

6.1. `string` — Common string operations

Source code: [Lib/string.py](#)

6.1.1. String constants

The constants defined in this module are:

`string.ascii_letters`

The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.ascii_lowercase`

The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

`string.ascii_uppercase`

The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

`string.digits`

The string `'0123456789'`.

`string.hexdigits`

The string `'0123456789abcdefABCDEF'`.

`string.octdigits`

The string `'01234567'`.

`string.punctuation`

String of ASCII characters which are considered punctuation characters in the `C` locale.

`string.printable`

String of ASCII characters which are considered printable. This is a combination of `digits`, `ascii_letters`, `punctuation`, and `whitespace`.

`string.whitespace`

A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

6.1.2. String Formatting

The built-in string class provides the ability to do complex variable substitutions and value formatting via the `format()` method described in [PEP 3101](#). The `Formatter` class in the `string` module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in `format()` method.

`class string.Formatter`

The `Formatter` class has the following public methods:

`format(format_string, *args, **kwargs)`

`format()` is the primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. `format()` is just a wrapper that calls `vformat()`.

`vformat(format_string, args, kwargs)`

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the `*args` and `**kwargs` syntax. `vformat()` does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the `Formatter` defines a number of methods that are intended to be replaced by subclasses:

`parse(format_string)`

Loop over the `format_string` and return an iterable of tuples (*literal_text*, *field_name*, *format_spec*, *conversion*). This is used by `vformat()` to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be `None`.

`get_field(field_name, args, kwargs)`

Given *field_name* as returned by `parse()` (see above), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form defined in [PEP 3101](#), such as “0[name]” or “label.title”. *args* and *kwargs* are as passed in to `vformat()`. The return value *used_key* has the same meaning as the *key* parameter to `get_value()`.

get_value(key, args, kwargs)

Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to `vformat()`, and the *kwargs* parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression '0.name' would cause `get_value()` to be called with a *key* argument of 0. The `name` attribute will be looked up after `get_value()` returns by calling the built-in `getattr()` function.

If the index or keyword refers to an item that does not exist, then an `IndexError` or `KeyError` should be raised.

check_unused_args(used_args, args, kwargs)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the *args* and *kwargs* that was passed to `vformat`. The set of unused args can be calculated from these parameters. `check_unused_args()` is assumed to raise an exception if the check fails.

format_field(value, format_spec)

`format_field()` simply calls the global `format()` built-in. The method is provided so that subclasses can override it.

convert_field(value, conversion)

Converts the value (returned by `get_field()`) given a conversion type (as in the tuple returned by the `parse()` method). The default version understands 's' (str), 'r' (repr) and 'a' (ascii) conversion types.

6.1.3. Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax).

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{}` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::= arg_name ("." attribute_name | "[" element_index "]") *
arg_name          ::= [identifier | integer]
attribute_name    ::= identifier
element_index     ::= integer | index_string
index_string      ::= <any source character except "]"> +
conversion        ::= "r" | "s" | "a"
format_spec       ::= <described in the next section>
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `!`, and a *format_spec*, which is preceded by a colon `:`. These specify a non-default format for the replacement value.

See also the [Format Specification Mini-Language](#) section.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. If the numerical *arg_names* in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `'[:-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using `getattr()`, while an expression of the form `'[index]'` does an index lookup using `__getitem__()`.

Changed in version 3.1: The positional argument specifiers can be omitted, so `'{} {}'` is equivalent to `'{0} {1}'`.

Some simple format string examples:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                  # Implicitly references the first positional
argument
"From {} to {}"                  # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument 'name'
"Weight in tons {0.weight}"      # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the `__format__()` method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling `__format__()`, the normal formatting logic is bypassed.

Three conversion flags are currently supported: `'!s'` which calls `str()` on the value, `'!r'` which calls `repr()` and `'!a'` which calls `ascii()`.

Some examples:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
"More {!a}"                     # Calls ascii() on the argument first
```

The *format_spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the *format_spec*.

Most built-in types support a common formatting mini-language, which is described in the next section.

A *format_spec* field can also include nested replacement fields within it. These nested replacement fields can contain only a field name; conversion flags and format specifications are not allowed. The replacement fields within the *format_spec* are substituted before the *format_spec* string is interpreted. This allows the formatting of a value to be dynamically specified.

See the [Format examples](#) section for some examples.

6.1.3.1. Format Specification Mini-Language

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see [Format String Syntax](#)). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string ("") produces the same result as if you had called `str()` on the value. A non-empty format string typically modifies the result.

The general form of a *standard format specifier* is:

