25 Pandas Functions You Didn't Know Existed

This notebook is a practice walkalong for the **TalkPythonToMe** Podcast titled **25 Pandas Functions You Didn't Know Existed.**

Whie listening to the podcast the Pandas API documentation is consulted for examples to demonstrate and practice the functions mentioned on the podcast which featured the author of the medium.com article - Bex Tuychiev as guest.

List of Pandas Function:

1. ExcelWriter
2. pipe
3. factorize
4. explode
5. Squeeze
6. Between
7. Transpose (T)
8. Style
9. Options
10. Convert_dtypes
11. Select_dtypes
12. Mask
13. Min and Max
14. nlargest and nsmallest
15. idxmax and idxmin
16. value_counts
17. clip
18. at_time
19. bdate_range
20. autocorr
21. hasnans
22. at and iat
23. argsort

24. cat
25. GroupBy.nth

Link to the podcast: https://podcasts.apple.com/gb/podcast/talk-python-to-me/id979020229?i=1000542266841

In [1]:
```
import pandas as pd
import numpy as np
```

ExcelWriter

class **pandas.ExcelWriter(path, engine=None, date_format=None, datetime_format=None, mode='w', storage_options=None, if_sheet_exists=None, engine_kwargs=None)**

Class for writing DataFrame objects into excel files

Default usage

In [2]:
```
df = pd.DataFrame([['ABC', 'XYZ']], columns=['Foo', 'Bar'])
df
```

In [3]:
```
with pd.ExcelWriter('data/path_to_file.xlsx') as writer:
    df.to_excel(writer)
```

To write to separate sheets in a single file:

In [4]:
```
df1 = pd.DataFrame([['AAA', 'BBB']], columns=['Spam', 'Egg'])
df2 = pd.DataFrame([['ABC', 'XTY']], columns=['Foo', 'Bar'])
with pd.ExcelWriter('data/path_to_file1.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet_1')
    df2.to_excel(writer, sheet_name='Sheet_2')
```

You can set the date format or datetime format:

In [5]:
```
from datetime import date, datetime
df = pd.DataFrame(
    [
        [date(2014, 1, 31), date(1999, 9, 24)],
        [datetime(1998, 5, 26, 23, 33, 4), datetime(2014, 2, 28, 13, 5, 13 )],
    ],
    index=['Date', 'Datetime'], columns=['X', 'Y']
    )

with pd.ExcelWriter(
    'data/path_to_file2.xlsx', date_format='YYYY-MM-DD', datetime_format='YYYY-MM-D
    df.to_excel(writer)
```

You can also append to an existing Excel file:

In [6]:
```
with pd.ExcelWriter('data/path_to_file2.xlsx', mode='a', engine='openpyxl') as writ
    df.to_excel(writer, sheet_name='Sheet_2')
```

Here, the `if_sheet_exists` parameter can be set to replace a sheet if it already exists:

In [7]:
```
with pd.ExcelWriter(
        'data/path_to_file2.xlsx',
        mode='a',
        engine='openpyxl',
        if_sheet_exists='replace') as writer:
    df.to_excel(writer, sheet_name="Sheet1")
```

You can also write multiple DataFrames to a single sheet. Note that the `if_sheet_exists` parameter needs to be set to `overlay`:

In [8]:
```
with pd.ExcelWriter('data/path_to_file.xlsx',
                    mode='a',
                    engine='openpyxl',
                    if_sheet_exists='overlay',
                    ) as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet1', startcol=3)
```

You can store Excel file in RAM

In [9]:
```
import io
df = pd.DataFrame([['ABC', 'XYZ']], columns=['Foo', 'Bar'])
buffer = io.BytesIO()
with pd.ExcelWriter(buffer) as writer:
    df.to_excel(writer)
```

You can pack Excel file into zip archive:

In [10]:
```
import zipfile
df = pd.DataFrame([['ABC', 'XYZ']], columns=['Foo', 'Bar'])
with zipfile.ZipFile('data/file_to_path2.zip', 'w') as zf:
    with zf.open('filename.xlsx', 'w') as buffer:
        with pd.ExcelWriter(buffer) as writer:
            df.to_excel(writer)
```

You can specify additional arguments to the underlying engine:

In [11]:
```
!pip install xlsxwriter
```

In [12]:
```
with pd.ExcelWriter('data/path_to_file4.xlsx',
                    engine='xlsxwriter',
                    engine_kwargs={'options': {'nan_inf_to_errors': True}}
                    ) as writer:
    df.to_excel(writer)
```

In append mode, `engine_kwargs` are passed through to openpyxl's `load_workbook`:

In [13]:
```
with pd.ExcelWriter(
    'data/path_to_file5.xlsx',
    engine='openpyxl',
    engine_kwargs={'keep_vba': True}) as writer:
    df.to_excel(writer, sheet_name='Sheet2')
```

Above is a bug in Pandas

https://github.com/pandas-dev/pandas/issues/45734

pipe

DataFrame.pipe(func, *args,* *kwargs)

Apply chainable functions that expect Series or DataFrames

Parameters:

func: function Function to apply to the Series/DataFrame. args, and kwargs are passed into func. Alternatively a (callable, data_keyword) tuple where data_keyword is a string indicating the keyword of callable that expects the Series/DataFrame.

