split on a Delimiter

You can split a string by specifying a delimiter.

```
# Split on comma
S = 'red,green,blue'

x = S.split(',')

print(x)
# Prints ['red', 'green', 'blue']

# Split on new line
S = 'First Line\nSecond Line'

x = S.split('\n')

print(x)
# Prints ['First Line', 'Second Line']
```

A delimiter can contain multiple characters.

```
S = 'the beginning is the end is the beginning'

x = S.split(' is ')

print(x)
# Prints ['the beginning', 'the end', 'the beginning']
```

Limit Splits With Maxsplit

When you specify maxsplit, only the given number of splits will be made. The resulting list will have the specified number of elements plus one.

```
S = 'The World is Beautiful'
x = S.split(None,1)
print(x)
# Prints ['The', 'World is Beautiful']


S = 'The World is Beautiful'
x = S.split(None,2)
print(x)
# Prints ['The', 'World', 'is Beautiful']
```

If maxsplit is not specified or -1, `split()` will make all possible splits (there is no limit on the number of splits).

```
S = 'The World is Beautiful'
x = S.split(None,-1)
print(x)
# Prints ['The', 'World', 'is', 'Beautiful']


S = 'The World is Beautiful'
x = S.split()
print(x)
# Prints ['The', 'World', 'is', 'Beautiful']
```

Split on Multiple Delimiters

The `split()` method does not allow for multiple delimiters. You can use the re.split() method (based on regular expression) instead.

```
# Split with comma ( , ) semicolon ( ; ) and colon ( : )
S = 'red,green;blue:yellow'
import re
x = re.split('[,;:]',S)
print(x)
```

```
# Prints ['red', 'green', 'blue', 'yellow']
```

split() vs rsplit()

If maxsplit is specified, `split()` counts splits from the left end, whereas rsplit() counts them from right. Otherwise, they both behave exactly the same.

```
# split()
S = 'The World is Beautiful'
x = S.split(None,1)
print(x)
# Prints ['The', 'World is Beautiful']


# rsplit()
S = 'The World is Beautiful'
x = S.rsplit(None,1)
print(x)
# Prints ['The World is', 'Beautiful']
```

Unpacking, Indexing and Slicing

As `split()` method returns a list of substrings, you can perform any operation that a list supports. Like multiple assignment(unpacking), indexing, slicing etc.

```
# multiple assignment
S = 'red,green,blue'
x,y,z = S.split(',')
print(x)
# Prints red
print(y)
# Prints green
print(z)
```

```
# Prints blue


# indexing
S = 'red,green,blue,yellow'
x = S.split(',')[2]
print(x)
# Prints blue


# slicing
S = 'red,green,blue,yellow'
x = S.split(',')[1:3]
print(x)
# Prints ['green', 'blue']
```

Python String splitlines() Method

Splits a string at line breaks

Usage

The `splitlines()` method splits a string at line breaks and returns them in a list.

If the optional keepends argument is specified and TRUE, line breaks are included in the resulting list.

Syntax

```
string.splitlines(keepends)
```

Basic Example

```
# Split a string at '\n' into a list
S = 'First line\nSecond line'
x = S.splitlines()
print(x)
# Prints ['First line', 'Second line']
```

Different Line breaks

Newline `\n`, carriage return `\r` and form feed `\f` are common examples of line breaks.

```
S = 'First\nSecond\r\nThird\fFourth'
x = S.splitlines()
print(x)
# Prints ['First', 'Second', 'Third', 'Fourth']
```

Keep Line Breaks in Result

If the optional keepends argument is specified and TRUE, line breaks are included in the resulting list.

```
S = 'First line\nSecond line'
x = S.splitlines(True)
print(x)
# Prints ['First line\n', 'Second line']
```

splitlines() vs split() on Newline

There are mainly two differences:

1. Unlike split(), splitlines() returns an empty list for the empty string.

```
# splitlines()
S = ''
x = S.splitlines()
print(x)
# Prints []

# split()
S = ''
x = S.split('\n')
print(x)
# Prints ['']
```

2. When you use `splitlines()` a terminal line break does not result in an extra line.

```
# splitlines()
S = 'One line\n'
x = S.splitlines()
print(x)
# Prints ['One line']

# split()
S = 'One line\n'
x = S.split('\n')
print(x)
# Prints ['One line', '']
```

Python String startswith() Method

Determines whether the string starts with a given substring

Usage

The `startswith()` method returns True if the string starts with the specified prefix, otherwise returns False.

You can limit the search by specifying optional arguments start and end.

`startswith()` also accepts a tuple of prefixes to look for.

