Working with Text Data

Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the `str` attribute and generally have names matching the equivalent (scalar) built-in string methods:

```
In [1]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [2]: s.str.lower()
Out[2]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object

In [3]: s.str.upper()
Out[3]:
0       A
1       B
2       C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object

In [4]: s.str.len()
Out[4]:
0    1.0
1    1.0
2    1.0
3    4.0
4    4.0
5    NaN
6    4.0
7    3.0
8    3.0
dtype: float64
```

```
In [5]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

In [6]: idx.str.strip()
Out[6]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [7]: idx.str.lstrip()
Out[7]: Index(['jack', 'jill ', 'jesse ', 'frank'], dtype='object')
```

```
In [8]: idx.str.rstrip()
Out[8]: Index([' jack', 'jill', ' jesse', 'frank'], dtype='object')
```

The string methods on Index are especially useful for cleaning up or transforming DataFrame columns. For instance, you may have columns with leading or trailing whitespace:

```
In [9]: df = pd.DataFrame(randn(3, 2), columns=[' Column A ', ' Column B '],
   ...:                   index=range(3))
   ...:

In [10]: df
Out[10]:
    Column A    Column B
0   -1.425575   -1.336299
1    0.740933    1.032121
2   -1.585660    0.913812
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```
In [11]: df.columns.str.strip()
Out[11]: Index(['Column A', 'Column B'], dtype='object')

In [12]: df.columns.str.lower()
Out[12]: Index([' column a ', ' column b '], dtype='object')
```

These string methods can then be used to clean up the columns as needed. Here we are removing leading and trailing whitespaces, lowercasing all names, and replacing any remaining whitespaces with underscores:

```
In [13]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [14]: df
Out[14]:
   column_a  column_b
0 -1.425575 -1.336299
1  0.740933  1.032121
2 -1.585660  0.913812
```

Note: If you have a `Series` where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`), it can be faster to convert the original `Series` to one of type `category` and then use `.str.<method>` or `.dt.<property>` on that. The performance difference comes from the fact that, for `Series` of type `category`, the string operations are done on the `.categories` and not on each element of the `Series`.

Please note that a `Series` of type `category` with string `.categories` has some limitations in comparison of `Series` of type string (e.g. you can't add strings to each other: `s + " " + s` won't work if `s` is a `Series` of type `category`). Also, `.str` methods which operate on elements of type `list` are not available on such a `Series`.

Splitting and Replacing Strings

Methods like `split` return a Series of lists:

```
In [15]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [16]: s2.str.split('_')
Out[16]:
0    [a, b, c]
1    [c, d, e]
2          NaN
3    [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using `get` or `[]` notation:

```
In [17]: s2.str.split('_').str.get(1)
Out[17]:
0      b
1      d
2    NaN
3      g
dtype: object

In [18]: s2.str.split('_').str[1]
Out[18]:
0      b
1      d
2    NaN
3      g
dtype: object
```

It is easy to expand this to return a DataFrame using `expand`.

```
In [19]: s2.str.split('_', expand=True)
Out[19]:
     0    1    2
0    a    b    c
1    c    d    e
2  NaN  NaN  NaN
3    f    g    h
```

It is also possible to limit the number of splits:

```
In [20]: s2.str.split('_', expand=True, n=1)
Out[20]:
     0    1
0    a  b_c
1    c  d_e
2  NaN  NaN
3    f  g_h
```

`rsplit` is similar to `split` except it works in the reverse direction, i.e., from the end of the string to the beginning of the string:

```
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
     0    1
0  a_b    c
1  c_d    e
2  NaN  NaN
3  f_g    h
```

`replace` by default replaces regular expressions:

```
In [22]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca',
   ....:                 '', np.nan, 'CABA', 'dog', 'cat'])
   ....:

In [23]: s3
Out[23]:
0       A
1       B
2       C
3    Aaba
4    Baca
5
6     NaN
7    CABA
8     dog
9     cat
dtype: object

In [24]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
Out[24]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6         NaN
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: object
```

Some caution must be taken to keep regular expressions in mind! For example, the following code will cause trouble because of the regular expression meaning of $:

```
# Consider the following badly formatted financial data
In [25]: dollars = pd.Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
In [26]: dollars.str.replace('$', '')
Out[26]:
0        12
1       -10
2    10,000
dtype: object
```

```
# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
Out[27]:
0         12
1       -$10
2    $10,000
dtype: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'-\$', '-')
Out[28]:
0         12
1        -10
2    $10,000
dtype: object
```

New in version 0.23.0.

If you do want literal replacement of a string (equivalent to **`str.replace()`**), you can set the optional `regex` parameter to `False`, rather than escaping each character. In this case both `pat` and `repl` must be strings:

```
# These lines are equivalent
In [29]: dollars.str.replace(r'-\$', '-')
Out[29]:
0         12
1        -10
2    $10,000
dtype: object

In [30]: dollars.str.replace('-$', '-', regex=False)
Out[30]:
0         12
1        -10
2    $10,000
dtype: object
```

New in version 0.20.0.

The `replace` method can also take a callable as replacement. It is called on every `pat` using **`re.sub()`**. The callable should expect one positional argument (a regex object) and return a string.

```
# Reverse every lowercase alphabetic word
In [31]: pat = r'[a-z]+'

In [32]: repl = lambda m: m.group(0)[::-1]

In [33]: pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(pat, repl)
Out[33]:
0    oof 123
1    rab zab
2        NaN
dtype: object

# Using regex groups
In [34]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"

In [35]: repl = lambda m: m.group('two').swapcase()
```

```
In [36]: pd.Series(['Foo Bar Baz', np.nan]).str.replace(pat, repl)
Out[36]:
0    bAR
1    NaN
dtype: object
```

New in version 0.20.0.

The `replace` method also accepts a compiled regular expression object from **`re.compile()`** as a pattern. All flags should be included in the compiled regular expression object.

```
In [37]: import re

In [38]: regex_pat = re.compile(r'^.a|dog', flags=re.IGNORECASE)

In [39]: s3.str.replace(regex_pat, 'XX-XX ')
Out[39]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6         NaN
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: object
```

Including a `flags` argument when calling `replace` with a compiled regular expression object will raise a `ValueError`.

```
In [40]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

Concatenation

There are several ways to concatenate a `Series` or `Index`, either with itself or others, all based on **`cat()`**, resp. `Index.str.cat`.

Concatenating a single Series into a string

The content of a `Series` (or `Index`) can be concatenated:

```
In [41]: s = pd.Series(['a', 'b', 'c', 'd'])

In [42]: s.str.cat(sep=',')
Out[42]: 'a,b,c,d'
```

If not specified, the keyword `sep` for the separator defaults to the empty string, `sep=''`:

```
In [43]: s.str.cat()
Out[43]: 'abcd'
```

By default, missing values are ignored. Using `na_rep`, they can be given a representation:

```
In [44]: t = pd.Series(['a', 'b', np.nan, 'd'])

In [45]: t.str.cat(sep=',')
Out[45]: 'a,b,d'

In [46]: t.str.cat(sep=',', na_rep='-')
Out[46]: 'a,b,-,d'
```

Concatenating a Series and something list-like into a Series

The first argument to **`cat()`** can be a list-like object, provided that it matches the length of the calling `Series` (or `Index`).

```
In [47]: s.str.cat(['A', 'B', 'C', 'D'])
Out[47]:
0    aA
1    bB
2    cC
3    dD
dtype: object
```

Missing values on either side will result in missing values in the result as well, unless `na_rep` is specified:

```
In [48]: s.str.cat(t)
Out[48]:
0     aa
1     bb
2    NaN
3     dd
dtype: object

In [49]: s.str.cat(t, na_rep='-')
Out[49]:
0    aa
1    bb
2    c-
3    dd
dtype: object
```

Concatenating a Series and something array-like into a Series

New in version 0.23.0.

The parameter `others` can also be two-dimensional. In this case, the number or rows must match the lengths of the calling `Series` (or `Index`).

```
In [50]: d = pd.concat([t, s], axis=1)

In [51]: s
Out[51]:
0    a
1    b
2    c
3    d
dtype: object

In [52]: d
Out[52]:
     0  1
0    a  a
1    b  b
2  NaN  c
3    d  d

In [53]: s.str.cat(d, na_rep='-')
Out[53]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and an indexed object into a Series, with alignment

New in version 0.23.0.

For concatenation with a `Series` or `DataFrame`, it is possible to align the indexes before concatenation by setting the `join`-keyword.