***args:** iterable, optional

Positional arguments passed into func.

****kwargs:** mapping, optional

A dictionary of keyword arguments passed into func.

Returns:

the return type of func

Example

In [14]:
```
import numpy as np

# Constructing a income DataFrame from a dictionary
data = [[8000, 1000], [9500, np.nan], [5000, 2000]]
df = pd.DataFrame(data, columns=['Salary', 'Others'])
df
```

In [15]:
```
# Functions that perform tax reductions on an income DataFrame
def substract_federal_tax(df):
    return df * 0.9


def substract_state_tax(df, rate):
    return df * (1- rate)


def substract_national_insurance(df, rate, rate_increase):
    new_rate = rate + rate_increase
    return df * (1 - new_rate)
```

In [16]:
```
# instead of writing
substract_national_insurance(
    substract_state_tax(substract_federal_tax(df), rate=0.12),
    rate=0.05,
    rate_increase=0.02)
```

In [17]:
```
# You write
(
    df.pipe(substract_federal_tax)
    .pipe(substract_state_tax, rate=0.12)
    .pipe(substract_national_insurance, rate=0.05, rate_increase=0.02)
)
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `national_insurance` takes its data as `df` in the second argument:

In [18]:
```
def subtract_national_insurance(rate, df, rate_increase):
    new_rate = rate + rate_increase
    return df * (1 - new_rate)
```

In [19]:
```
(
    df.pipe(substract_federal_tax)
    .pipe(substract_state_tax, rate=0.12)
    .pipe(
        (substract_national_insurance, 'df'),
        rate=0.05,
        rate_increase=0.02)
)
```

factorize

pandas.factorize(values, sort=False, use_na_sentinel=True, size_hint=None)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. factorize is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`

Parameters: **values: sequence**

A 1-D sequence. Sequences that aren't pandas objects are coerced to ndarrays before factorization.

sort: bool, default False

Sort uniques and shuffle codes to maintain the relationship.

use_na_sentinel: bool, default True

If True, the sentinel -1 will be used for NaN values. If False, NaN values will be encoded as non-negative integers and will not drop the NaN from the uniques of the values.

In [20]:
```
## Similar to OneHotEncoder in SKlearn
codes, uniques = pd.factorize(np.array(['b', 'b', 'c', 'b', 'a'], dtype='O'))
```

In [21]:
```
codes

# The results are identical for methods like `Series.factorize()`
```

In [22]:
```
uniques
```

With `sort=True` the ***uniques*** will be shuffled so that the relationship is the maintained

In [23]:
```
codes, uniques = pd.factorize(np.array(['b', 'b', 'a', 'c', 'b'], dtype='O'),
                              sort=True)
```

In [24]:
```
codes
```

In [25]:
```
uniques
```

When `use_na_sentinel=True` (the default), missing values are indicated in the ***codes*** with the sentinel value `-1` and missing values are not included in ***uniques***

In [26]:
```
codes, uniques = pd.factorize(np.array(['b', None, 'a', 'c', 'b'], dtype='O'))
```

In [27]:
```
codes
```

In [28]:
```
uniques
```

List are intermally coerced to NumPy arrays. When factorizing pandas objects, the type of uniques will differ. For Categoricals, a ***Categorical*** is returned.

In [29]:
```
cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])

codes, uniques = pd.factorize(cat)
```

In [30]:
```
codes
```

In [31]:
```
uniques
```

Notice that `'b'` is in `uniques.categories`, despite not being present in `cat.values`

For all other pandas objects, an index of the appropriate type is returned

In [32]:
```
cat = pd.Series(['a', 'a', 'c'])

codes, uniques = pd.factorize(cat)
```

In [33]:
```
codes
```

In [34]:
```
uniques
```

If NaN is in the values, and we want to include NaN in the uniques of the values, it can be achieved by setting `use_na_sentinel=False`

In [35]:
```
values = np.array([1, 2, 1, np.nan])

codes, uniques = pd.factorize(values) # default:use_na_sentinel=True

codes, uniques
```

In [36]:
```
codes, uniques = pd.factorize(values, use_na_sentinel=False)
```

In [37]:
```
codes
```

In [38]:
```
uniques
```

explode

DataFrame.explode(column, ignore_index=False)

Transform each element of a list-like to a row, replicating index values.

Parameters:
column: IndexLabel
Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

ignore_index: bool, default False
If True, the resulting index will be labeled 0, 1, …, n - 1.

Returns:
DataFrame Exploded lists to rows of the subset columns; index will be duplicated for these rows.

Raises:
ValueError
-If columns of the frame are not unique.

-If specified columns to explode is empty list.

-If specified columns to explode have not matching count of elements rowwise in the frame.

In [39]:
```
# Create a DataFrame
df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
                   'B': 1,
```

