Examples

Checks for Alphabetic and Numeric Characters

```
>>> s1 = pd.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s1.str.isalnum()
0     True
1     True
2     True
3    False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = pd.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0    False
1    False
2    False
dtype: bool
```

More Detailed Checks for Numeric Characters

There are several different but overlapping sets of numeric characters that can be checked for.

```
>>> s3 = pd.Series(['23', '³', '', ''])
```

The `s3.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s3.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s.str.isdigit` method is the same as `s3.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s3.str.isdigit()
0     True
```

(continues on next page)

(continued from previous page)

```
1     True
2    False
3    False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s3.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s3.str.isnumeric()
0     True
1     True
2     True
3    False
dtype: bool
```

Checks for Whitespace

```
>>> s4 = pd.Series([' ', '\t\r\n ', ''])
>>> s4.str.isspace()
0     True
1     True
2    False
dtype: bool
```

Checks for Character Case

```
>>> s5 = pd.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

```
>>> s5.str.islower()
0     True
1    False
2    False
3    False
dtype: bool
```

```
>>> s5.str.isupper()
0    False
1    False
2     True
3    False
dtype: bool
```

The `s5.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```
>>> s5.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

pandas.Series.str.get_dummies

`Series.str.get_dummies(self, sep='|')`

Split each string in the Series by sep and return a DataFrame of dummy/indicator variables.

Parameters

sep [str, default "|"] String to split on.

Returns

DataFrame Dummy variables corresponding to values of the Series.

See also:

`get_dummies` Convert categorical variable into dummy/indicator variables.

Examples

```
>>> pd.Series(['a|b', 'a', 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  1  0  0
2  1  0  1
```

```
>>> pd.Series(['a|b', np.nan, 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  0  0  0
2  1  0  1
```

Categorical accessor

Categorical-dtype specific methods and attributes are available under the `Series.cat` accessor.

`Series.cat.categories`	The categories of this categorical.
`Series.cat.ordered`	Whether the categories have an ordered relationship.
`Series.cat.codes`	Return Series of codes as well as the index.

pandas.Series.cat.categories

`Series.cat.categories`

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to *categories* is a inplace operation!

Raises

ValueError If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

`rename_categories`

reorder_categories
add_categories
remove_categories
remove_unused_categories
set_categories

pandas.Series.cat.ordered

`Series.cat.ordered`
Whether the categories have an ordered relationship.

pandas.Series.cat.codes

`Series.cat.codes`
Return Series of codes as well as the index.

Series.cat.rename_categories(self, *args, ...)	Rename categories.
Series.cat.reorder_categories(self, *args, ...)	Reorder categories as specified in new_categories.
Series.cat.add_categories(self, *args, **kwargs)	Add new categories.
Series.cat.remove_categories(self, *args, ...)	Remove the specified categories.
Series.cat.remove_unused_categories(self, ...)	Remove categories which are not used.
Series.cat.set_categories(self, *args, **kwargs)	Set the categories to the specified new_categories.
Series.cat.as_ordered(self, *args, **kwargs)	Set the Categorical to be ordered.
Series.cat.as_unordered(self, *args, **kwargs)	Set the Categorical to be unordered.

pandas.Series.cat.rename_categories

`Series.cat.rename_categories`(*self*, **args*, ***kwargs*)
Rename categories.

Parameters

new_categories [list-like, dict-like or callable] New categories which will replace old categories.

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.

New in version 0.21.0..

- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

New in version 0.23.0..

inplace [bool, default False] Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

Returns

cat [Categorical or None] With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

Raises

ValueError If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

See also:

reorder_categories
add_categories
remove_categories
remove_unused_categories
set_categories

