(continued from previous page)

```
In [118]: df = cols + pd.DataFrame((np.random.randint(5, size=(n, 4))
   .....:                           // [2, 1, 2, 1]).astype(str))
   .....:

In [119]: df.columns = cols

In [120]: df = df.join(pd.DataFrame(np.random.rand(n, 2).round(2)).add_prefix('val'))

In [121]: df
Out[121]:
     key   row   item   col  val0  val1
0   key0  row3  item1  col3  0.81  0.04
1   key1  row2  item1  col2  0.44  0.07
2   key1  row0  item1  col0  0.77  0.01
3   key0  row4  item0  col2  0.15  0.59
4   key1  row0  item2  col1  0.81  0.64
..   ...   ...    ...   ...   ...   ...
15  key0  row3  item1  col1  0.31  0.23
16  key0  row0  item2  col3  0.86  0.01
17  key0  row4  item0  col3  0.64  0.21
18  key2  row2  item2  col0  0.13  0.45
19  key0  row2  item0  col4  0.37  0.70

[20 rows x 6 columns]
```

### Pivoting with single aggregations

Suppose we wanted to pivot `df` such that the `col` values are columns, `row` values are the index, and the mean of `val0` are the values? In particular, the resulting DataFrame should look like:

```
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65
row2  0.13    NaN  0.395  0.500  0.25
row3   NaN  0.310    NaN  0.545   NaN
row4   NaN  0.100  0.395  0.760  0.24
```

This solution uses *pivot_table()*. Also note that `aggfunc='mean'` is the default. It is included here to be explicit.

```
In [122]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean')
   .....:
Out[122]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65
row2  0.13    NaN  0.395  0.500  0.25
row3   NaN  0.310    NaN  0.545   NaN
row4   NaN  0.100  0.395  0.760  0.24
```

Note that we can also replace the missing values by using the `fill_value` parameter.

```
In [123]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='mean', fill_value=0)
   .....:
Out[123]:
col   col0   col1   col2   col3  col4
row
row0  0.77  0.605  0.000  0.860  0.65
row2  0.13  0.000  0.395  0.500  0.25
row3  0.00  0.310  0.000  0.545  0.00
row4  0.00  0.100  0.395  0.760  0.24
```

Also note that we can pass in other aggregation functions as well. For example, we can also pass in `sum`.

```
In [124]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc='sum', fill_value=0)
   .....:
Out[124]:
col   col0  col1  col2  col3  col4
row
row0  0.77  1.21  0.00  0.86  0.65
row2  0.13  0.00  0.79  0.50  0.50
row3  0.00  0.31  0.00  1.09  0.00
row4  0.00  0.10  0.79  1.52  0.24
```

Another aggregation we can do is calculate the frequency in which the columns and rows occur together a.k.a. "cross tabulation". To do this, we can pass `size` to the `aggfunc` parameter.

```
In [125]: df.pivot_table(index='row', columns='col', fill_value=0, aggfunc='size')
Out[125]:
col   col0  col1  col2  col3  col4
row
row0     1     2     0     1     1
row2     1     0     2     1     2
row3     0     1     0     2     0
row4     0     1     2     2     1
```

### Pivoting with multiple aggregations

We can also perform multiple aggregations. For example, to perform both a `sum` and `mean`, we can pass in a list to the `aggfunc` argument.

```
In [126]: df.pivot_table(
   .....:     values='val0', index='row', columns='col', aggfunc=['mean', 'sum'])
   .....:
Out[126]:
      mean                                sum
col   col0   col1   col2   col3  col4  col0  col1  col2  col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65  0.77  1.21   NaN  0.86  0.65
row2  0.13    NaN  0.395  0.500  0.25  0.13   NaN  0.79  0.50  0.50
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.31   NaN  1.09   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.10  0.79  1.52  0.24
```