```
In [54]: u = pd.Series(['b', 'd', 'a', 'c'], index=[1, 3, 0, 2])

In [55]: s
Out[55]:
0    a
1    b
2    c
3    d
dtype: object

In [56]: u
Out[56]:
1    b
3    d
0    a
2    c
dtype: object

In [57]: s.str.cat(u)
Out[57]:
0    ab
1    bd
```

```
2    ca
3    dc
dtype: object

In [58]: s.str.cat(u, join='left')
Out[58]:
0    aa
1    bb
2    cc
3    dd
dtype: object
```

Warning: If the `join` keyword is not passed, the method **`cat()`** will currently fall back to the behavior before version 0.23.0 (i.e. no alignment), but a `FutureWarning` will be raised if any of the involved indexes differ, since this default will change to `join='left'` in a future version.

The usual options are available for `join` (one of `'left', 'outer', 'inner', 'right'`). In particular, alignment also means that the different lengths do not need to coincide anymore.

```
In [59]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4])

In [60]: s
Out[60]:
0    a
1    b
2    c
3    d
dtype: object

In [61]: v
Out[61]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: object

In [62]: s.str.cat(v, join='left', na_rep='-')
Out[62]:
0    aa
1    bb
2    c-
3    dd
dtype: object

In [63]: s.str.cat(v, join='outer', na_rep='-')
Out[63]:
-1    -z
 0    aa
 1    bb
 2    c-
 3    dd
 4    -e
dtype: object
```

The same alignment can be used when `others` is a `DataFrame`:

```
In [64]: f = d.loc[[3, 2, 1, 0], :]

In [65]: s
Out[65]:
0    a
1    b
2    c
3    d
dtype: object

In [66]: f
Out[66]:
     0  1
3    d  d
2  NaN  c
1    b  b
0    a  a

In [67]: s.str.cat(f, join='left', na_rep='-')
Out[67]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

Concatenating a Series and many objects into a Series

All one-dimensional list-likes can be arbitrarily combined in a list-like container (including iterators, `dict`-views, etc.):

```
In [68]: s
Out[68]:
0    a
1    b
2    c
3    d
dtype: object

In [69]: u
Out[69]:
1    b
3    d
0    a
2    c
dtype: object

In [70]: s.str.cat([u, pd.Index(u.values), ['A', 'B', 'C', 'D'], map(str, u.index)], na
Out[70]:
0    abbA1
1    bddB3
2    caaC0
3    dccD2
dtype: object
```

All elements must match in length to the calling `Series` (or `Index`), except those having an index if `join` is not None:

```
In [71]: v
Out[71]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: object

In [72]: s.str.cat([u, v, ['A', 'B', 'C', 'D']], join='outer', na_rep='-')
Out[72]:
-1    --z-
 0    aaaA
 1    bbbB
 2    cc-C
 3    dddD
 4    --e-
dtype: object
```

If using `join='right'` on a list of `others` that contains different indexes, the union of these indexes will be used as the basis for the final concatenation:

```
In [73]: u.loc[[3]]
Out[73]:
3    d
dtype: object

In [74]: v.loc[[-1, 0]]
Out[74]:
-1    z
 0    a
dtype: object

In [75]: s.str.cat([u.loc[[3]], v.loc[[-1, 0]]], join='right', na_rep='-')
Out[75]:
-1    --z
 0    a-a
 3    dd-
dtype: object
```

Indexing with `.str`

You can use `[]` notation to directly index by position locations. If you index past the end of the string, the result will be a `NaN`.

```
In [76]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
   ....:                'CABA', 'dog', 'cat'])
   ....:

In [77]: s.str[0]
Out[77]:
0      A
1      B
2      C
3      A
4      B
```

```
5    NaN
6      C
7      d
8      c
dtype: object

In [78]: s.str[1]
Out[78]:
0    NaN
1    NaN
2    NaN
3      a
4      a
5    NaN
6      A
7      o
8      a
dtype: object
```

Extracting Substrings

Extract first match in each subject (extract)

Warning: In version 0.18.0, `extract` gained the `expand` argument. When `expand=False` it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0). When `expand=True` it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user. `expand=True` is the default since version 0.23.0.