Examples

```
>>> c = pd.Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

pandas.Series.cat.reorder_categories

`Series.cat.`**`reorder_categories`**(*self*, **args*, ***kwargs*)

Reorder categories as specified in new_categories.

new_categories need to include all old categories and no new category items.

Parameters

new_categories [Index-like] The categories in new order.

ordered [bool, optional] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

inplace [bool, default False] Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

Returns

cat [Categorical with reordered categories or None if inplace.]

Raises

ValueError If the new categories do not contain all old category items or any new ones

See also:

rename_categories
add_categories
remove_categories
remove_unused_categories
set_categories

pandas.Series.cat.add_categories

`Series.cat.add_categories(self, *args, **kwargs)`

Add new categories.

new_categories will be included at the last/highest place in the categories and will be unused directly after this call.

Parameters

new_categories [category or list-like of category] The new categories to be included.

inplace [bool, default False] Whether or not to add the categories inplace or return a copy of this categorical with added categories.

Returns

cat [Categorical with new categories added or None if inplace.]

Raises

ValueError If the new categories include old categories or do not validate as categories

See also:

rename_categories
reorder_categories
remove_categories
remove_unused_categories
set_categories

pandas.Series.cat.remove_categories

`Series.cat.remove_categories(self, *args, **kwargs)`

Remove the specified categories.

removals must be included in the old categories. Values which were in the removed categories will be set to NaN

Parameters

removals [category or list of categories] The categories which should be removed.

inplace [bool, default False] Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

Returns

cat [Categorical with removed categories or None if inplace.]

Raises

ValueError If the removals are not contained in the categories

See also:

rename_categories
reorder_categories
add_categories
remove_unused_categories
set_categories

pandas.Series.cat.remove_unused_categories

`Series.cat.`**`remove_unused_categories`**(*self*, **args*, ***kwargs*)

Remove categories which are not used.

Parameters

inplace [bool, default False] Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

Returns

cat [Categorical with unused categories dropped or None if inplace.]

See also:

rename_categories
reorder_categories
add_categories
remove_categories
set_categories

pandas.Series.cat.set_categories

`Series.cat.`**`set_categories`**(*self*, **args*, ***kwargs*)

Set the categories to the specified new_categories.

new_categories can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simple be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this methods does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes, which does not considers a S1 string equal to a single char python string.

Parameters

new_categories [Index-like] The categories in new order.

ordered [bool, default False] Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

rename [bool, default False] Whether or not the new_categories should be considered as a rename of the old categories or as reordered categories.

inplace [bool, default False] Whether or not to reorder the categories in-place or return a copy of this categorical with reordered categories.

Returns

Categorical with reordered categories or None if inplace.

Raises

ValueError If new_categories does not validate as categories

See also:

rename_categories
reorder_categories
add_categories
remove_categories
remove_unused_categories

pandas.Series.cat.as_ordered

Series.cat.**as_ordered**(*self*, **args*, ***kwargs*)

Set the Categorical to be ordered.

Parameters

inplace [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to True.

Returns

Categorical Ordered Categorical.

pandas.Series.cat.as_unordered

Series.cat.**as_unordered**(*self*, **args*, ***kwargs*)

Set the Categorical to be unordered.

Parameters

inplace [bool, default False] Whether or not to set the ordered attribute in-place or return a copy of this categorical with ordered set to False.

Returns

Categorical Unordered Categorical.

Sparse accessor

Sparse-dtype specific methods and attributes are provided under the `Series.sparse` accessor.

Series.sparse.npoints	The number of non- `fill_value` points.
Series.sparse.density	The percent of non- `fill_value` points, as decimal.
Series.sparse.fill_value	Elements in *data* that are *fill_value* are not stored.
Series.sparse.sp_values	An ndarray containing the non- `fill_value` values.

pandas.Series.sparse.npoints

`Series.sparse.`**`npoints`**

The number of non- `fill_value` points.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.npoints
3
```

pandas.Series.sparse.density

`Series.sparse.`**`density`**

The percent of non- `fill_value` points, as decimal.

Examples

```
>>> s = SparseArray([0, 0, 1, 1, 1], fill_value=0)
>>> s.density
0.6
```

pandas.Series.sparse.fill_value

`Series.sparse.`**`fill_value`**

Elements in *data* that are *fill_value* are not stored.

For memory savings, this should be the most common value in the array.

pandas.Series.sparse.sp_values

`Series.sparse.`**`sp_values`**

An ndarray containing the non- `fill_value` values.

Examples

```
>>> s = SparseArray([0, 0, 1, 0, 2], fill_value=0)
>>> s.sp_values
array([1, 2])
```

`Series.sparse.from_coo`(A[, dense_index])	Create a Series with sparse values from a scipy.sparse.coo_matrix.
`Series.sparse.to_coo`(self[, row_levels, . . .])	Create a scipy.sparse.coo_matrix from a Series with MultiIndex.

pandas.Series.sparse.from_coo

classmethod Series.sparse.**from_coo**(*A*, *dense_index=False*)

Create a Series with sparse values from a scipy.sparse.coo_matrix.