Syntax

```
string.startswith(prefix,start,end)
```

Basic Example

```
# Check if the string starts with 'Bob'
S = 'Bob is a CEO.'
x = S.startswith('Bob')
print(x)
# Prints True
```

Limit startswith() Search to Substring

To limit the search to the substring, specify the start and end parameters.

```
# Check if the substring (9th to 18th character) starts with 'CEO'
S = 'Bob is a CEO at ABC'
x = S.startswith('CEO',9,18)
print(x)
# Prints True
```

Provide Multiple Prefixes to Look for

You can provide multiple prefixes to the method in the form of a tuple. If the string starts with any item of the tuple, the method returns True, otherwise returns False.

```
S = 'Bob is a CEO'
prefixes = ('Bob','Max','Sam')
x = S.startswith(prefixes)
print(x)
# Prints True

S = 'Max is a COO'
prefixes = ('Bob','Max','Sam')
x = S.startswith(prefixes)
print(x)
# Prints True
```

Python String strip() Method

Strips leading and trailing characters

Usage

The `strip()` method removes whitespace from the beginning (leading) and end (trailing) of the string by default.

By adding `chars` parameter, you can also specify the characters you want to strip.

Syntax

string.strip(chars)

Return Value

The method return a copy of the string with the leading and trailing characters removed.

Strip Whitespace

By default, the method removes leading and trailing whitespace.

```
S = '   Hello, World!   '
x = S.strip()
print(x)

# Prints Hello, World!
```

Newline `'\n'`, tab `'\t'` and carriage return `'\r'` are also considered as whitespace characters.

```
S = '  \t Hello, World! \n\r '
x = S.strip()
print(x)

# Prints Hello, World!
```

Strip Characters

By adding chars parameter, you can also specify the character you want to strip.

```
# Strip single character 'a'
S = 'aaabaaaa'
x = S.strip('a')
print(x)

# Prints b
```

Strip Multiple Characters

The chars parameter is not a prefix or suffix; rather, all combinations of its values are stripped.

In below example, strip() would strip all the characters provided in the argument i.e. 'c', 'm', 'o', 'w', 'z' and '.'

```
S = 'www.example.com'
x = S.strip('cmowz.')
print(x)

# Prints example
```

More About strip() Method

Characters are removed from both ends until reaching a string character that is not contained in the set of characters in chars.

```
S = 'xxxxSxxxxSxxxx'
x = S.strip('x')
print(x)

# Prints SxxxxS
```

Here is another example:

```
S = '... - Version 3.2 Model-32 ...'
x = S.strip('.- ')
print(x)

# Prints Version 3.2 Model-32
```

Python String swapcase() Method

Swaps case of all characters in a string

Usage

The `swapcase()` method returns a copy of the string with uppercase characters converted to lowercase and vice versa. This method does not change the original string.

Syntax

```
string.swapcase()
```

Examples

```
# Swap case of all characters in a string
S = 'Hello, World!'
x = S.swapcase()
print(x)
# Prints hELLO, wORLD!
```

`swapcase()` method ignores numbers and special characters in a string.

```
S = '123 abc $@%'
x = S.swapcase()
print(x)
# Prints 123 ABC $@%
```

Python String title() Method

Converts string to “Title Case”

Usage

The `title()` method returns a copy of the string with first letter of each word is converted to uppercase and remaining letters are lowercase.

The method does not change the original string.

Syntax

```
string.title()
```

Basic Example

```
# Convert string to titlecase
S = 'hello, world!'
x = S.title()
print(x)
# Prints Hello, World!
```

Unexpected Behavior of title() Method

The first letter after every number or special character (such as Apostrophe) is converted into a upper case letter.

```
S = "c3po is a droid"
x = S.title()
print(x)
# Prints C3Po Is A Droid


S = "they're bob's friends."
x = S.title()
```

```
print(x)
# Prints They'Re Bob'S Friends.
```

Workaround

As a workaround for this you can use string.capwords()

```
import string
S = "c3po is a droid"
x = string.capwords(S)
print(x)
# Prints C3po Is A Droid


import string
S = "they're bob's friends."
x = string.capwords(S)
print(x)
# Prints They're Bob's Friends.
```

Python String upper() Method

Converts all characters in a string to uppercase

Usage

The `upper()` method returns a copy of the string with all the characters converted to uppercase. This method does not change the original string.

Syntax

string.upper()

Examples

```
# Convert all characters to uppercase
S = 'Hello, World!'
x = S.upper()
print(x)
# Prints HELLO, WORLD!
```

`upper()` method ignores numbers and special characters in a string.

```
S = '123 abc $@%'
x = S.upper()
print(x)
# Prints 123 ABC $@%
```

Python String zfill() Method

Pads a string on the left with zeros

Usage

The `zfill()` method returns a copy of string left padded with '0' characters to make a string of length width.

The original string is returned, if width is less than or equal to string length.

Syntax

```
string.zfill(width)
```

Basic Example

```
# Zero-pad a string until it is 6 characters long
S = '42'
x = S.zfill(6)
print(x)
# Prints 000042
```

String with Sign Prefix

If the string contains a leading sign `+` or `-`, zeros are padded after the sign character rather than before.

```
S = '+42'
x = S.zfill(6)
print(x)
# Prints +00042
S = '-42'
x = S.zfill(6)
print(x)
# Prints -00042
```

Equivalent Method

You can achieve the same result by using format() method.

```
S = '42'
x = '{:0>6}'.format(S)
print(x)
# Prints 000042
```