(continued from previous page)

```
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.date_range(start='2014-08-01 09:00', freq='H',
...                     periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
                dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
                dtype='datetime64[ns]', freq='H')
```

pandas.Series.dt.normalize

`Series.dt.`**`normalize`**(*self*, **args*, ***kwargs*)

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on Datetime Array/Index.

Returns

DatetimeArray, DatetimeIndex or Series The same type as the original data. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

floor Floor the datetimes to the specified freq.
ceil Ceil the datetimes to the specified freq.
round Round the datetimes to the specified freq.

Examples

```
>>> idx = pd.date_range(start='2014-08-01 10:00', freq='H',
...                     periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
                dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
              dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

pandas.Series.dt.strftime

Series.dt.**strftime**(*self*, **args*, ***kwargs*)

Convert to Index using specified date_format.

Return an Index of formatted strings specified by date_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

Parameters

date_format [str] Date format string (e.g. "%Y-%m-%d").

Returns

ndarray NumPy ndarray of formatted strings.

See also:

to_datetime Convert the given argument to datetime.
DatetimeIndex.normalize Return DatetimeIndex with times to midnight.
DatetimeIndex.round Round the DatetimeIndex to the specified freq.
DatetimeIndex.floor Floor the DatetimeIndex to the specified freq.

Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

pandas.Series.dt.round

Series.dt.**round**(*self*, **args*, ***kwargs*)

Perform round operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the *freq* cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.round("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.floor

Series.dt.**floor**(*self*, **args*, ***kwargs*)

Perform floor operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time
- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the *freq* cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.floor("H")
0   2018-01-01 11:00:00
1   2018-01-01 12:00:00
2   2018-01-01 12:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.ceil

`Series.dt.`**`ceil`**(*self*, **args*, ***kwargs*)

Perform ceil operation on the data to the specified *freq*.

Parameters

freq [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end). See *frequency aliases* for a list of possible *freq* values.

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise'] Only relevant for DatetimeIndex:

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times.

New in version 0.24.0.

nonexistent ['shift_forward', 'shift_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift_forward' will shift the nonexistent time forward to the closest existing time

- 'shift_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an NonExistentTimeError if there are nonexistent times.

New in version 0.24.0.

Returns

DatetimeIndex, TimedeltaIndex, or Series Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

Raises

ValueError if the *freq* cannot be converted.

Examples

DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Series

```
>>> pd.Series(rng).dt.ceil("H")
0   2018-01-01 12:00:00
1   2018-01-01 12:00:00
2   2018-01-01 13:00:00
dtype: datetime64[ns]
```

pandas.Series.dt.month_name

`Series.dt.`**`month_name`**(*self*, **args*, ***kwargs*)

Return the month names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the month name. Default is English locale.

Returns

Index Index of month names.

Examples

```
>>> idx = pd.date_range(start='2018-01', freq='M', periods=3)
>>> idx
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31'],
              dtype='datetime64[ns]', freq='M')
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

pandas.Series.dt.day_name

Series.dt.**day_name**(*self*, **args*, ***kwargs*)

Return the day names of the DateTimeIndex with specified locale.

New in version 0.23.0.

Parameters

locale [str, optional] Locale determining the language in which to return the day name. Default is English locale.

Returns

Index Index of day names.