Parameters

A [scipy.sparse.coo_matrix]

dense_index [bool, default False] If False (default), the SparseSeries index consists of only the coords of the non-null entries of the original coo_matrix. If True, the SparseSeries index consists of the full sorted (row, col) coordinates of the coo_matrix.

Returns

s [Series] A Series with sparse values.

Examples

```
>>> from scipy import sparse
>>> A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
                          shape=(3, 4))
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> ss = pd.Series.sparse.from_coo(A)
>>> ss
0  2    1
   3    2
1  0    3
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

pandas.Series.sparse.to_coo

Series.sparse.**to_coo**(*self*, *row_levels=0*, *column_levels=1*, *sort_labels=False*)

Create a scipy.sparse.coo_matrix from a Series with MultiIndex.

Use row_levels and column_levels to determine the row and column coordinates respectively. row_levels and column_levels are the names (labels) or numbers of the levels. {row_levels, column_levels} must be a partition of the MultiIndex level names (or numbers).

Parameters

row_levels [tuple/list]

column_levels [tuple/list]

sort_labels [bool, default False] Sort the row and column labels before forming the sparse matrix.

Returns

y [scipy.sparse.coo_matrix]

rows [list (row labels)]

columns [list (column labels)]

Examples

```
>>> s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])
>>> s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
                                         (1, 2, 'a', 1),
                                         (1, 1, 'b', 0),
                                         (1, 1, 'b', 1),
                                         (2, 1, 'b', 0),
                                         (2, 1, 'b', 1)],
                                        names=['A', 'B', 'C', 'D'])
>>> ss = s.astype("Sparse")
>>> A, rows, columns = ss.sparse.to_coo(row_levels=['A', 'B'],
...                                     column_levels=['C', 'D'],
...                                     sort_labels=True)
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  3.],
[ 3.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.]])
>>> rows
[(1, 1), (1, 2), (2, 1)]
>>> columns
[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

Metadata

Series.attrs is a dictionary for storing global metadata for this Series.

Warning: `Series.attrs` is considered experimental and may change without warning.

Series.attrs	Dictionary of global attributes on this object.

3.3.14 Plotting

`Series.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `Series.plot.<kind>`.

Series.plot([kind, ax, figsize, ….])	Series plotting accessor and method

Series.plot.area(self[, x, y])	Draw a stacked area plot.
Series.plot.bar(self[, x, y])	Vertical bar plot.
Series.plot.barh(self[, x, y])	Make a horizontal bar plot.

continues on next page

Table 54 – continued from previous page

Series.plot.box(self[, by])	Make a box plot of the DataFrame columns.
Series.plot.density(self[, bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
Series.plot.hist(self[, by, bins])	Draw one histogram of the DataFrame's columns.
Series.plot.kde(self[, bw_method, ind])	Generate Kernel Density Estimate plot using Gaussian kernels.
Series.plot.line(self[, x, y])	Plot Series or DataFrame as lines.
Series.plot.pie(self, **kwargs)	Generate a pie plot.

pandas.Series.plot.area

`Series.plot.`**`area`**`(`*self*, *x=None*, *y=None*, ***kwargs*`)`

Draw a stacked area plot.

An area plot displays quantitative data visually. This function wraps the matplotlib area function.

Parameters

x [label or position, optional] Coordinates for the X axis. By default uses the index.

y [label or position, optional] Column to plot. By default uses all columns.

stacked [bool, default True] Area plots are stacked by default. Set to False to create a unstacked plot.

****kwargs** Additional keyword arguments are documented in *DataFrame.plot()*.

Returns

matplotlib.axes.Axes or numpy.ndarray Area plot, or array of area plots if subplots is True.

See also:

DataFrame.plot Make plots of DataFrame using matplotlib / pylab.