```
                   'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})

df
```

Single-column explode

In [40]:
```
df.explode('A')
```

Multi-column explode

In [41]:
```
df.explode(list('AC'))
```

squeeze

DataFrame.squeeze(axis=None)

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call squeeze to ensure you have a Series.

Parameters:
axis:{0 or 'index', 1 or 'columns', None}, default None
A specific axis to squeeze. By default, all length-1 axes are squeezed. For Series this parameter is unused and defaults to None.

Returns:
DataFrame, Series, or scalar
The projection after squeezing axis or all the axes.

In [42]:
```
primes = pd.Series([2, 3, 5, 7])
```

In [43]:
```
even_primes = primes[primes % 2 == 0]
```

In [44]:
```
even_primes
```

In [45]:
```
type(even_primes)
```

In [46]:
```
even_primes.squeeze()
```

Squeezing objects with more than one value in every axis does nothing

In [47]:
```
odd_primes = primes[primes % 2 == 1]
```

In [48]:
```
odd_primes
```

In [49]:
```
odd_primes.squeeze()
```

Squeezing is even more effective when used with DataFrames

In [50]:
```
df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
df
```

In [51]:
```
# Slicing a single column will produce a DataFrame with columns having only one val
df_a = df[['a']]
df_a
```

In [52]:
```
type(df_a)
```

In [53]:
```
# So the column can be squeezed down, resulting in a Series
df_s = df_a.squeeze()
type(df_s)
```

In [54]:
```
df_s
```

In [55]:
```
# Slicing a single row from the single column will produce a single scalar DataFram
df_0a = df.loc[df.index < 1, ['a']]
df_0a
```

In [56]:
```
# Squeezing the rows produces a sihgle scalar Series
df_0a.squeeze('rows')
```

In [57]:
```
# Squeezing all axes will project directly into a scalar
df_0a.squeeze()
```

between

Series.between(left, right, inclusive='both')

Returns boolean Series equivalent to left <= series <= right

This function returns a boolean vector containing `True` wherever the corresponding Series element is between the boundary values `left` and `right`. NA values are treated as `False`.

Parameters: **left:** ***scalar or list-like*** Left boundary.

right: ***scalar or list-like*** Right boundary.

inclusive: ***{"both", "neither", "left", "right"}*** Include boundaries. Whether to set each bound as closed or open.

In [58]:
```
s = pd.Series([2, 0, 4, 8, np.nan])
```

Bounday values are included by default - `inclusive` = `both`

In [59]:
```
s.between(1, 4)
```

In [60]:
```
# With `inclusive` set to `neither` boundary values are excluded:
s.between(1, 4, inclusive='neither')
```

In [61]:
```
# `left` and `right` can be any scalar value:
s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
```

In [62]:
```
s.between('Anna', 'Daniel')
```

In [63]:
```
s = pd.Series(['Anna', 'Alice', 'Bob', 'Carol', 'Eve'])
```

In [64]:
```
s.between('Anna', 'Daniel')
```

In [65]:
```
df = pd.read_csv('data/uk_accidents_pd.csv')
```

In [66]:
```
# pd.sample() can be used instead of head() or tail()
df.sample(10)
```

Transpose

DataFrame.transpose(*args, copy=False)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property T is an accessor to the method transpose().

Parameters: *argstuple, optional Accepted for compatibility with NumPy.

copy: *bool, default False* Whether to copy the data after transposing, even for DataFrames with a single dtype.

```
Note that a copy is always required for mixed dtype DataFrames, or
for DataFrames with any extension types.
```

Returns: DataFrame The transposed DataFrame.

In [67]:
```
df.describe()
```

In [68]:
```
df.describe().T
```

style

pandas.DataFrame.style

***property* DataFrame.style**
Returns a Styler object.

Contains methods for building a styled HTML representation of the DataFrame.

See also

io.formats.style.Styler Helps style a DataFrame or Series according to the data with HTML and CSS.

In [69]:
```
df = pd.DataFrame({'A': [1, 2, 3]})
df
```

In [70]:
```
df.style
```

Styling and output display customisation should be performed after the data in a DataFrame has been processed. The Styler is not dynamically updated if further changes to the DataFrame are made. The `DataFrame.style` attribute is a property that returns a Styler object. It has a _repr*html* method defined on it so it is rendered automatically in Jupyter Notebook.

In [71]:
```
import numpy as np
import matplotlib as mpl

df = pd.DataFrame ( {
        'strings': ['Adam', 'Mike'],
        'ints': [1, 3],
        'floats': [1.123, 1000.13]
})

df.style \
    .format(precision=3, thousands='.', decimal=',') \
    .format_index(str.upper, axis=1) \
    .relabel_index(['row1', 'row2'], axis=0)
```

In [72]:
```
weather_df = pd.DataFrame(np.random.rand(10, 2)* 5,
                          index=pd.date_range(start='2021-01-01', periods=10),
                                              columns=['Tokyo', 'Beijing'])


def rain_condition(v):
    if v < 1.75:
        return 'Dry'
    elif v < 2.75:
        return 'Rain'
    return 'Heavy Rain'

def make_pretty(styler):
    styler.set_caption('Weather Conditions')
    styler.format(rain_condition)
    styler.format_index(lambda v: v.strftime('%A'))
    styler.background_gradient(axis=None, vmin=1, vmax=5, cmap='YlGnBu')

    return styler

weather_df
```

In [73]:
```
weather_df.loc['2021-01-04':'2021-01-08'].style.pipe(make_pretty)
```

Pandas style documentation

https://pandas.pydata.org/docs/user_guide/style.html

set_option

pandas.set_option(pat, value) =***<pandas._config.config.CallableDynamicDoc object>
Sets the value of the specified option.***

Available options:

compute.[use_bottleneck, use_numba, use_numexpr]

display.[chop_threshold, colheader_justify, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]

display.html.[border, table_schema, use_mathjax]

display.[large_repr, max_categories, max_columns, max_colwidth, max_dir_items, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, min_rows, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]

display.unicode.[ambiguous_as_wide, east_asian_width]

display.[width]

future.[infer_string]

io.excel.ods.[reader, writer]

io.excel.xls.[reader]

io.excel.xlsb.[reader]

io.excel.xlsm.[reader, writer]

io.excel.xlsx.[reader, writer]

io.hdf.[default_format, dropna_table]

io.parquet.[engine]

io.sql.[engine]

mode.[chained_assignment, copy_on_write, data_manager, sim_interactive, string_storage, use_inf_as_na]

plotting.[backend]

plotting.matplotlib.[register_converters]

styler.format.[decimal, escape, formatter, na_rep, precision, thousands]

styler.html.[mathjax]

styler.latex.[environment, hrules, multicol_align, multirow_align]

styler.render.[encoding, max_columns, max_elements, max_rows, repr]

styler.sparse.[columns, index]

Parameters:
pat: str Regexp which should match a single option. Note: partial matches are supported for

convenience, but unless you use the full option name (e.g. x.y.z.option_name), your code may break in future versions if new options with similar names are introduced.

value: object New value of option.

Raises: OptionError if no such option exists

Pandas set_option documentation
https://pandas.pydata.org/docs/reference/api/pandas.set_option.html

In [74]:
```
pd.set_option('display.max_columns', 2)
```

In [75]:
```
df = pd.DataFrame([[1, 2, 3, 4, 5],
                   [6, 7,8,9,10]])
df
```

In [76]:
```
pd.reset_option('display.max_columns')
```

In [77]:
```
df
```

convert_dtype()

DataFrame.convert_dtypes *(infer_objects=True, convert_string=True, convert_integer=True, convert_boolean=True, convert_floating=True, dtype_backend='numpy_nullable')*

Convert columns to the best possible dtypes using dtypes supporting `pd.NA`

Parameters:
infer_objects: bool, default True Whether object dtypes should be converted to the best possible types.

convert_string: bool, default True Whether object dtypes should be converted to String Dtype().

convert_integer: bool, default True Whether, if possible, conversion can be done to integer extension types.

convert_booleanbool, defaults True Whether object dtypes should be converted to Boolean Dtypes().

convert_floating: bool, defaults True Whether, if possible, conversion can be done to floating extension types. If `convert_integer` is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

dtype_backend{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable' Back-end data type applied to the resultant DataFrame (still experimental)

In [78]:
```
df = pd.DataFrame({
        'a': pd.Series([1, 2, 3], dtype=np.dtype('int32')),
        'b': pd.Series(['x', 'y', 'z'], dtype=np.dtype('O')),
        'c': pd.Series([True, False, np.nan], dtype=np.dtype('O')),
```

```
        'd': pd.Series(['h', 'i', np.nan], dtype=np.dtype('O')),
        'e': pd.Series([10, np.nan, 20], dtype=np.dtype('float')),
        'f': pd.Series([np.nan, 100.5, 200], dtype=np.dtype('float'))
})

df
```

In [79]:
```
df.dtypes
```

In [80]:
```
# Convert the DataFrame to use best possible dtypes
dfn = df.convert_dtypes()
```

In [81]:
```
dfn
```

In [82]:
```
dfn.dtypes
```

In [83]:
```
# Start with a Series of strings and missing data represented by `np.nan`
s = pd. Series(['a', 'b', np.nan])
s
```

In [84]:
```
# Obtain a Series with dtype `StringDtype`
s.convert_dtypes()
```

`convert_dtype` cannot the datetime Dtypes!

select_dtypes()

DataFrame.select_dtypes(include=None, exclude=None)

Return a subset of the DataFrame's columns based on the column dtypes.

Parameters:
include, exclude: *scalar or list-like
A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

Returns:
DataFrame
The subset of the frame including the dtypes in include and excluding the dtypes in exclude.

Raises: ValueError

In [85]:
```
df = pd.DataFrame({
    'a': [1, 2] * 3,
    'b': [True, False] * 3,
    'c': [1.0, 2.0] * 3
})
df
```

In [86]:
```
df.select_dtypes(include='bool')
```

In [87]:
```
df.select_dtypes(include='float64')
```

In [88]:
```
df.select_dtypes(include=['int64'])
```

In [89]:
```
df.select_dtypes(exclude='int64')

# NB: scalar or list-like parameters ('int64' or ['int64'])
```

mask

DataFrame.mask(cond, other=_NoDefault.no_default, *, inplace=False, axis=None, level=None)

Replace values where the condition is True.

Parameters:

cond: ***bool Series/DataFrame, array-like, or callable***

Where cond is False, keep the original value. Where True, replace with corresponding value from other. If cond is callable, it is computed on the Series/DataFrame and should return boolean Series/DataFrame or array. The callable must not change input Series/DataFrame (though pandas doesn't check it).

other: ***scalar, Series/DataFrame, or callable***

Entries where cond is True are replaced with corresponding value from other. If other is callable, it is computed on the Series/DataFrame and should return scalar or Series/DataFrame. The callable must not change input Series/DataFrame (though pandas doesn't check it). If not specified, entries will be filled with the corresponding NULL value (np.nan for numpy dtypes, pd.NA for extension dtypes).

inplace: ***bool, default False***

Whether to perform the operation in place on the data.

axis: ***int, default None***

Alignment axis if needed. For Series this parameter is unused and defaults to 0.

level: ***int, default None***

Alignment level if needed.

Returns:

Same type as caller or None if inplace=True .

In [90]:
```
s = pd.Series(range(5))
```

In [91]:
```
s.where(s>0)
```

In [92]:
```
s.mask(s>0)
```

In [93]:
```
s = pd.Series(range(5))
t = pd.Series([True, False])
s.where(t, 99)
```

In [94]:
```
s.mask(t, 99)
```

In [95]:
```
s.where(s>1, 10)
```

In [96]:
```
s.mask(s>1, 10)
```

In [97]:
```
df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
df
```

In [98]:
```
m = df % 3 == 0
```

In [99]:
```
df.where(m, -df)
```

In [100…
```
df.mask(m, -df)
```

In [101…
```
df.where(m, -df) == np.where(m, df, -df)
```

In [102…
```
df.where(m, -df) == df.mask(~m, -df)
```

min() max()

DataFrame.min(axis=0, skipna=True, numeric_only=False, **kwargs)

Parameters:
axis: {index (0), columns (1)}
Axis for the function to be applied on. For Series this parameter is unused and defaults to 0.

For DataFrames, specifying axis=None will apply the aggregation across both axes.

skipna: bool, default True
Exclude NA/null values when computing the result.

numeric_only: bool, default False
Include only float, int, boolean columns. Not implemented for Series.

****kwargs**
Additional keyword arguments to be passed to the function.

Returns:
Series or scalar

See also `Series.sum`

    Return the sum.

`Series.min`

    Return the minimum.

`Series.max`

    Return the maximum.

`Series.idxmin`

Return the index of the minimum.

`Series.idxmax`

Return the index of the maximum.

`DataFrame.sum`

Return the sum over the requested axis.

`DataFrame.min`

Return the minimum over the requested axis.

`DataFrame.max`

Return the maximum over the requested axis.

`DataFrame.idxmin`

Return the index of the minimum over the requested axis.

`DataFrame.idxmax`

Return the index of the maximum over the requested axis.

In [103…
```
idx = pd.MultiIndex.from_arrays([
    ['warm', 'warm', 'cold', 'cold'],
    ['dog', 'falcon', 'fish', 'spider']],
    names=['blooded', 'animal'])
s= pd.Series([4, 2, 0, 8], name='legs', index=idx)
s
```

In [104…
```
s.min()
```

nlargest() nsmallest

DataFrame.nlargest(n, columns, keep='first')

DataFrame.nsmallest(n, columns, keep='first')

Parameters:

n:int Number of rows to return.

columns: label or list of labels
Column label(s) to order by.

keep:{‘first’, ‘last’, ‘all’}, default ‘first’
Where there are duplicate values:

```
        * first : prioritize the first occurrence(s)

        * last : prioritize the last occurrence(s)

        * all : do not drop any duplicates, even it means selecting
    more than n items.
```

Returns: DataFrame The first n rows ordered by the given columns in descending order.

See also

`DataFrame.sort_values` Sort DataFrame by the values.

`DataFrame.head` Return the first n rows without re-ordering.

In [105…
```
df = pd.DataFrame({'population': [59000000, 65000000, 434000,
                                  434000, 434000, 337000, 337000,
                                  11300, 11300],
                   'GDP': [1937894, 25835600, 12011, 4520, 12128,
                           17036, 182, 38, 311],
                   'aplha-2': ['IT', 'FR', 'MT', 'MV','BN', 'IS',
                               'NR', 'TV', 'AI']},
                      index=['Italy', 'France', 'Malta', 'Maldives',
                             'Brunei', 'Iceland', 'Nauru', 'Tuvala', 'Anguilla'])
df
```

In [106…
```
# Use `nsmallest to select the three rows having the smallest values in column "pop
df.nsmallest(3, 'population')
```

In [107…
```
# When using `keep=last`, ties are resolved in reverse order.
df.nsmallest(3, 'population', keep='last')
```

In [108…
```
# When using `keep=all`, all duplicate items are maintained:
df.nsmallest(3, 'population', keep='all')
```

In [109…
```
# Order by the smallest values in column 'population' and the 'GDP', we can specify
df.nsmallest(3, ['population', 'GDP'])
```

In [110…
```
df.nlargest(3, 'population')
```

In [111…
```
# Using `keep='last'`, ties are resolved in reverse order:
df.nlargest(3, 'population', keep='last')
```

In [112…
```
# When using `keep='all'`, all duplicates items are maintained
df.nlargest(3, 'population', keep='all')
```

In [113…
```
# Specify multiple columns
df.nlargest(3, ['population', 'GDP'])
```

argmax() argmin()**

Series.argmax(axis=None, skipna=True, *args, **kwargs)

```
Return int position of the largest value in the Series.

If the maximum is achieved in multiple locations, the first row
position is returned.
```

Series.argmin(axis=None, skipna=True, *args, **kwargs)

```
Return int position of the smallest value in the Series.

If the minimum is achieved in multiple locations, the first row
position is returned.
```

Parameters:

axis:{None}

Unused. Parameter needed for compatibility with DataFrame.

skipna:bool, default True

Exclude NA/null values when showing the result.

***args, **kwargs**

Additional arguments and keywords for compatibility with NumPy.

Returns: int Row position of the maximum value.

In [114…
```
s = pd.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
               'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
s
```

In [115…
```
s.argmax()
```

In [116…
```
s.argmin()
```

idxmax() idxmin()

DataFrame.idxmax(axis=0, skipna=True, numeric_only=False)

Return index of first occurence of maximum over requested axis

NA/Null values are excluded

DataFrame.idxmin(axis=0, skipna=True, numeric_only=False)

Return index of first occurrence of minimum over requested axis.

Parameters: axis: {0 or 'index', 1 or 'columns'}, default 0

The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

skipna: bool, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA.

numeric_only: bool, default False
Include only float, int or boolean data.

Returns: Series
Indexes of maxima along the specified axis.

Indexes of minima along the specified axis.

This method is the DataFrame version of `ndarray.argmin`.

In [117…
```
df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
                   'co2_emissions': [37.2, 19.66, 1712]},
                  index=['Pork', 'Wheat Products', 'Beef'])

df
```

In [118…
```
df.idxmin()
```

In [119…
```
df.idxmin(axis='columns')
```

In [120…
```
df.idxmax()
```

In [121…
```
df.idxmax(axis='columns')
```

alue_counts(subset=None, normalize=False, sort=True, ascending=False, dropna=True) Return a Series containing the frequency of each distinct row in the Dataframe.

Parameters: subset: *label or list of labels, optional*
Columns to use when counting unique combinations.

normalize: *bool, default False*
Return proportions rather than frequencies.

sort: *bool, default True*
Sort by frequencies when True. Sort by DataFrame column values when False.

ascending: *bool, default False* Sort in ascending order.

dropna: *bool, default True* Don't include counts of rows that contain NA values.

Returns:
Series

In [122…
```
df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
                   'num_wings': [2, 0, 0, 0]},
                  index=['falcon', 'dog', 'cat', 'ant'])
```

```
df
```

In [123…
```
df.value_counts()
```

In [124…
```
df.value_counts(sort=False)
```

In [125…
```
df.value_counts(ascending=True)
```

In [126…
```
df.value_counts(normalize=True)
```

In [127…
```
df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
                   'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})

df
```

In [128…
```
df.value_counts()
```

In [129…
```
df.value_counts(dropna=False)
```

In [130…
```
df.value_counts('first_name')
```

clip

DataFrame.clip(lower=None, upper=None, *, axis=None, inplace=False, kwargs)**

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis

Parameters:
lower: ***float or array-like, default None***
Minimum threshold value. All values below this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.

upper: ***float or array-like, default None***
Maximum threshold value. All values above this threshold will be set to it. A missing threshold (e.g NA) will not clip the value.

axis: ***{0 or 'index', 1 or 'columns', None}, default None***
Align object with lower and upper along the given axis. For Series this parameter is unused and defaults to None.

inplace: ***bool, default False***
Whether to perform the operation in place on the data.

args, **kwargs:** ***Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns:
Series or DataFrame or None Same type as calling object with the values outside the clip

boundaries replaced or None if inplace=True.

In [131…
```
data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-1, -7, 6, 8, -5]}
df= pd.DataFrame(data)
df
```

In [132…
```
# Clips per column using lower and upper thresholds:
df.clip(-4, 6)
```

In [133…
```
# Clips using specific lower and upper thresholds per column element:
t = pd.Series([2, -4, -1, 6, 3])
t
```

In [134…
```
df.clip(t, t+4, axis=0)
```

In [135…
```
# Clips using specific lower threshold per column element, with missing values:
t = pd.Series([2, -4, np.nan, 6, 3])
t
```

In [136…
```
df.clip(t, axis=0)
```

at_time()

DataFrame.at_time(time, asof=False, axis=None) Select values at particular time of day (e.g. 9:30AM)

Parameters: time: ***datetime.time or str***
The values to select.

axis: ***{0 or 'index', 1 or 'columns'}, default 0***
For Series this parameter is unused and defaults to 0.

Returns:
Series or DataFrame

Raises:
TypeError If the index is not a `DatetimeIndex`

See also

`between_time`
Select values between particular times of the day.

`first`
Select initial periods of time series based on a date offset.

`last`
Select final periods of time series based on a date offset.

`DatetimeIndex.indexer_at_time`
Get just the index locations for values at particular time of the day.

In [137…
```
i = pd.date_range('2018-04-09', periods=4, freq='12H')
ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
ts
```

In [138…
```
ts.at_time('12:00')
```

bdate_range

pandas.bdate_range *(start=None, end=None, periods=None, freq='B', tz=None, normalize=True, name=None, weekmask=None, holidays=None, inclusive='both', *kwargs)*

Return a fixed frequency DatetimeIndex with business days as the default

Parameters:

start: ***str or datetime-like, default None***
Left bound for generating dates.

end: ***str or datetime-like, default None***
Right bound for generating dates.

periods: ***int, default None***
Number of periods to generate.

freq: ***str, Timedelta, datetime.timedelta, or DateOffset, default 'B'***
Frequency strings can have multiples, e.g. '5H'. The default is business daily ('B').

tz: ***str or None***
Time zone name for returning localized DatetimeIndex, for example Asia/Beijing.

normalize: ***bool, default False***
Normalize start/end dates to midnight before generating date range.

name: ***str, default None***
Name of the resulting DatetimeIndex.

weekmask: ***str or None, default None***
Weekmask of valid business days, passed to numpy.busdaycalendar, only used when custom frequency strings are passed. The default value None is equivalent to 'Mon Tue Wed Thu Fri'.

holidays: ***list-like or None, default None***
Dates to exclude from the set of valid business days, passed to numpy.busdaycalendar, only used when custom frequency strings are passed.

Inclusive: ***{"both", "neither", "left", "right"}, default "both"***
Include boundaries; Whether to set each bound as closed or open.

****kwargs**
For compatibility. Has no effect on the result.

Returns: DatetimeIndex

In [139…
```
pd.bdate_range(start='1/1/2018', end='1/08/2018')
```

autocorr

Series.autocorr(lag=1)

Compute the lag-N autocorrelation.

This method computes the Pearson correlation between the Series and its shifted self.

Parameters: lag: ***int, default 1*** Number of lags to apply before performing autocorrelation.

Returns: float The Pearson correlation between self and self.shift(lag).

In [140…
```
s = pd.Series([0.25, 0.5, 0.2, -0.05])
```

In [141…
```
s.autocorr()
```

In [142…
```
s.autocorr(lag=2)
```

In [143…
```
s = pd.Series([1, 0, 0, 0])
```

In [144…
```
s.autocorr()
```

hasnans

pandas.Series.hasnans

***property** Series.hasnans Return True if there are any NaNs.

Enables various performance speedups.

Returns: bool

In [145…
```
s = pd.Series([1, 2, 3, None])
s
```

In [146…
```
s.hasnans
```

at and iat

property **DataFrame.at**

Access a single value for a row/column label pair

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises:
KeyError

- If getting a value and 'label' does not exist in a DataFrame or Series.

  ValueError

- If row/column label pair is not a tuple or if any label from the pair is not a scalar for DataFrame.

- If label is list-like (excluding NamedTuple) for Series.

***property* DataFrame.iat**

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

Raises:
IndexError
When integer position is out of bounds

In [147…
```
df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                  index=[4, 5, 6], columns=['A', 'B', 'C'])
df
```

In [148…
```
# Get value at specified row/column pair
df.at[4, 'B']
```

In [149…
```
# Set value at specified row/column pair
df.at[4, 'B'] = 10
```

In [150…
```
df.at[4, 'B']
```

In [151…
```
# Get value within a Series
df.loc[5].at['B']
```

In [152…
```
df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                  columns=['A', 'B', 'C'])
df
```

In [153…
```
# Get value at specified row/column pair
df.iat[1, 2]
```

In [154…
```
# Set value at a specified row/column pair
df.iat[1, 2] = 10
```

In [155…
```
df.iat[1, 2]
```

In [156…
```
# Get value within a series
df.loc[0].iat[1]
```

argsort()

Series.argsort(axis=0, kind='quicksort', order=None)

Return the integer indices that would sort the Series values

Override ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values

Parameters:
axis: *{0 or 'index'}*
Unused. Parameter needed for compatibility with DataFrame.

kind: *{'mergesort', 'quicksort', 'heapsort', 'stable'}, default 'quicksort'*
Choice of sorting algorithm. See `numpy.sort()` for more information. 'mergesort' and 'stable' are the only stable algorithms.

order: *None* Has no effect but is accepted for compatibility with numpy.

Returns:
Series[np.intp]
Positions of values within the sort order with -1 indicating nan values

In [157…
```
s = pd.Series([3, 2, 1])
```

In [158…
```
s.argsort()
```

cat()

Series.cat()

Accessor object for categorical properties of the Series values.

Parameters:
data: *Series or CategoricalIndex*

In [159…
```
s = pd.Series(list('abbccc')).astype('category')
s
```

In [160…
```
s.cat.categories
```

In [161…
```
s.cat.rename_categories(list('cba'))
```

In [162…
```
s.cat.add_categories(['d', 'e'])
```

In [163…
```
s.cat.remove_categories(['a', 'c'])
```

In [164…
```
s1 = s.cat.add_categories(['d', 'e'])
```

In [165…
```
s1.cat.remove_unused_categories()
```

In [166…
```
s1
```

In [167…
```
s
```

In [168…
```
s.cat.set_categories(list('abcde'))
```

In [169…
```
s.cat.as_ordered()
```

In [170…
```
s.cat.as_unordered()
```

GroupBy.nth

***property* DataFrameGroupBy.nth**

Take the nth row from each group if n is an int, otherwise a subset of rows.

Can be either a call or an `index. dropna` is not available with index notation. Index notation accepts a comma separated list of integers and slices.

If dropna, will take the nth non-null row, dropna is either 'all' or 'any'; this is equivalent to calling `dropna(how=dropna)` before the groupby.

Parameters:
n: ***int, slice or list of ints and slices***
A single nth value for the row or a list of nth values or slices.

dropna: ***{'any', 'all', None}, default None***
Apply the specified dropna operation before counting which row is the nth row. Only supported if n is an int.

Returns:
Series or DataFrame N-th value within each group.

In [171…
```
df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
                   'B': [np.nan, 2, 3, 4, 5]}, columns=['A', 'B'])
df
```

In [172…
```
g = df.groupby('A')
```

In [173…
```
g
```

In [174…
```
g.nth(0)
```

In [175…
```
g.nth(1)
```

In [176…
```
g.nth(-1)
```

In [177…
```
g.nth([0, 1])
```

In [178…
```
g.nth(slice(None, -1))
```

In [179…
```
# Index notation may also be used
g.nth[0,1]
```

In [180…
```
g.nth[:-1]
```

In [181…
```
# Specifying dropna allows ignoring `NaN` values
g.nth(0, dropna='any')
```

In [182…
```
# When the specified `n` is larger than any of the groups, an empty DataFrame is re
g.nth(3, dropna='any')
```

In [ ]:
```

```