Examples

```
>>> idx = pd.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03'],
              dtype='datetime64[ns]', freq='D')
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

Period properties

Series.dt.qyear
Series.dt.start_time
Series.dt.end_time

pandas.Series.dt.qyear

Series.dt.**qyear**

pandas.Series.dt.start_time

Series.dt.**start_time**

pandas.Series.dt.end_time

Series.dt.**end_time**

Timedelta properties

Series.dt.days	Number of days for each element.
Series.dt.seconds	Number of seconds (>= 0 and less than 1 day) for each element.
Series.dt.microseconds	Number of microseconds (>= 0 and less than 1 second) for each element.
Series.dt.nanoseconds	Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.
Series.dt.components	Return a Dataframe of the components of the Timedeltas.

pandas.Series.dt.days

Series.dt.**days**
Number of days for each element.

pandas.Series.dt.seconds

Series.dt.**seconds**
Number of seconds (>= 0 and less than 1 day) for each element.

pandas.Series.dt.microseconds

Series.dt.**microseconds**
Number of microseconds (>= 0 and less than 1 second) for each element.

pandas.Series.dt.nanoseconds

Series.dt.**nanoseconds**
Number of nanoseconds (>= 0 and less than 1 microsecond) for each element.

pandas.Series.dt.components

Series.dt.**components**

Return a Dataframe of the components of the Timedeltas.

Returns

DataFrame

Examples

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='s'))
>>> s
0   00:00:00
1   00:00:01
2   00:00:02
3   00:00:03
4   00:00:04
dtype: timedelta64[ns]
>>> s.dt.components
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     0      0        0        0             0             0            0
1     0      0        0        1             0             0            0
2     0      0        0        2             0             0            0
3     0      0        0        3             0             0            0
4     0      0        0        4             0             0            0
```

Timedelta methods

Series.dt.to_pytimedelta(self)	Return an array of native *datetime.timedelta* objects.
Series.dt.total_seconds(self, *args, **kwargs)	Return total duration of each element expressed in seconds.

pandas.Series.dt.to_pytimedelta

Series.dt.**to_pytimedelta**(*self*)

Return an array of native *datetime.timedelta* objects.

Python's standard *datetime* library uses a different representation timedelta's. This method converts a Series of pandas Timedeltas to *datetime.timedelta* format with the same length as the original Series.

Returns

numpy.ndarray Array of 1D containing data with *datetime.timedelta* type.

See also:

datetime.timedelta

Examples

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0   0 days
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.to_pytimedelta()
array([datetime.timedelta(0), datetime.timedelta(1),
       datetime.timedelta(2), datetime.timedelta(3),
       datetime.timedelta(4)], dtype=object)
```

pandas.Series.dt.total_seconds

Series.dt.**total_seconds**(*self*, **args*, ***kwargs*)

Return total duration of each element expressed in seconds.

This method is available directly on TimedeltaArray, TimedeltaIndex and on Series containing timedelta values under the `.dt` namespace.

Returns

seconds [[ndarray, Float64Index, Series]] When the calling object is a TimedeltaArray, the return type is ndarray. When the calling object is a TimedeltaIndex, the return type is a Float64Index. When the calling object is a Series, the return type is Series of type *float64* whose index is the same as the original.

See also:

datetime.timedelta.total_seconds Standard library version of this method.
TimedeltaIndex.components Return a DataFrame with components of each Timedelta.

Examples

Series

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0   0 days
1   1 days
2   2 days
3   3 days
4   4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.total_seconds()
0         0.0
1     86400.0
2    172800.0
3    259200.0
4    345600.0
dtype: float64
```

TimedeltaIndex

```
>>> idx = pd.to_timedelta(np.arange(5), unit='d')
>>> idx
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