Examples

Draw an area plot based on basic business metrics:

```
>>> df = pd.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                        freq='M'))
>>> ax = df.plot.area()
```

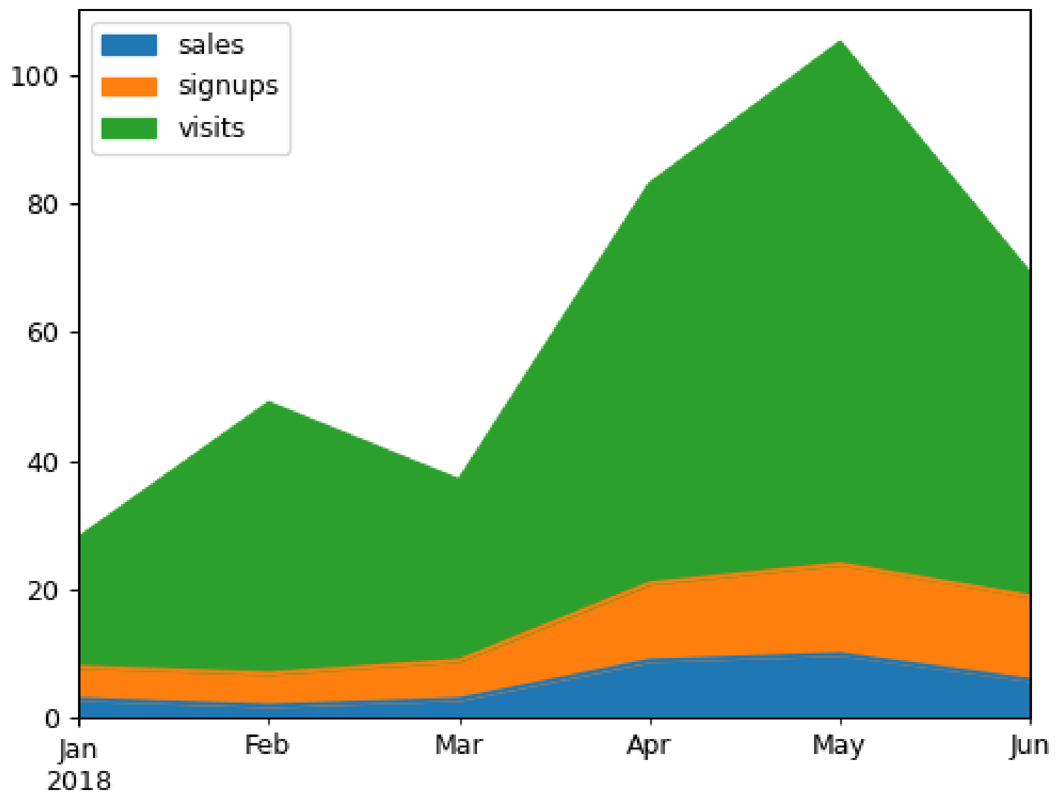
Area plots are stacked by default. To produce an unstacked plot, pass `stacked=False`:

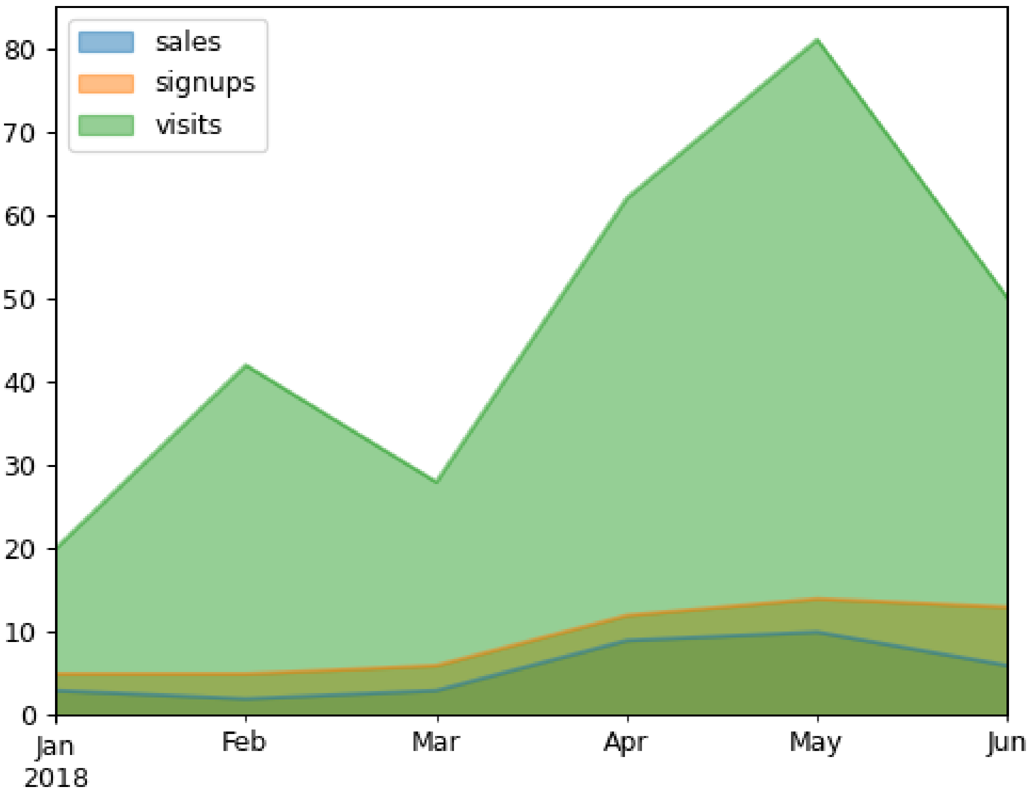
```
>>> ax = df.plot.area(stacked=False)
```