The `extract` method accepts a regular expression with at least one capture group.

Extracting a regular expression with more than one group returns a DataFrame with one column per group.

```
In [79]: pd.Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)', expand=False)
Out[79]:
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

Elements that do not match return a row filled with `NaN`. Thus, a Series of messy strings can be “converted” into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects. The dtype of the result is always object, even if no match is found and the result only contains `NaN`.

Named groups like

```
In [80]: pd.Series(['a1', 'b2', 'c3']).str.extract('(?P<letter>[ab])(?P<digit>\d)', exp
Out[80]:
  letter digit
0      a     1
```

```
1      b     2
2    NaN   NaN
```

and optional groups like

```
In [81]: pd.Series(['a1', 'b2', '3']).str.extract('([ab])?(\d)', expand=False)
Out[81]:
     0  1
0    a  1
1    b  2
2  NaN  3
```

can also be used. Note that any capture group names in the regular expression will be used for column names; otherwise capture group numbers will be used.

Extracting a regular expression with one group returns a `DataFrame` with one column if `expand=True`.

```
In [82]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=True)
Out[82]:
     0
0    1
1    2
2  NaN
```

It returns a Series if `expand=False`.

```
In [83]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=False)
Out[83]:
0      1
1      2
2    NaN
dtype: object
```

Calling on an `Index` with a regex with exactly one capture group returns a `DataFrame` with one column if `expand=True`.

```
In [84]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])

In [85]: s
Out[85]:
A11    a1
B22    b2
C33    c3
dtype: object

In [86]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[86]:
  letter
0      A
1      B
2      C
```

It returns an `Index` if `expand=False`.

```
In [87]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[87]: Index(['A', 'B', 'C'], dtype='object', name='letter')
```

Calling on an `Index` with a regex with more than one capture group returns a `DataFrame` if `expand=True`.

```
In [88]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[88]:
  letter   1
0      A  11
1      B  22
2      C  33
```

It raises `ValueError` if `expand=False`.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

The table below summarizes the behavior of `extract(expand=False)` (input subject in first column, number of groups in regex in first row)

	1 group	>1 group
Index	Index	ValueError
Series	Series	DataFrame

Extract all matches in each subject (extractall)

New in version 0.18.0.

Unlike `extract` (which returns only the first match),

```
In [89]: s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])

In [90]: s
Out[90]:
A    a1a2
B      b1
C      c1
dtype: object

In [91]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'

In [92]: s.str.extract(two_groups, expand=True)
Out[92]:
  letter digit
A      a     1
B      b     1
C      c     1
```

the `extractall` method returns every match. The result of `extractall` is always a `DataFrame` with a `MultiIndex` on its rows. The last level of the `MultiIndex` is named `match` and indicates the order in the subject.

```
In [93]: s.str.extractall(two_groups)
Out[93]:
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
C 0          c     1
```

When each subject string in the Series has exactly one match,

```
In [94]: s = pd.Series(['a3', 'b3', 'c2'])

In [95]: s
Out[95]:
0    a3
1    b3
2    c2
dtype: object
```

then `extractall(pat).xs(0, level='match')` gives the same result as `extract(pat)`.

```
In [96]: extract_result = s.str.extract(two_groups, expand=True)

In [97]: extract_result
Out[97]:
  letter digit
0      a     3
1      b     3
2      c     2

In [98]: extractall_result = s.str.extractall(two_groups)

In [99]: extractall_result
Out[99]:
        letter digit
  match
0 0          a     3
1 0          b     3
2 0          c     2

In [100]: extractall_result.xs(0, level="match")
Out[100]:
  letter digit
0      a     3
1      b     3
2      c     2
```

`Index` also supports `.str.extractall`. It returns a `DataFrame` which has the same result as a `Series.str.extractall` with a default index (starts from 0).

New in version 0.19.0.

```
In [101]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[101]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1

In [102]: pd.Series(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[102]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1
```

Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

```
In [103]: pattern = r'[0-9][a-z]'

In [104]: pd.Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
Out[104]:
0    False
1    False
2     True
3     True
4     True
dtype: bool
```

Or whether elements match a pattern:

```
In [105]: pd.Series(['1', '2', '3a', '3b', '03c']).str.match(pattern)
Out[105]:
0    False
1    False
2     True
3     True
4    False
dtype: bool
```

The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match`, while `contains` relies on `re.search`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered True or False:

```
In [106]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [107]: s4.str.contains('A', na=False)
Out[107]:
0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
dtype: bool
```

Creating Indicator Variables

You can extract dummy variables from string columns. For example if they are separated by a `'|'`:

```
In [108]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])

In [109]: s.str.get_dummies(sep='|')
Out[109]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1
```

String `Index` also supports `get_dummies` which returns a `MultiIndex`.

New in version 0.18.1.

```
In [110]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])

In [111]: idx.str.get_dummies(sep='|')
Out[111]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
           labels=[[1, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 1]],
           names=['a', 'b', 'c'])
```

See also **`get_dummies()`**.

Method Summary

Method	Description
`cat()`	Concatenate strings
`split()`	Split strings on delimiter
`rsplit()`	Split strings on delimiter working from the end of the string
`get()`	Index into each element (retrieve i-th element)
`join()`	Join strings in each element of the Series with passed separator

Method	Description
`get_dummies()`	Split strings on the delimiter returning DataFrame of dummy variables
`contains()`	Return boolean array if each string contains pattern/regex
`replace()`	Replace occurrences of pattern/regex/string with some other string or the return value of a callable given the occurrence
`repeat()`	Duplicate values (`s.str.repeat(3)` equivalent to `x * 3`)
`pad()`	Add whitespace to left, right, or both sides of strings
`center()`	Equivalent to `str.center`
`ljust()`	Equivalent to `str.ljust`
`rjust()`	Equivalent to `str.rjust`
`zfill()`	Equivalent to `str.zfill`
`wrap()`	Split long strings into lines with length less than a given width
`slice()`	Slice each string in the Series
`slice_replace()`	Replace slice in each string with passed value
`count()`	Count occurrences of pattern
`startswith()`	Equivalent to `str.startswith(pat)` for each element
`endswith()`	Equivalent to `str.endswith(pat)` for each element
`findall()`	Compute list of all occurrences of pattern/regex for each string
`match()`	Call `re.match` on each element, returning matched groups as list
`extract()`	Call `re.search` on each element, returning DataFrame with one row for each element and one column for each regex capture group
`extractall()`	Call `re.findall` on each element, returning DataFrame with one row for each match and one column for each regex capture group
`len()`	Compute string lengths
`strip()`	Equivalent to `str.strip`
`rstrip()`	Equivalent to `str.rstrip`
`lstrip()`	Equivalent to `str.lstrip`
`partition()`	Equivalent to `str.partition`
`rpartition()`	Equivalent to `str.rpartition`
`lower()`	Equivalent to `str.lower`
`upper()`	Equivalent to `str.upper`
`find()`	Equivalent to `str.find`
`rfind()`	Equivalent to `str.rfind`
`index()`	Equivalent to `str.index`
`rindex()`	Equivalent to `str.rindex`
`capitalize()`	Equivalent to `str.capitalize`
`swapcase()`	Equivalent to `str.swapcase`
`normalize()`	Return Unicode normal form. Equivalent to `unicodedata.normalize`
`translate()`	Equivalent to `str.translate`
`isalnum()`	Equivalent to `str.isalnum`
`isalpha()`	Equivalent to `str.isalpha`
`isdigit()`	Equivalent to `str.isdigit`
`isspace()`	Equivalent to `str.isspace`
`islower()`	Equivalent to `str.islower`
`isupper()`	Equivalent to `str.isupper`
`istitle()`	Equivalent to `str.istitle`
`isnumeric()`	Equivalent to `str.isnumeric`
`isdecimal()`	Equivalent to `str.isdecimal`