```
>>> idx.total_seconds()
Float64Index([0.0, 86400.0, 172800.0, 259200.00000000003, 345600.0],
             dtype='float64')
```

String handling

`Series.str` can be used to access the values of the series as strings and apply several methods to it. These can be accessed like `Series.str.<function/property>`.

Series.str.capitalize(self)	Convert strings in the Series/Index to be capitalized.
Series.str.casefold(self)	Convert strings in the Series/Index to be casefolded.
Series.str.cat(self[, others, sep, na_rep, join])	Concatenate strings in the Series/Index with given separator.
Series.str.center(self, width[, fillchar])	Filling left and right side of strings in the Series/Index with an additional character.
Series.str.contains(self, pat[, case, ...])	Test if pattern or regex is contained within a string of a Series or Index.
Series.str.count(self, pat[, flags])	Count occurrences of pattern in each string of the Series/Index.
Series.str.decode(self, encoding[, errors])	Decode character string in the Series/Index using indicated encoding.
Series.str.encode(self, encoding[, errors])	Encode character string in the Series/Index using indicated encoding.
Series.str.endswith(self, pat[, na])	Test if the end of each string element matches a pattern.
Series.str.extract(self, pat[, flags, expand])	Extract capture groups in the regex *pat* as columns in a DataFrame.
Series.str.extractall(self, pat[, flags])	For each subject string in the Series, extract groups from all matches of regular expression pat.
Series.str.find(self, sub[, start, end])	Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].
Series.str.findall(self, pat[, flags])	Find all occurrences of pattern or regular expression in the Series/Index.
Series.str.get(self, i)	Extract element from each component at specified position.
Series.str.index(self, sub[, start, end])	Return lowest indexes in each strings where the substring is fully contained between [start:end].
Series.str.join(self, sep)	Join lists contained as elements in the Series/Index with passed delimiter.
Series.str.len(self)	Compute the length of each element in the Series/Index.
Series.str.ljust(self, width[, fillchar])	Filling right side of strings in the Series/Index with an additional character.
Series.str.lower(self)	Convert strings in the Series/Index to lowercase.
Series.str.lstrip(self[, to_strip])	Remove leading and trailing characters.

continues on next page

Table 47 – continued from previous page

Series.str.match(self, pat[, case, flags, na])	Determine if each string matches a regular expression.
Series.str.normalize(self, form)	Return the Unicode normal form for the strings in the Series/Index.
Series.str.pad(self, width[, side, fillchar])	Pad strings in the Series/Index up to width.
Series.str.partition(self[, sep, expand])	Split the string at the first occurrence of *sep*.
Series.str.repeat(self, repeats)	Duplicate each string in the Series or Index.
Series.str.replace(self, pat, repl[, n, ...])	Replace occurrences of pattern/regex in the Series/Index with some other string.
Series.str.rfind(self, sub[, start, end])	Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].
Series.str.rindex(self, sub[, start, end])	Return highest indexes in each strings where the substring is fully contained between [start:end].
Series.str.rjust(self, width[, fillchar])	Filling left side of strings in the Series/Index with an additional character.
Series.str.rpartition(self[, sep, expand])	Split the string at the last occurrence of *sep*.
Series.str.rstrip(self[, to_strip])	Remove leading and trailing characters.
Series.str.slice(self[, start, stop, step])	Slice substrings from each element in the Series or Index.
Series.str.slice_replace(self[, start, ...])	Replace a positional slice of a string with another value.
Series.str.split(self[, pat, n, expand])	Split strings around given separator/delimiter.
Series.str.rsplit(self[, pat, n, expand])	Split strings around given separator/delimiter.
Series.str.startswith(self, pat[, na])	Test if the start of each string element matches a pattern.
Series.str.strip(self[, to_strip])	Remove leading and trailing characters.
Series.str.swapcase(self)	Convert strings in the Series/Index to be swapcased.
Series.str.title(self)	Convert strings in the Series/Index to titlecase.
Series.str.translate(self, table)	Map all characters in the string through the given mapping table.
Series.str.upper(self)	Convert strings in the Series/Index to uppercase.
Series.str.wrap(self, width, **kwargs)	Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given width.
Series.str.zfill(self, width)	Pad strings in the Series/Index by prepending '0' characters.
Series.str.isalnum(self)	Check whether all characters in each string are alphanumeric.
Series.str.isalpha(self)	Check whether all characters in each string are alphabetic.
Series.str.isdigit(self)	Check whether all characters in each string are digits.
Series.str.isspace(self)	Check whether all characters in each string are whitespace.
Series.str.islower(self)	Check whether all characters in each string are lowercase.
Series.str.isupper(self)	Check whether all characters in each string are uppercase.
Series.str.istitle(self)	Check whether all characters in each string are titlecase.
Series.str.isnumeric(self)	Check whether all characters in each string are numeric.
Series.str.isdecimal(self)	Check whether all characters in each string are decimal.
Series.str.get_dummies(self[, sep])	Split each string in the Series by sep and return a DataFrame of dummy/indicator variables.

pandas.Series.str.capitalize

Series.str.**capitalize**(*self*)

Convert strings in the Series/Index to be capitalized.

Equivalent to `str.capitalize()`.

Returns

Series or Index of object

See also:

Series.str.lower	Converts all characters to lowercase.
Series.str.upper	Converts all characters to uppercase.
Series.str.title	Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize	Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase	Converts uppercase to lowercase and lowercase to uppercase.
Series.str.casefold	Removes all case distinctions in the string.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.casefold

Series.str.**casefold**(*self*)

Convert strings in the Series/Index to be casefolded.

New in version 0.25.0.

Equivalent to `str.casefold()`.

Returns

Series or Index of object

See also:

Series.str.lower Converts all characters to lowercase.
Series.str.upper Converts all characters to uppercase.
Series.str.title Converts first character of each word to uppercase and remaining to lowercase.
Series.str.capitalize Converts first character to uppercase and remaining to lowercase.
Series.str.swapcase Converts uppercase to lowercase and lowercase to uppercase.
Series.str.casefold Removes all case distinctions in the string.

Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                 lower
1              CAPITALS
2    this is a sentence
3              SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0                 lower
1              capitals
2    this is a sentence
3              swapcase
dtype: object
```

```
>>> s.str.upper()
0                 LOWER
1              CAPITALS
2    THIS IS A SENTENCE
3              SWAPCASE
dtype: object
```

```
>>> s.str.title()
0                 Lower
```