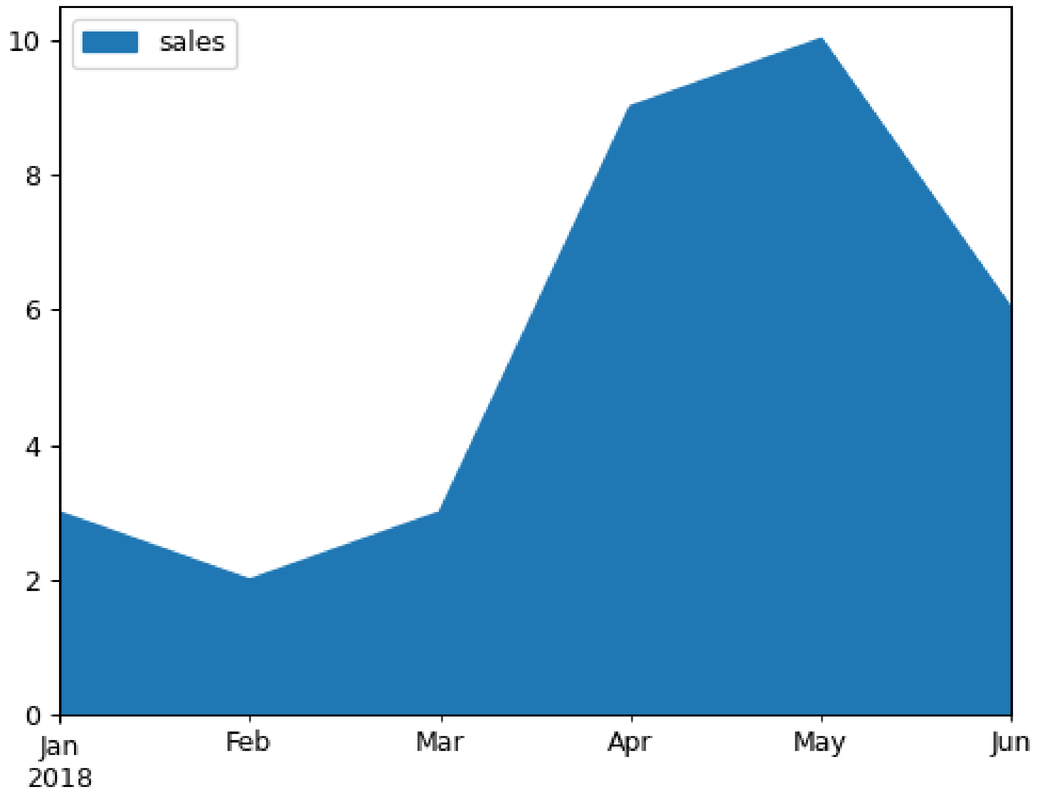
Draw an area plot for a single column:

```
>>> ax = df.plot.area(y='sales')
```

