string — Text Constants and Templates

Purpose: Contains constants and classes for working with text.

The `string` module dates from the earliest versions of Python. Many of the functions previously implemented in this module have been moved to methods of `str` objects. The `string` module retains several useful constants and classes for working with `str` objects. This discussion will concentrate on them.

Functions

The function `capwords()` capitalizes all of the words in a string.

```
# string_capwords.py

import string

s = 'The quick brown fox jumped over the lazy dog.'

print(s)
print(string.capwords(s))
```

The results are the same as those obtained by calling `split()`, capitalizing the words in the resulting list, and then calling `join()` to combine the results.

```
$ python3 string_capwords.py

The quick brown fox jumped over the lazy dog.
The Quick Brown Fox Jumped Over The Lazy Dog.
```

Templates

String templates were added as part of **PEP 292** and are intended as an alternative to the built-in interpolation syntax. With `string.Template` interpolation, variables are identified by prefixing the name with $ (e.g., $var). Alternatively, if necessary to set them off from surrounding text, they can also be wrapped with curly braces (e.g., ${var}).

This example compares a simple template with similar string interpolation using the `%` operator and the new format string syntax using `str.format()`.

```
# string_template.py

import string

values = {'var': 'foo'}

t = string.Template("""
Variable        : $var
Escape          : $$
Variable in text: ${var}iable
""")

print('TEMPLATE:', t.substitute(values))

s = """
Variable        : %(var)s
Escape          : %%
Variable in text: %(var)siable
"""

print('INTERPOLATION:', s % values)

s = """
Variable        : {var}
```

```
Escape          : {{}}
Variable in text: {var}iable
"""

print('FORMAT:', s.format(**values))
```

In the first two cases, the trigger character ($ or %) is escaped by repeating it twice. For the format syntax, both { and } need to be escaped by repeating them.

```
$ python3 string_template.py

TEMPLATE:
Variable        : foo
Escape          : $
Variable in text: fooiable

INTERPOLATION:
Variable        : foo
Escape          : %
Variable in text: fooiable

FORMAT:
Variable        : foo
Escape          : {}
Variable in text: fooiable
```

One key difference between templates and string interpolation or formatting is that the type of the arguments is not taken into account. The values are converted to strings, and the strings are inserted into the result. No formatting options are available. For example, there is no way to control the number of digits used to represent a floating-point value.

A benefit, though, is that use of the `safe_substitute()` method makes it possible to avoid exceptions if not all of the values needed by the template are provided as arguments.

```
# string_template_missing.py

import string

values = {'var': 'foo'}

t = string.Template("$var is here but $missing is not provided")

try:
    print('substitute()     :', t.substitute(values))
except KeyError as err:
    print('ERROR:', str(err))

print('safe_substitute():', t.safe_substitute(values))
```

Since there is no value for `missing` in the values dictionary, a `KeyError` is raised by `substitute()`. Instead of raising the error, `safe_substitute()` catches it and leaves the variable expression alone in the text.

```
$ python3 string_template_missing.py

ERROR: 'missing'
safe_substitute(): foo is here but $missing is not provided
```

Advanced Templates

The default syntax for `string.Template` can be changed by adjusting the regular expression patterns it uses to find the variable names in the template body. A simple way to do that is to change the `delimiter` and `idpattern` class attributes.

```
# string_template_advanced.py

import string


class MyTemplate(string.Template):
    delimiter = '%'
    idpattern = '[a-z]+_[a-z]+'


template_text = '''
  Delimiter : %%
  Replaced  : %with_underscore
  Ignored   : %notunderscored
'''

d = {
    'with_underscore': 'replaced',
    'notunderscored': 'not replaced',
}

t = MyTemplate(template_text)
print('Modified ID pattern:')
print(t.safe_substitute(d))
```

In this example, the substitution rules are changed so that the delimiter is `%` instead of `$` and variable names must include an underscore somewhere in the middle. The pattern `%notunderscored` is not replaced by anything, because it does not include an underscore character.

```
$ python3 string_template_advanced.py

Modified ID pattern:

  Delimiter : %
  Replaced  : replaced
  Ignored   : %notunderscored
```