(continues on next page)

(continued from previous page)

```
1              Capitals
2    This Is A Sentence
3              Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0                 Lower
1              Capitals
2    This is a sentence
3              Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0                 LOWER
1              capitals
2    THIS IS A SENTENCE
3              sWaPcAsE
dtype: object
```

pandas.Series.str.cat

`Series.str.cat`(*self*, *others=None*, *sep=None*, *na_rep=None*, *join='left'*)

Concatenate strings in the Series/Index with given separator.

If *others* is specified, this function concatenates the Series/Index and elements of *others* element-wise. If *others* is not passed, then all values in the Series/Index are concatenated into a single string with a given *sep*.

Parameters

others [Series, Index, DataFrame, np.ndarray or list-like] Series, Index, DataFrame, np.ndarray (one- or two-dimensional) and other list-likes of strings must have the same length as the calling Series/Index, with the exception of indexed objects (i.e. Series/Index/DataFrame) if *join* is not None.

If others is a list-like that contains a combination of Series, Index or np.ndarray (1-dim), then all elements will be unpacked and must satisfy the above criteria individually.

If others is None, the method returns the concatenation of all strings in the calling Series/Index.

sep [str, default ‘’] The separator between the different elements/columns. By default the empty string *‘’* is used.

na_rep [str or None, default None] Representation that is inserted for all missing values:

- If *na_rep* is None, and *others* is None, missing values in the Series/Index are omitted from the result.
- If *na_rep* is None, and *others* is not None, a row containing a missing value in any of the columns (before concatenation) will have a missing value in the result.

join [{‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘left’] Determines the join-style between the calling Series/Index and any Series/Index/DataFrame in *others* (objects without an index need to match the length of the calling Series/Index). To disable alignment, use *.values* on any Series/Index/DataFrame in *others*.

New in version 0.23.0.

Changed in version 1.0.0: Changed default of *join* from None to *‘left’*.

Returns

str, Series or Index If *others* is None, *str* is returned, otherwise a *Series/Index* (same type as caller) of objects is returned.

See also:

split Split each string in the Series/Index.
join Join lists contained as elements in the Series/Index.

Examples

When not passing *others*, all values are concatenated into a single string:

```
>>> s = pd.Series(['a', 'b', np.nan, 'd'])
>>> s.str.cat(sep=' ')
'a b d'
```

By default, NA values in the Series are ignored. Using *na_rep*, they can be given a representation:

```
>>> s.str.cat(sep=' ', na_rep='?')
'a b ? d'
```

If *others* is specified, corresponding values are concatenated with the separator. Result will be a Series of strings.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',')
0    a,A
1    b,B
2    NaN
3    d,D
dtype: object
```

Missing values will remain missing in the result, but can again be represented using *na_rep*

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',', na_rep='-')
0    a,A
1    b,B
2    -,C
3    d,D
dtype: object
```

If *sep* is not specified, the values are concatenated without separation.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], na_rep='-')
0    aA
1    bB
2    -C
3    dD
dtype: object
```

Series with different indexes can be aligned before concatenation. The *join*-keyword works as in other methods.

```
>>> t = pd.Series(['d', 'a', 'e', 'c'], index=[3, 0, 4, 2])
>>> s.str.cat(t, join='left', na_rep='-')
0    aa
1    b-
2    -c
3    dd
```

(continues on next page)

(continued from previous page)

```
dtype: object
>>>
>>> s.str.cat(t, join='outer', na_rep='-')
0    aa
1    b-
2    -c
3    dd
4    -e
dtype: object
>>>
>>> s.str.cat(t, join='inner', na_rep='-')
0    aa
2    -c
3    dd
dtype: object
>>>
>>> s.str.cat(t, join='right', na_rep='-')
3    dd
0    aa
4    -e
2    -c
dtype: object
```

For more examples, see *here*.

pandas.Series.str.center

`Series.str.`**`center`**(*self*, *width*, *fillchar=' '*)

Filling left and right side of strings in the Series/Index with an additional character. Equivalent to `str.center()`.

Parameters

width [int] Minimum width of resulting string; additional characters will be filled with `fillchar`.

fillchar [str] Additional character for filling, default is whitespace.

Returns

filled [Series/Index of objects.]

pandas.Series.str.contains

`Series.str.`**`contains`**(*self*, *pat*, *case=True*, *flags=0*, *na=nan*, *regex=True*)

Test if pattern or regex is contained within a string of a Series or Index.

Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

Parameters

pat [str] Character sequence or regular expression.

case [bool, default True] If True, case sensitive.

flags [int, default 0 (no flags)] Flags to pass through to the re module, e.g. re.IGNORECASE.

na [default NaN] Fill value for missing values.

regex [bool, default True] If True, assumes the pat is a regular expression.

If False, treats the pat as a literal string.

Returns

Series or Index of boolean values A Series or Index of boolean values indicating whether the given pattern is contained within the string of each element of the Series or Index.

See also:

match Analogous, but stricter, relying on re.match instead of re.search.
Series.str.startswith Test if the start of each string element matches a pattern.
Series.str.endswith Same as startswith, but tests the end of string.

Examples

Returning a Series of booleans using only a literal pattern.

```
>>> s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s1.str.contains('og', regex=False)
0    False
1     True
2    False
3    False
4      NaN
dtype: object
```

Returning an Index of booleans using only a literal pattern.

```
>>> ind = pd.Index(['Mouse', 'dog', 'house and parrot', '23.0', np.NaN])
>>> ind.str.contains('23', regex=False)
Index([False, False, False, True, nan], dtype='object')
```

Specifying case sensitivity using *case*.

```
>>> s1.str.contains('oG', case=True, regex=True)
0    False
1    False
2    False
3    False
4      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN* replaces NaN values with *False*. If Series or Index does not contain NaN values the resultant dtype will be *bool*, otherwise, an *object* dtype.

```
>>> s1.str.contains('og', na=False, regex=True)
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

Returning ‘house’ or ‘dog’ when either expression occurs in a string.

```
>>> s1.str.contains('house|dog', regex=True)
0    False
1     True
2     True
3    False
4      NaN
dtype: object
```

Ignoring case sensitivity using *flags* with regex.

```
>>> import re
>>> s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0    False
1    False
2     True
3    False
4      NaN
dtype: object
```

Returning any digit using regular expression.

```
>>> s1.str.contains('\d', regex=True)
0    False
1    False
2    False
3     True
4      NaN
dtype: object
```

Ensure *pat* is a not a literal pattern when *regex* is set to True. Note in the following example one might expect only *s2[1]* and *s2[3]* to return *True*. However, '.0' as a regex matches any character followed by a 0.

```
>>> s2 = pd.Series(['40', '40.0', '41', '41.0', '35'])
>>> s2.str.contains('.0', regex=True)
0     True
1     True
2    False
3     True
4    False
dtype: bool
```

pandas.Series.str.count

`Series.str.`**`count`**(*self*, *pat*, *flags=0*, ***kwargs*)

Count occurrences of pattern in each string of the Series/Index.

This function is used to count the number of times a particular regex pattern is repeated in each of the string elements of the *Series*.

Parameters

pat [str] Valid regular expression.

flags [int, default 0, meaning no flags] Flags for the *re* module. For a complete list, see here.

****kwargs** For compatibility with other string methods. Not used.

Returns

Series or Index Same type as the calling object containing the integer counts.

See also:

`re` Standard library module for regular expressions.
`str.count` Standard library version, without regular expression support.

Notes

Some characters need to be escaped when passing in *pat*. eg. `'$'` has a special meaning in regex and must be escaped when finding this literal character.

Examples

```
>>> s = pd.Series(['A', 'B', 'Aaba', 'Baca', np.nan, 'CABA', 'cat'])
>>> s.str.count('a')
0    0.0
1    0.0
2    2.0
3    2.0
4    NaN
5    0.0
6    1.0
dtype: float64
```

Escape `'$'` to find the literal dollar sign.

```
>>> s = pd.Series(['$', 'B', 'Aab$', '$$ca', 'C$B$', 'cat'])
>>> s.str.count('\$')
0    1
1    0
2    1
3    2
4    2
5    0
dtype: int64
```

This is also available on Index

```
>>> pd.Index(['A', 'A', 'Aaba', 'cat']).str.count('a')
Int64Index([0, 0, 2, 1], dtype='int64')
```

pandas.Series.str.decode

`Series.str.`**`decode`**(*self*, *encoding*, *errors='strict'*)

Decode character string in the Series/Index using indicated encoding. Equivalent to `str.decode()` in python2 and `bytes.decode()` in python3.

Parameters

encoding [str]

errors [str, optional]

Returns

Series or Index

pandas.Series.str.encode

Series.str.**encode**(*self*, *encoding*, *errors='strict'*)

Encode character string in the Series/Index using indicated encoding. Equivalent to `str.encode()`.

Parameters

encoding [str]

errors [str, optional]

Returns

encoded [Series/Index of objects]

pandas.Series.str.endswith

Series.str.**endswith**(*self*, *pat*, *na=nan*)

Test if the end of each string element matches a pattern.

Equivalent to `str.endswith()`.

Parameters

pat [str] Character sequence. Regular expressions are not accepted.

na [object, default NaN] Object shown if element tested is not a string.

Returns

Series or Index of bool A Series of booleans indicating whether the given pattern matches the end of each string element.

See also:

str.endswith Python standard library string method.
Series.str.startswith Same as endswith, but tests the start of string.
Series.str.contains Tests if string element contains a pattern.

Examples

```
>>> s = pd.Series(['bat', 'bear', 'caT', np.nan])
>>> s
0     bat
1    bear
2     caT
3     NaN
dtype: object
```

```
>>> s.str.endswith('t')
0     True
1    False
2    False
3      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN*.

```
>>> s.str.endswith('t', na=False)
0     True
1    False
2    False
3    False
dtype: bool
```

pandas.Series.str.extract

Series.str.**extract**(*self*, *pat*, *flags=0*, *expand=True*)

Extract capture groups in the regex *pat* as columns in a DataFrame.

For each subject string in the Series, extract groups from the first match of regular expression *pat*.

Parameters

pat [str] Regular expression pattern with capturing groups.

flags [int, default 0 (no flags)] Flags from the `re` module, e.g. `re.IGNORECASE`, that modify regular expression matching for things like case, spaces, etc. For more details, see `re`.

expand [bool, default True] If True, return DataFrame with one column per capture group. If False, return a Series/Index if there is one capture group or DataFrame if there are multiple capture groups.

Returns

DataFrame or Series or Index A DataFrame with one row for each subject string, and one column for each group. Any capture group names in regular expression pat will be used for column names; otherwise capture group numbers will be used. The dtype of each result column is always object, even when no match is found. If `expand=False` and pat has only one capture group, then return a Series (if subject is a Series) or Index (if subject is an Index).

See also:

extractall Returns all matches (not just the first match).

Examples

A pattern with two groups will return a DataFrame with two columns. Non-matches will be NaN.

```
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.extract(r'([ab])(\d)')
     0    1
0    a    1
1    b    2
2  NaN  NaN
```

A pattern may contain optional groups.

```
>>> s.str.extract(r'([ab])?(\d)')
     0  1
0    a  1
1    b  2
2  NaN  3
```

Named groups will become column names in the result.

```
>>> s.str.extract(r'(?P<letter>[ab])(?P<digit>\d)')
  letter digit
0      a     1
1      b     2
2    NaN   NaN
```

A pattern with one group will return a DataFrame with one column if expand=True.

```
>>> s.str.extract(r'[ab](\d)', expand=True)
     0
0    1
1    2
2  NaN
```

A pattern with one group will return a Series if expand=False.

```
>>> s.str.extract(r'[ab](\d)', expand=False)
0      1
1      2
2    NaN
dtype: object
```

pandas.Series.str.extractall

`Series.str.`**`extractall`**(*self*, *pat*, *flags=0*)

For each subject string in the Series, extract groups from all matches of regular expression pat. When each subject string in the Series has exactly one match, extractall(pat).xs(0, level='match') is the same as extract(pat).

Parameters

pat [str] Regular expression pattern with capturing groups.

flags [int, default 0 (no flags)] A `re` module flag, for example `re.IGNORECASE`. These allow to modify regular expression matching for things like case, spaces, etc. Multiple flags can be combined with the bitwise OR operator, for example `re.IGNORECASE | re.MULTILINE`.

Returns

DataFrame A `DataFrame` with one row for each match, and one column for each group. Its rows have a `MultiIndex` with first levels that come from the subject `Series`. The last level is named 'match' and indexes the matches in each item of the `Series`. Any capture group names in regular expression pat will be used for column names; otherwise capture group numbers will be used.

See also:

`extract` Returns first match only (not all matches).

Examples

A pattern with one group will return a DataFrame with one column. Indices with no matches will not appear in the result.

```
>>> s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])
>>> s.str.extractall(r"[ab](\d)")
        0
  match
A 0     1
  1     2
B 0     1
```

Capture group names are used for column names of the result.

```
>>> s.str.extractall(r"[ab](?P<digit>\d)")
        digit
  match
A 0         1
  1         2
B 0         1
```

A pattern with two groups will return a DataFrame with two columns.

```
>>> s.str.extractall(r"(?P<letter>[ab])(?P<digit>\d)")
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
```

Optional groups that do not match are NaN in the result.

```
>>> s.str.extractall(r"(?P<letter>[ab])?(?P<digit>\d)")
        letter digit
  match
A 0          a     1
  1          a     2
B 0          b     1
C 0        NaN     1
```

pandas.Series.str.find

Series.str.**find**(*self*, *sub*, *start=0*, *end=None*)

Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard `str.find()`.

Parameters

sub [str] Substring being searched.

start [int] Left edge index.

end [int] Right edge index.

Returns

Series or Index of int.

See also:

rfind Return highest indexes in each strings.

pandas.Series.str.findall

Series.str.**findall**(*self*, *pat*, *flags=0*, ***kwargs*)

Find all occurrences of pattern or regular expression in the Series/Index.

Equivalent to applying `re.findall()` to all the elements in the Series/Index.

Parameters

pat [str] Pattern or regular expression.

flags [int, default 0] Flags from `re` module, e.g. *re.IGNORECASE* (default is 0, which means no flags).

Returns

Series/Index of lists of strings All non-overlapping matches of pattern or regular expression in each string of this Series/Index.

See also:

count Count occurrences of pattern or regular expression in each string of the Series/Index.

extractall For each string in the Series, extract groups from all matches of regular expression and return a DataFrame with one row for each match and one column for each group.

`re.findall` The equivalent `re` function to all non-overlapping matches of pattern or regular expression in string, as a list of strings.

Examples

```
>>> s = pd.Series(['Lion', 'Monkey', 'Rabbit'])
```

The search for the pattern 'Monkey' returns one match:

```
>>> s.str.findall('Monkey')
0          []
1    [Monkey]
2          []
dtype: object
```

On the other hand, the search for the pattern 'MONKEY' doesn't return any match:

```
>>> s.str.findall('MONKEY')
0    []
1    []
2    []
dtype: object
```

Flags can be added to the pattern or regular expression. For instance, to find the pattern 'MONKEY' ignoring the case:

```
>>> import re
>>> s.str.findall('MONKEY', flags=re.IGNORECASE)
0          []
1    [Monkey]
2          []
dtype: object
```