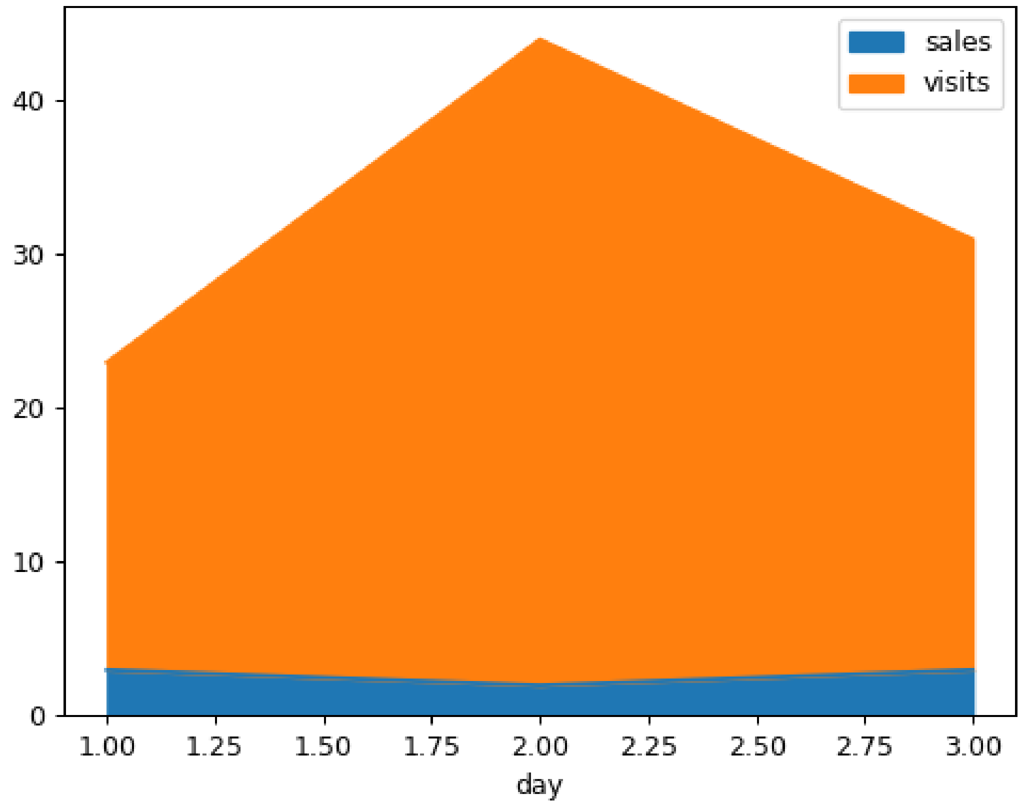
Draw with a different *x*:


sales
signups
visits
0
20
40
60
80
100
Jan
2018
Feb
Mar
Apr
May
Jun


sales
signups
visits
80
70
60
50
40
30
20
10
0
Jan
2018
Feb
Mar
Apr
May
Jun


sales
10
8
6
4
2
0
Jan
2018
Feb
Mar
Apr
May
Jun

```
>>> df = pd.DataFrame({
...     'sales': [3, 2, 3],
...     'visits': [20, 42, 28],
...     'day': [1, 2, 3],
... })
>>> ax = df.plot.area(x='day')
```



pandas.Series.plot.bar

`Series.plot.bar` (*self*, *x=None*, *y=None*, ***kwargs*)

Vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

Parameters

x [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.

y [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.

****kwargs** Additional keyword arguments are documented in `DataFrame.plot()`.

Returns

matplotlib.axes.Axes or np.ndarray of them An ndarray is returned with one `matplotlib.axes.Axes` per column when `subplots=True`.