```
format_spec ::= [[fill]align][sign][#][0][width][,][.precision][type]  
fill         ::= <any character>  
align        ::= "<" | ">" | "=" | "^"  
sign         ::= "+" | "-" | " "  
width        ::= integer  
precision    ::= integer  
type         ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" |  
"o" | "s" | "x" | "X" | "%"
```

If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. Note that it is not possible to use `{` and `}` as *fill* char while using the `str.format()` method; this limitation however doesn't affect the `format()` function.

The meaning of the various alignment options is as follows:

Option	Meaning
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types.
'^'	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

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

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The '#' option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or '0x' to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

The `' , '` option signals the use of a comma for a thousands separator. For a locale aware separator, use the `' n '` integer presentation type instead.

Changed in version 3.1: Added the `' , '` option (see also [PEP 378](#)).

width is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

Preceding the *width* field by a zero (`' 0 '`) character enables sign-aware zero-padding for numeric types. This is equivalent to a *fill* character of `' 0 '` with an *alignment* type of `' = '`.

The *precision* is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with `' f '` and `' F '`, or before and after the decimal point for a floating point value formatted with `' g '` or `' G '`. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer values.

Finally, the *type* determines how the data should be presented.

The available string presentation types are:

Type	Meaning
<code>' s '</code>	String format. This is the default type for strings and may be omitted.
None	The same as <code>' s '</code> .

The available integer presentation types are:

Type	Meaning
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as 'd'.

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

Type	Meaning
'e'	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
'E'	Exponent notation. Same as 'e' except it uses an upper case 'E' as the separator character.
'f'	Fixed point. Displays the number as a fixed-point number. The default precision is 6.

Type	Meaning
'F'	Fixed point. Same as 'f', but converts <code>nan</code> to <code>NAN</code> and <code>inf</code> to <code>INF</code> .
'g'	<p>General format. For a given precision <code>p</code> ≥ 1, this rounds the number to <code>p</code> significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude.</p> <p>The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision <code>p-1</code> would have exponent <code>exp</code>. Then if $-4 \leq \text{exp} < p$, the number is formatted with presentation type 'f' and precision <code>p-1-exp</code>. Otherwise, the number is formatted with presentation type 'e' and precision <code>p-1</code>. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it.</p> <p>Positive and negative infinity, positive and negative zero, and nans, are formatted as <code>inf</code>, <code>-inf</code>, <code>0</code>, <code>-0</code> and <code>nan</code> respectively, regardless of the precision.</p> <p>A precision of <code>0</code> is treated as equivalent to a precision of <code>1</code>. The default precision is <code>6</code>.</p>
'G'	General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.
'n'	Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters.
'%'	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.
None	Similar to 'g', except with at least one digit past the decimal point and a default precision of 12. This is intended to match <code>str()</code> , except you can add the other format modifiers.

6.1.3.2. Format examples

This section contains examples of the new format syntax and comparison with the old %-formatting.

In most of the cases the syntax is similar to the old %-formatting, with the addition of the `{}` and with `:` used instead of `%`. For example, `'%03.2f'` can be translated to `'{:03.2f}'`.

The new format syntax also supports new and different options, shown in the follow examples.

Accessing arguments by position:

```
>>>
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} , {} , {}'.format('a', 'b', 'c')    # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')           # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')       # arguments' indices can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>>
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N',
longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' attributes:

```
>>>
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary
part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```
>>>
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Replacing %s and %r:

```
>>>
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```
>>>
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered') # use '*' as a fill char
'*****centered*****'
```

Replacing %+f, %-f, and % f and specifying a sign:

```
>>>
>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14) # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14) # show only the minus -- same as '{:f};
{:f}'
'3.140000; -3.140000'
```

Replacing `%x` and `%o` and converting the value to different bases:

```
>>>
```

```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

Using the comma as a thousands separator:

```
>>>
```

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>>
```

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Using type-specific formatting:

```
>>>
```

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Nesting arguments and more complex examples:

```
>>>
```

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width),
end=' ')
...     print()
...
    5      5      5    101
    6      6      6    110
    7      7      7    111
    8      8     10   1000
    9      9     11   1001
   10     A     12   1010
   11     B     13   1011
```

6.1.4. Template strings

Templates provide simpler string substitutions as described in [PEP 292](#). Instead of the normal `%`-based substitutions, Templates support `$`-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `${identifier}` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` must spell a Python identifier. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${{identifier}}` is equivalent to `${identifier}`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `"${noun}ification"`.

Any other appearance of `$` in the string will result in a `ValueError` being raised.

The `string` module provides a `Template` class that implements these rules. The methods of `Template` are:

`class string.Template(template)`

The constructor takes a single argument which is the template string.

`substitute(mapping, **kws)`

Performs the template substitution, returning a new string. *mapping* is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both *mapping* and *kws* are given and there are duplicates, the placeholders from *kws* take precedence.

`safe_substitute(mapping, **kws)`

Like `substitute()`, except that if placeholders are missing from *mapping* and *kws*, instead of raising a `KeyError` exception, the original placeholder will appear in the resulting string intact. Also, unlike with `substitute()`, any other appearances of the `$` will simply return `$` instead of raising `ValueError`.

While other exceptions may still occur, this method is called “safe” because substitutions always tries to return a usable string instead of raising an exception. In another sense, `safe_substitute()` may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

`Template` instances also provide one public data attribute:

template

This is the object passed to the constructor's *template* argument. In general, you shouldn't change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```
>>>
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of `Template` to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- *delimiter* – This is the literal string describing a placeholder introducing delimiter. The default value is `$`. Note that this should *not* be a regular expression, as the implementation will call `re.escape()` on this string as needed.
- *idpattern* – This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression `[_a-z][_a-z0-9]*`.
- *flags* – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom *idpatterns* must follow conventions for verbose regular expressions.

New in version 3.2.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute *pattern*. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- *escaped* – This group matches the escape sequence, e.g. `$$`, in the default pattern.
- *named* – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- *braced* – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- *invalid* – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression