Strings

A string is an immutable sequence that represent *Unicode* code points.

```
'Single "embedded"'
"Hello 'embedded'"
'''Three single
  quotes'''
"""Hello
World"""
str(8) # Convert to string
```

Formatting Strings

The String Literal

Prefixes:

```
"f"|"F" # Formatted String
        # Literals
"r"|"R" # Raw strings
and any combination of those.

"u"|"U" # Unicode
```

The Bytes Literal

Prefixes:

```
"b"|"B" # Bytes prefix
may be combined with "r"|"R"
```

Formatted String Literals *New in version 3.6.* A formatted string literal is prefixed with `f` or `F` which may contain replacement fields delimited by curly braces {}.

```
> name = "Fred"
> f"He said his name is {name!r}."
"He said his name is 'Fred'."
> f"His name is {repr(name)}."
"His name is 'Fred'."

> from decimal import Decimal
> w = 10
> precision = 4
> v = Decimal("12.34567")
> f"result: {v:{w}.{precision}}"
'result:      12.35'
```

Backslashes are not allowed in format expressions:

```
> f"newline: {ord('\n')}" # raises
    # SyntaxError
> newline = ord('\n')
> f"newline: {newline}"
'newline: 10' # works
```

See also **PEP 498**

Format String with **str.format** or with the **Formatter** class

```
replacement_field ::=
    "{" [field_name]
     ["!" conversion]
     [":" format_spec] "}"}"
```

The *replacement_field* can start with a *field_name* to specify the object whose value is to be formatted.

```
"Bring me a {}".format('beer')
"Weight in tons {0.weight}".
    format(dict_of_weightwatchers)
"Units destroyed: {players[0]}".
    format(list_of_players)
```

The *field_name* may be followed by a *conversion* field

```
!s  # calls str()
!r  # calls repr()
!a  # calls ascii()
```

and a *format_spec* (See below)

Format Specification Mini-Language is used within *replacement_fields* to define how individual values are presented.

```
format_spec ::= [[fill]align]
    [sign][#][0][width]
    [grouping_option][.precision]
    [type]
```

If there is an *align* value, it can be preceded by a *fill* character (default is space).

```
<   # Left alignment
>   # Right alignment
=   # padding before the digits
^   # Center alignment
```

Available integer presentation types:

```
b    # Binary format (Base 2)
c    # Character
d    # Decimal integer (Base 10)
o    # Octal format (Base 8)
x    # Hex format in lower case
X    # Hex format in upper case
n    # Number ('d' but localized)
None # Same as 'd'
```

Floating point and decimal presentation types

```
e    # Exponent notation
E    # Like e but uppercase E
f    # Fixed-point notation (.6)
F    # Like f but nan -> NAN
     #            inf -> INF
g    # General format
G    # Same as g but uppercase E
n    # Like g but localized
%    # Percentage (in f format)
None # Like g but at least one
     # digit after decimal point
```

Methods

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 like descibed in Unicode Standard Section 3.13

More aggresive like **str.lowercase** and converts 'ß' to 'ss'.

str.center() Return centered in a string of length *width* using *fillchar*, if present

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

str.count() Return the number of non-overlapping occurrences of substring *sub* in the range *[stard, end]*. *start* and *end* are interpreted as in **slice** notation.

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

str.encode() Return an encoded version of the string as a bytes object.

```
str.encode(encodig="utf-8"
error="strict")
```

More on **encoding** and **error**.

str.endswith() Return `True` if the string ends with *suffix*, otherwise return `False`. *suffix* can be a **tuple**.

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

str.expandtabs() Replaces all tab characters by one or more spaces, depending on the current column and the given tab size

```
str.expandtabs(tabsize=8)
```

str.find() Return lowest index where substring *sub* is found within the **slice** `s[start:end]` or if *sub* is not found `-1`

```
str.find[sub[,start[,end]]]
```

str.format() Performs a string formatting operation. More about the **Format String Syntax** in the section below.

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

str.format_map() Similar to `str.format(**mapping)` except that `mapping` is used directly.

str.index() Like **find**, but raise **ValueError** when the substring is not found.

str.is. . . () Perform checks about ascii type and classes.

str.join() Concatenate strings in *iterable* with str as separator.

```
str.join(iterable)
```

str.ljust() Left justify **str** in string of length *width* padded with *fillchar*, if present (default is ASCII space).

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

str.lower() Converts all cased characters to lowercase as of **Unicode Standard** Section 3.13

str.lstrip() Removes leading characters *chars* defaults to any whitespace.

```
str.lstrip([chars])
```

***static* str.maketrans()** Returns a translation table usable for **str.translate**

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

str.partition() Splits the string at the first occurrence of *sep* and return a **3-tuple**. containig the part before *sep*, *sep* and the part after *sep* or if *sep* is not found the string followed by two empty strings.

```
str.partition(sep)
```

str.replace() Replaces all occurrences of substring *old* replaced by *new*, *count* times if present.

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

str.rfind() Returns the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`, or return `-1` on failure.

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

str.rindex() Like **rfind** but raises **ValueError** when *sub* is not found

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

str.rjust() Right justify in a string of length *width* padded with *fillchar* if present (default is ASCII space)

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

str.rpartition() Splits the string at the last occurrence of *sep* and return a **3-tuple**. containig the part before *sep*, *sep* and the part after *sep* or if *sep* is not found two empty strings followed by **str**.

```
str.rpartition(sep)
```

str.rsplit() Return a list of words, using *sep* as delimiter at most *maxsplit* times if given splitting from the right. *sep* defaults to any whitespace character.

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

str.rstrip() Removes trailing characters *chars*, which default to any whitespace.

```
str.rstrip([chars])
```

str.split() Like str.rsplit() but from splitting from the left. Splits around whitespace if *sep* is `None` or missing regarding consecutive whitespace characeters as one.

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

str.splitlines() Breaks a string around **line boundaries** and returns a list of lines and if *keepends* is given including the line breaks.

```
str.splitlines([keepends])
```

str.startswith() `True` if string starts with the *prefix* if given starting at *start* and stopping at *end* position. *prefix* can also be a tuple.

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

str.strip() like str.rstrip() but removes leading **and** trailing whitespace or *chars* if present.

```
str.strip([chars])
```

str.swapcase() Converts uppercase characters to lowercase and vice versa. It is not necessarily true that `s.swapcase().swapcase() == s`. str.title()
Words start with an Uppercase character and remaining characters are lowercase.

str.translate() Maps each character through a given translation table.

```
str.translate(table)
```

str.upper() Converts all **cased characters** to uppercase like it is described in the **Unicode Standard**.

str.zfill() Fills the string from the left with ASCII `0` digits to make a string of length *width* after a possible leading sign prefix `+`/`-`.

```
str.zfill(width)
```