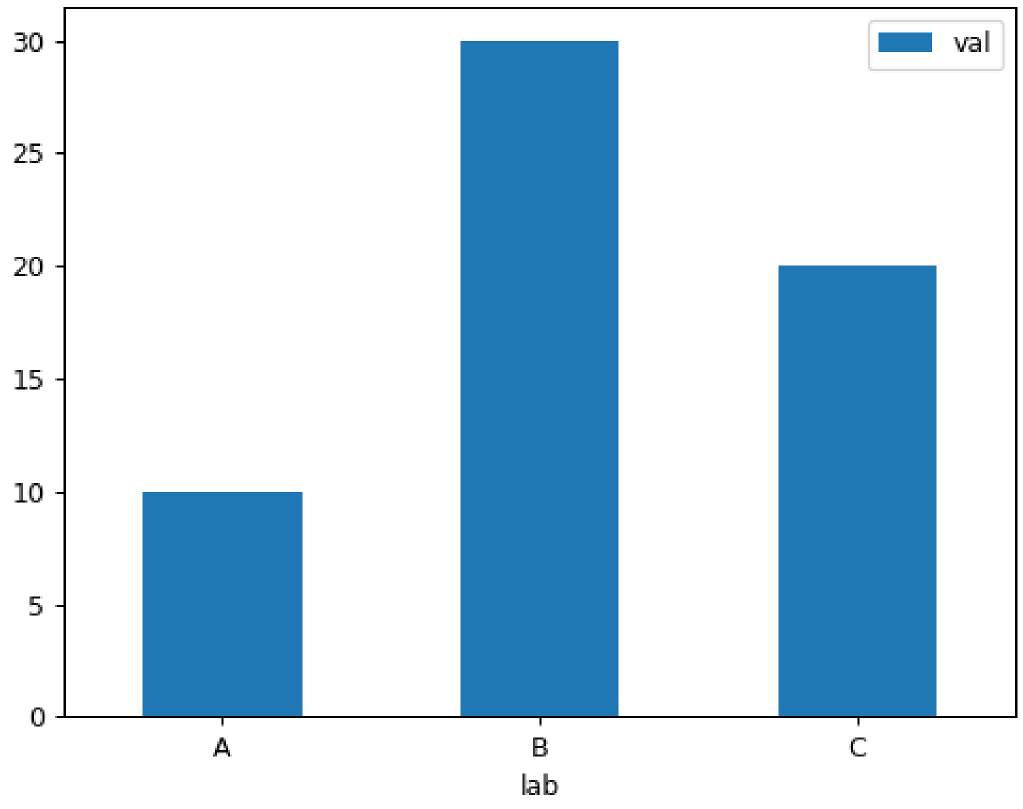
See also:

`DataFrame.plot.barh` Horizontal bar plot.
`DataFrame.plot` Make plots of a DataFrame.
`matplotlib.pyplot.bar` Make a bar plot with matplotlib.

Examples

Basic plot.

```
>>> df = pd.DataFrame({'lab':['A', 'B', 'C'], 'val':[10, 30, 20]})
>>> ax = df.plot.bar(x='lab', y='val', rot=0)
```



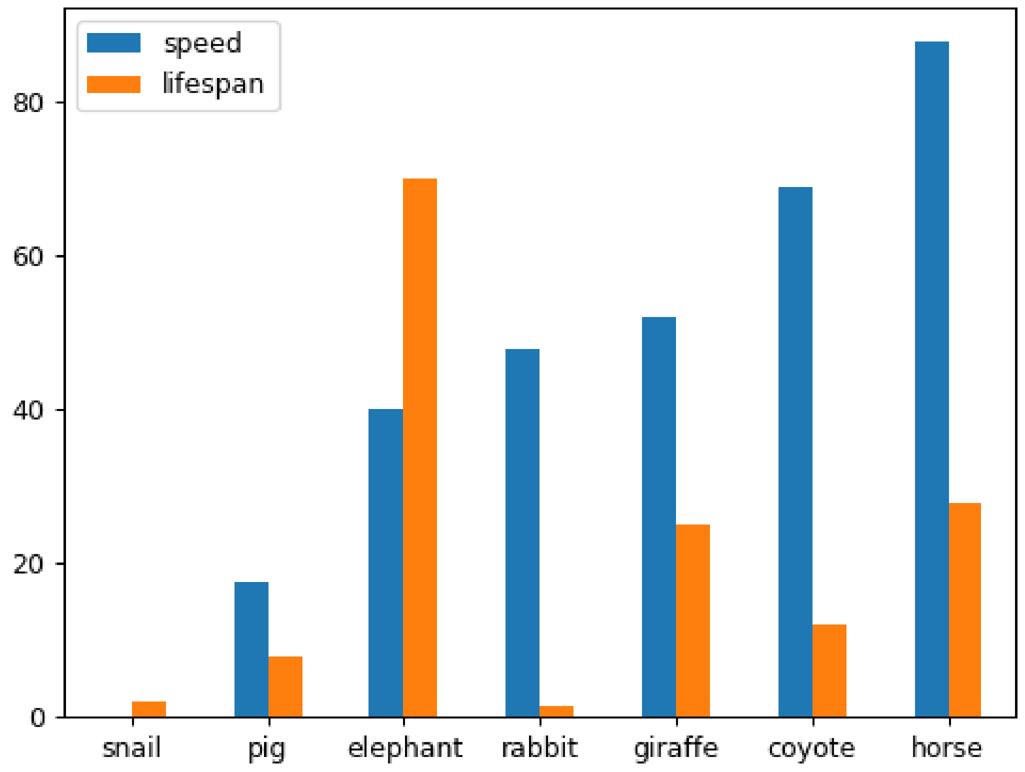
Plot a whole dataframe to a bar plot. Each column is assigned a distinct color, and each row is nested in a group along the horizontal axis.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
```

(continues on next page)

(continued from previous page)