For even more complex changes, it is possible to override the `pattern` attribute and define an entirely new regular expression. The pattern provided must contain four named groups for capturing the escaped delimiter, the named variable, a braced version of the variable name, and invalid delimiter patterns.

```
# string_template_defaultpattern.py

import string

t = string.Template('$var')
print(t.pattern.pattern)
```

The value of `t.pattern` is a compiled regular expression, but the original string is available via its `pattern` attribute.

```
\$(?:
  (?P<escaped>\$) |                # two delimiters
  (?P<named>[_a-z][_a-z0-9]*)    | # identifier
  {(?P<braced>[_a-z][_a-z0-9]*)} | # braced identifier
  (?P<invalid>)                    # ill-formed delimiter exprs
)
```

This example defines a new pattern to create a new type of template, using `{{var}}` as the variable syntax.

```
# string_template_newsyntax.py

import re
import string


class MyTemplate(string.Template):
    delimiter = '{{'
    pattern = r'''
    \{\{(?:
    (?P<escaped>\{\{)|
    (?P<named>[_a-z][_a-z0-9]*)\}\}|
    (?P<braced>[_a-z][_a-z0-9]*)\}\}|
    (?P<invalid>)
    )
    '''
```

```
t = MyTemplate('''
{{{{
{{var}}
''')

print('MATCHES:', t.pattern.findall(t.template))
print('SUBSTITUTED:', t.safe_substitute(var='replacement'))
```

Both the named and braced patterns must be provided separately, even though they are the same. Running the sample program generates the following output:

```
$ python3 string_template_newsyntax.py

MATCHES: [('{{', '', '', ''), ('', 'var', '', '')]
SUBSTITUTED:
{{
replacement
```

Formatter

The Formatter class implements the same layout specification language as the format() method of str. Its features include type coersion, alignment, attribute and field references, named and positional template arguments, and type-specific formatting options. Most of the time the format() method is a more convenient interface to these features, but Formatter is provided as a way to build subclasses, for cases where variations are needed.

Constants

The string module includes a number of constants related to ASCII and numerical character sets.

```
# string_constants.py

import inspect
import string


def is_str(value):
    return isinstance(value, str)


for name, value in inspect.getmembers(string, is_str):
    if name.startswith('_'):
        continue
    print('%s=%r\n' % (name, value))
```

These constants are useful when working with ASCII data, but since it is increasingly common to encounter non-ASCII text in some form of Unicode, their application is limited.

```
$ python3 string_constants.py

ascii_letters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
XYZ'

ascii_lowercase='abcdefghijklmnopqrstuvwxyz'

ascii_uppercase='ABCDEFGHIJKLMNOPQRSTUVWXYZ'

digits='0123456789'

hexdigits='0123456789abcdefABCDEF'

octdigits='01234567'

printable='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
RSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'

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

whitespace=' \t\n\r\x0b\x0c'
```

```
whitespace=' \t\n\r\x0b\x0c
```

See also

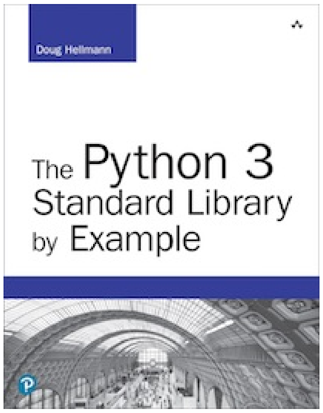
- Standard library documentation for string
- String Methods – Methods of `str` objects that replace the deprecated functions in `string`.
- **PEP 292** – Simpler String Substitutions
- Format String Syntax – The formal definition of the layout specification language used by `Formatter` and `str.format()`.

Text
textwrap — Formatting Text Paragraphs

Quick Links
Functions
Templates
Advanced Templates
Formatter
Constants

This page was last updated 2017-01-28.

Navigation
Text
textwrap — Formatting Text Paragraphs



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site
Module Index
Index

Other Writing
Blog
The Python Standard Library By Example