Note to aggregate over multiple value columns, we can pass in a list to the `values` parameter.

```
In [127]: df.pivot_table(
   .....:     values=['val0', 'val1'], index='row', columns='col', aggfunc=['mean'])
   .....:
Out[127]:
      mean
      val0                              val1
col   col0   col1   col2   col3  col4  col0   col1  col2   col3  col4
row
row0  0.77  0.605    NaN  0.860  0.65  0.01  0.745   NaN  0.010  0.02
row2  0.13    NaN  0.395  0.500  0.25  0.45    NaN  0.34  0.440  0.79
row3   NaN  0.310    NaN  0.545   NaN   NaN  0.230   NaN  0.075   NaN
row4   NaN  0.100  0.395  0.760  0.24   NaN  0.070  0.42  0.300  0.46
```

Note to subdivide over multiple columns we can pass in a list to the `columns` parameter.

```
In [128]: df.pivot_table(
   .....:     values=['val0'], index='row', columns=['item', 'col'], aggfunc=['mean'])
   .....:
Out[128]:
      mean
      val0
item item0             item1                         item2
col   col2  col3  col4  col0  col1  col2  col3  col4  col0   col1  col3  col4
row
row0   NaN   NaN   NaN  0.77   NaN   NaN   NaN   NaN   NaN  0.605  0.86  0.65
row2  0.35   NaN  0.37   NaN   NaN  0.44   NaN   NaN  0.13    NaN  0.50  0.13
row3   NaN   NaN   NaN   NaN  0.31   NaN  0.81   NaN   NaN    NaN  0.28   NaN
row4  0.15  0.64   NaN   NaN  0.10  0.64  0.88  0.24   NaN    NaN   NaN   NaN
```

### 2.5.11 Exploding a list-like column

New in version 0.25.0.

Sometimes the values in a column are list-like.

```
In [129]: keys = ['panda1', 'panda2', 'panda3']

In [130]: values = [['eats', 'shoots'], ['shoots', 'leaves'], ['eats', 'leaves']]

In [131]: df = pd.DataFrame({'keys': keys, 'values': values})

In [132]: df
Out[132]:
     keys            values
0  panda1    [eats, shoots]
1  panda2  [shoots, leaves]
2  panda3    [eats, leaves]
```

We can 'explode' the `values` column, transforming each list-like to a separate row, by using *explode()*. This will replicate the index values from the original row:

```
In [133]: df['values'].explode()
Out[133]:
0      eats
0    shoots
1    shoots
```

(continues on next page)

(continued from previous page)

```
1    leaves
2      eats
2    leaves
Name: values, dtype: object
```

You can also explode the column in the DataFrame.

```
In [134]: df.explode('values')
Out[134]:
     keys  values
0  panda1    eats
0  panda1  shoots
1  panda2  shoots
1  panda2  leaves
2  panda3    eats
2  panda3  leaves
```

*Series.explode()* will replace empty lists with np.nan and preserve scalar entries. The dtype of the resulting Series is always object.

```
In [135]: s = pd.Series([[1, 2, 3], 'foo', [], ['a', 'b']])

In [136]: s
Out[136]:
0    [1, 2, 3]
1          foo
2           []
3       [a, b]
dtype: object

In [137]: s.explode()
Out[137]:
0      1
0      2
0      3
1    foo
2    NaN
3      a
3      b
dtype: object
```

Here is a typical usecase. You have comma separated strings in a column and want to expand this.

```
In [138]: df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1},
   .....:                    {'var1': 'd,e,f', 'var2': 2}])
   .....:

In [139]: df
Out[139]:
    var1  var2
0  a,b,c     1
1  d,e,f     2
```

Creating a long form DataFrame is now straightforward using explode and chained operations

```
In [140]: df.assign(var1=df.var1.str.split(',')).explode('var1')
Out[140]:
  var1  var2
0    a     1
0    b     1
0    c     1
1    d     2
1    e     2
1    f     2
```

## 2.6 Working with text data

### 2.6.1 Text Data Types

New in version 1.0.0.

There are two ways to store text data in pandas:

1. `object` -dtype NumPy array.
2. `StringDtype` extension type.

We recommend using `StringDtype` to store text data.

Prior to pandas 1.0, `object` dtype was the only option. This was unfortunate for many reasons:

1. You can accidentally store a *mixture* of strings and non-strings in an `object` dtype array. It's better to have a dedicated dtype.
2. `object` dtype breaks dtype-specific operations like `DataFrame.select_dtypes()`. There isn't a clear way to select *just* text while excluding non-text but still object-dtype columns.
3. When reading code, the contents of an `object` dtype array is less clear than `'string'`.

Currently, the performance of `object` dtype arrays of strings and `arrays.StringArray` are about the same. We expect future enhancements to significantly increase the performance and lower the memory overhead of `StringArray`.

> **Warning:** `StringArray` is currently considered experimental. The implementation and parts of the API may change without warning.

For backwards-compatibility, `object` dtype remains the default type we infer a list of strings to

```
In [1]: pd.Series(['a', 'b', 'c'])
Out[1]:
0    a
1    b
2    c
dtype: object
```

To explicitly request `string` dtype, specify the `dtype`

```
In [2]: pd.Series(['a', 'b', 'c'], dtype="string")
Out[2]:
0    a
```

(continues on next page)

(continued from previous page)

```
1    b
2    c
dtype: string

In [3]: pd.Series(['a', 'b', 'c'], dtype=pd.StringDtype())
Out[3]:
0    a
1    b
2    c
dtype: string
```

Or `astype` after the `Series` or `DataFrame` is created

```
In [4]: s = pd.Series(['a', 'b', 'c'])

In [5]: s
Out[5]:
0    a
1    b
2    c
dtype: object

In [6]: s.astype("string")
Out[6]:
0    a
1    b
2    c
dtype: string
```

### Behavior differences

These are places where the behavior of `StringDtype` objects differ from `object` dtype

1. For `StringDtype`, *string accessor methods* that return **numeric** output will always return a nullable integer dtype, rather than either int or float dtype, depending on the presence of NA values. Methods returning **boolean** output will return a nullable boolean dtype.

```
In [7]: s = pd.Series(["a", None, "b"], dtype="string")

In [8]: s
Out[8]:
0       a
1    <NA>
2       b
dtype: string

In [9]: s.str.count("a")
Out[9]:
0       1
1    <NA>
2       0
dtype: Int64

In [10]: s.dropna().str.count("a")
Out[10]:
```

(continues on next page)

(continued from previous page)

```
0    1
2    0
dtype: Int64
```

Both outputs are `Int64` dtype. Compare that with object-dtype

```
In [11]: s2 = pd.Series(["a", None, "b"], dtype="object")

In [12]: s2.str.count("a")
Out[12]:
0    1.0
1    NaN
2    0.0
dtype: float64

In [13]: s2.dropna().str.count("a")
Out[13]:
0    1
2    0
dtype: int64
```

When NA values are present, the output dtype is float64. Similarly for methods returning boolean values.

```
In [14]: s.str.isdigit()
Out[14]:
0    False
1     <NA>
2    False
dtype: boolean

In [15]: s.str.match("a")
Out[15]:
0     True
1     <NA>
2    False
dtype: boolean
```

2. Some string methods, like *`Series.str.decode()`* are not available on `StringArray` because `StringArray` only holds strings, not bytes.
3. In comparison operations, *`arrays.StringArray`* and `Series` backed by a `StringArray` will return an object with *`BooleanDtype`*, rather than a `bool` dtype object. Missing values in a `StringArray` will propagate in comparison operations, rather than always comparing unequal like `numpy.nan`.

Everything else that follows in the rest of this document applies equally to `string` and `object` dtype.

### 2.6.2 String Methods

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 [16]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'],
   ....:               dtype="string")
   ....:
```

(continues on next page)

(continued from previous page)

```
In [17]: s.str.lower()
Out[17]:
0       a
1       b
2       c
3    aaba
4    baca
5    <NA>
6    caba
7     dog
8     cat
dtype: string

In [18]: s.str.upper()
Out[18]:
0       A
1       B
2       C
3    AABA
4    BACA
5    <NA>
6    CABA
7     DOG
8     CAT
dtype: string

In [19]: s.str.len()
Out[19]:
0       1
1       1
2       1
3       4
4       4
5    <NA>
6       4
7       3
8       3
dtype: Int64
```

```
In [20]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

In [21]: idx.str.strip()
Out[21]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [22]: idx.str.lstrip()
Out[22]: Index(['jack', 'jill ', 'jesse ', 'frank'], dtype='object')

In [23]: idx.str.rstrip()
Out[23]: 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 [24]: df = pd.DataFrame(np.random.randn(3, 2),
   ....:                   columns=[' Column A ', ' Column B '], index=range(3))
```

(continues on next page)

(continued from previous page)

```
   ....:

In [25]: df
Out[25]:
    Column A    Column B
0   0.469112   -0.282863
1  -1.509059   -1.135632
2   1.212112   -0.173215
```

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

```
In [26]: df.columns.str.strip()
Out[26]: Index(['Column A', 'Column B'], dtype='object')

In [27]: df.columns.str.lower()
Out[27]: 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, lower casing all names, and replacing any remaining whitespaces with underscores:

```
In [28]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [29]: df
Out[29]:
   column_a  column_b
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
```

**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 to `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`.

**Warning:** Before v.0.25.0, the `.str`-accessor did only the most rudimentary type checks. Starting with v.0.25.0, the type of the Series is inferred and the allowed types (i.e. strings) are enforced more rigorously.

Generally speaking, the `.str` accessor is intended to work only on strings. With very few exceptions, other uses are not supported, and may be disabled at a later point.

### 2.6.3 Splitting and replacing strings

Methods like `split` return a Series of lists:

```
In [30]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'], dtype="string")

In [31]: s2.str.split('_')
Out[31]:
0    [a, b, c]
1    [c, d, e]
2         <NA>
3    [f, g, h]
dtype: object
```

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

```
In [32]: s2.str.split('_').str.get(1)
Out[32]:
0       b
1       d
2    <NA>
3       g
dtype: object

In [33]: s2.str.split('_').str[1]
Out[33]:
0       b
1       d
2    <NA>
3       g
dtype: object
```

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

```
In [34]: s2.str.split('_', expand=True)
Out[34]:
      0     1     2
0     a     b     c
1     c     d     e
2  <NA>  <NA>  <NA>
3     f     g     h
```

When original `Series` has *`StringDtype`*, the output columns will all be *`StringDtype`* as well.

It is also possible to limit the number of splits:

```
In [35]: s2.str.split('_', expand=True, n=1)
Out[35]:
      0     1
0     a   b_c
1     c   d_e
2  <NA>  <NA>
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 [36]: s2.str.rsplit('_', expand=True, n=1)
Out[36]:
      0     1
0   a_b     c
1   c_d     e
2  <NA>  <NA>
3   f_g     h
```

replace by default replaces regular expressions:

```
In [37]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca',
   ....:                  '', np.nan, 'CABA', 'dog', 'cat'],
   ....:                 dtype="string")
   ....:

In [38]: s3
Out[38]:
0       A
1       B
2       C
3    Aaba
4    Baca
5
6    <NA>
7    CABA
8     dog
9     cat
dtype: string

In [39]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
Out[39]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6        <NA>
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: string
```

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 [40]: dollars = pd.Series(['12', '-$10', '$10,000'], dtype="string")

# This does what you'd naively expect:
In [41]: dollars.str.replace('$', '')
Out[41]:
0        12
1       -10
2    10,000
dtype: string
```

(continues on next page)

(continued from previous page)

```
# But this doesn't:
In [42]: dollars.str.replace('-$', '-')
Out[42]:
0         12
1       -$10
2    $10,000
dtype: string

# We need to escape the special character (for >1 len patterns)
In [43]: dollars.str.replace(r'-\$', '-')
Out[43]:
0         12
1        -10
2    $10,000
dtype: string
```

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 [44]: dollars.str.replace(r'-\$', '-')
Out[44]:
0         12
1        -10
2    $10,000
dtype: string

In [45]: dollars.str.replace('-$', '-', regex=False)
Out[45]:
0         12
1        -10
2    $10,000
dtype: string
```

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 [46]: pat = r'[a-z]+'

In [47]: def repl(m):
   ....:     return m.group(0)[::-1]
   ....:

In [48]: pd.Series(['foo 123', 'bar baz', np.nan],
   ....:           dtype="string").str.replace(pat, repl)
   ....:
Out[48]:
0    oof 123
1    rab zab
2       <NA>
dtype: string

# Using regex groups
In [49]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
```

(continues on next page)

(continued from previous page)

```
In [50]: def repl(m):
   ....:     return m.group('two').swapcase()
   ....:

In [51]: pd.Series(['Foo Bar Baz', np.nan],
   ....:           dtype="string").str.replace(pat, repl)
   ....:
Out[51]:
0     bAR
1    <NA>
dtype: string
```

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 [52]: import re

In [53]: regex_pat = re.compile(r'^.a|dog', flags=re.IGNORECASE)

In [54]: s3.str.replace(regex_pat, 'XX-XX ')
Out[54]:
0           A
1           B
2           C
3    XX-XX ba
4    XX-XX ca
5
6        <NA>
7    XX-XX BA
8      XX-XX
9     XX-XX t
dtype: string
```

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

```
In [55]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
---------------------------------------------------------------------------
ValueError: case and flags cannot be set when pat is a compiled regex
```

### 2.6.4 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 [56]: s = pd.Series(['a', 'b', 'c', 'd'], dtype="string")

In [57]: s.str.cat(sep=',')
Out[57]: 'a,b,c,d'
```

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

```
In [58]: s.str.cat()
Out[58]: 'abcd'
```

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

```
In [59]: t = pd.Series(['a', 'b', np.nan, 'd'], dtype="string")

In [60]: t.str.cat(sep=',')
Out[60]: 'a,b,d'

In [61]: t.str.cat(sep=',', na_rep='-')
Out[61]: '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 [62]: s.str.cat(['A', 'B', 'C', 'D'])
Out[62]:
0    aA
1    bB
2    cC
3    dD
dtype: string
```

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

```
In [63]: s.str.cat(t)
Out[63]:
0      aa
1      bb
2    <NA>
3      dd
dtype: string

In [64]: s.str.cat(t, na_rep='-')
Out[64]:
0    aa
```

(continues on next page)

(continued from previous page)

```
1    bb
2    c-
3    dd
dtype: string
```

### 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 [65]: d = pd.concat([t, s], axis=1)

In [66]: s
Out[66]:
0    a
1    b
2    c
3    d
dtype: string

In [67]: d
Out[67]:
      0  1
0     a  a
1     b  b
2  <NA>  c
3     d  d

In [68]: s.str.cat(d, na_rep='-')
Out[68]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: string
```

### 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 [69]: u = pd.Series(['b', 'd', 'a', 'c'], index=[1, 3, 0, 2],
   ....:               dtype="string")
   ....:

In [70]: s
Out[70]:
0    a
1    b
2    c
```

(continues on next page)

(continued from previous page)

```
3    d
dtype: string

In [71]: u
Out[71]:
1    b
3    d
0    a
2    c
dtype: string

In [72]: s.str.cat(u)
Out[72]:
0    aa
1    bb
2    cc
3    dd
dtype: string

In [73]: s.str.cat(u, join='left')
Out[73]:
0    aa
1    bb
2    cc
3    dd
dtype: string
```

**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 [74]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4],
   ....:               dtype="string")
   ....:

In [75]: s
Out[75]:
0    a
1    b
2    c
3    d
dtype: string

In [76]: v
Out[76]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: string
```

(continues on next page)

(continued from previous page)

```
In [77]: s.str.cat(v, join='left', na_rep='-')
Out[77]:
0    aa
1    bb
2    c-
3    dd
dtype: string

In [78]: s.str.cat(v, join='outer', na_rep='-')
Out[78]:
-1    -z
 0    aa
 1    bb
 2    c-
 3    dd
 4    -e
dtype: string
```

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

```
In [79]: f = d.loc[[3, 2, 1, 0], :]

In [80]: s
Out[80]:
0    a
1    b
2    c
3    d
dtype: string

In [81]: f
Out[81]:
      0  1
3     d  d
2  <NA>  c
1     b  b
0     a  a

In [82]: s.str.cat(f, join='left', na_rep='-')
Out[82]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: string
```

### Concatenating a Series and many objects into a Series

Several array-like items (specifically: `Series`, `Index`, and 1-dimensional variants of `np.ndarray`) can be combined in a list-like container (including iterators, `dict`-views, etc.).

```
In [83]: s
Out[83]:
0    a
1    b
2    c
3    d
dtype: string

In [84]: u
Out[84]:
1    b
3    d
0    a
2    c
dtype: string

In [85]: s.str.cat([u, u.to_numpy()], join='left')
Out[85]:
0    aab
1    bbd
2    cca
3    ddc
dtype: string
```

All elements without an index (e.g. `np.ndarray`) within the passed list-like must match in length to the calling `Series` (or `Index`), but `Series` and `Index` may have arbitrary length (as long as alignment is not disabled with `join=None`):

```
In [86]: v
Out[86]:
-1    z
 0    a
 1    b
 3    d
 4    e
dtype: string

In [87]: s.str.cat([v, u, u.to_numpy()], join='outer', na_rep='-')
Out[87]:
-1    -z--
 0    aaab
 1    bbbd
 2    c-ca
 3    dddc
 4    -e--
dtype: string
```

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

```
In [88]: u.loc[[3]]
Out[88]:
```

(continues on next page)

(continued from previous page)

```
3    d
dtype: string

In [89]: v.loc[[-1, 0]]
Out[89]:
-1    z
 0    a
dtype: string

In [90]: s.str.cat([u.loc[[3]], v.loc[[-1, 0]]], join='right', na_rep='-')
Out[90]:
-1    --z
 0    a-a
 3    dd-
dtype: string
```

### 2.6.5 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 [91]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
   ....:                'CABA', 'dog', 'cat'],
   ....:               dtype="string")
   ....:

In [92]: s.str[0]
Out[92]:
0       A
1       B
2       C
3       A
4       B
5    <NA>
6       C
7       d
8       c
dtype: string

In [93]: s.str[1]
Out[93]:
0    <NA>
1    <NA>
2    <NA>
3       a
4       a
5    <NA>
6       A
7       o
8       a
dtype: string
```

### 2.6.6 Extracting substrings

#### Extract first match in each subject (extract)

> **Warning:** Before version 0.23, argument `expand` of the `extract` method defaulted to `False`. When `expand=False`, `expand` returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern. When `expand=True`, it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user. `expand=True` has been 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 [94]: pd.Series(['a1', 'b2', 'c3'],
   ....:           dtype="string").str.extract(r'([ab])(\d)', expand=False)
   ....:
Out[94]:
      0     1
0     a     1
1     b     2
2  <NA>  <NA>
```

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 [95]: pd.Series(['a1', 'b2', 'c3'],
   ....:           dtype="string").str.extract(r'(?P<letter>[ab])(?P<digit>\d)',
   ....:                                       expand=False)
   ....:
Out[95]:
  letter digit
0      a     1
1      b     2
2   <NA>  <NA>
```

and optional groups like

```
In [96]: pd.Series(['a1', 'b2', '3'],
   ....:           dtype="string").str.extract(r'([ab])?(\d)', expand=False)
   ....:
Out[96]:
      0  1
0     a  1
1     b  2
2  <NA>  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 [97]: pd.Series(['a1', 'b2', 'c3'],
   ....:           dtype="string").str.extract(r'[ab](\d)', expand=True)
   ....:
Out[97]:
      0
0     1
1     2
2  <NA>
```

It returns a Series if `expand=False`.

```
In [98]: pd.Series(['a1', 'b2', 'c3'],
   ....:           dtype="string").str.extract(r'[ab](\d)', expand=False)
   ....:
Out[98]:
0       1
1       2
2    <NA>
dtype: string
```

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

```
In [99]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"],
   ....:               dtype="string")
   ....:

In [100]: s
Out[100]:
A11    a1
B22    b2
C33    c3
dtype: string

In [101]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[101]:
  letter
0      A
1      B
2      C
```

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

```
In [102]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[102]: 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 [103]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[103]:
  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)

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

```
In [104]: s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"],
   .....:                 dtype="string")
   .....:

In [105]: s
Out[105]:
A    a1a2
B      b1
C      c1
dtype: string

In [106]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'

In [107]: s.str.extract(two_groups, expand=True)
Out[107]:
  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 [108]: s.str.extractall(two_groups)
Out[108]:
        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 [109]: s = pd.Series(['a3', 'b3', 'c2'], dtype="string")

In [110]: s
Out[110]:
0    a3
1    b3
```

(continues on next page)

(continued from previous page)

```
2    c2
dtype: string
```

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

```
In [111]: extract_result = s.str.extract(two_groups, expand=True)

In [112]: extract_result
Out[112]:
  letter digit
0      a     3
1      b     3
2      c     2

In [113]: extractall_result = s.str.extractall(two_groups)

In [114]: extractall_result
Out[114]:
        letter digit
  match
0 0          a     3
1 0          b     3
2 0          c     2

In [115]: extractall_result.xs(0, level="match")
Out[115]:
  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).

```
In [116]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
Out[116]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1

In [117]: pd.Series(["a1a2", "b1", "c1"], dtype="string").str.extractall(two_groups)
Out[117]:
        letter digit
  match
0 0          a     1
  1          a     2
1 0          b     1
2 0          c     1
```

### 2.6.7 Testing for Strings that match or contain a pattern

You can check whether elements contain a pattern:

```
In [118]: pattern = r'[0-9][a-z]'

In [119]: pd.Series(['1', '2', '3a', '3b', '03c'],
   .....:           dtype="string").str.contains(pattern)
   .....:
Out[119]:
0    False
1    False
2     True
3     True
4     True
dtype: boolean
```

Or whether elements match a pattern:

```
In [120]: pd.Series(['1', '2', '3a', '3b', '03c'],
   .....:           dtype="string").str.match(pattern)
   .....:
Out[120]:
0    False
1    False
2     True
3     True
4    False
dtype: boolean
```

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 [121]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat
↪'],
   .....:                dtype="string")
   .....:

In [122]: s4.str.contains('A', na=False)
Out[122]:
0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
dtype: boolean
```

### 2.6.8 Creating indicator variables

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

```
In [123]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'], dtype="string")

In [124]: s.str.get_dummies(sep='|')
Out[124]:
   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`.

```
In [125]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])

In [126]: idx.str.get_dummies(sep='|')
Out[126]:
MultiIndex([(1, 0, 0),
            (1, 1, 0),
            (0, 0, 0),
            (1, 0, 1)],
           names=['a', 'b', 'c'])
```

See also *get_dummies()*.

### 2.6.9 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 |
| *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 |

continues on next page