```
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.bar(rot=0)
```



Instead of nesting, the figure can be split by column with `subplots=True`. In this case, a `numpy.ndarray` of `matplotlib.axes.Axes` are returned.

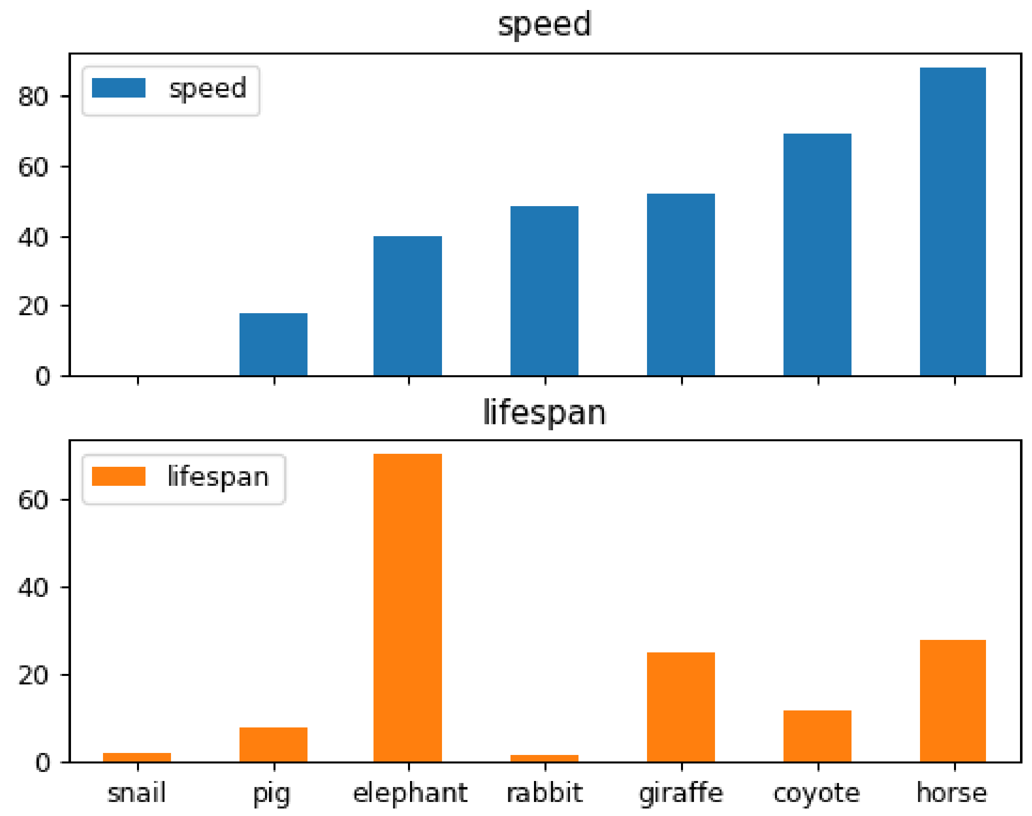
```
>>> axes = df.plot.bar(rot=0, subplots=True)
>>> axes[1].legend(loc=2)
```

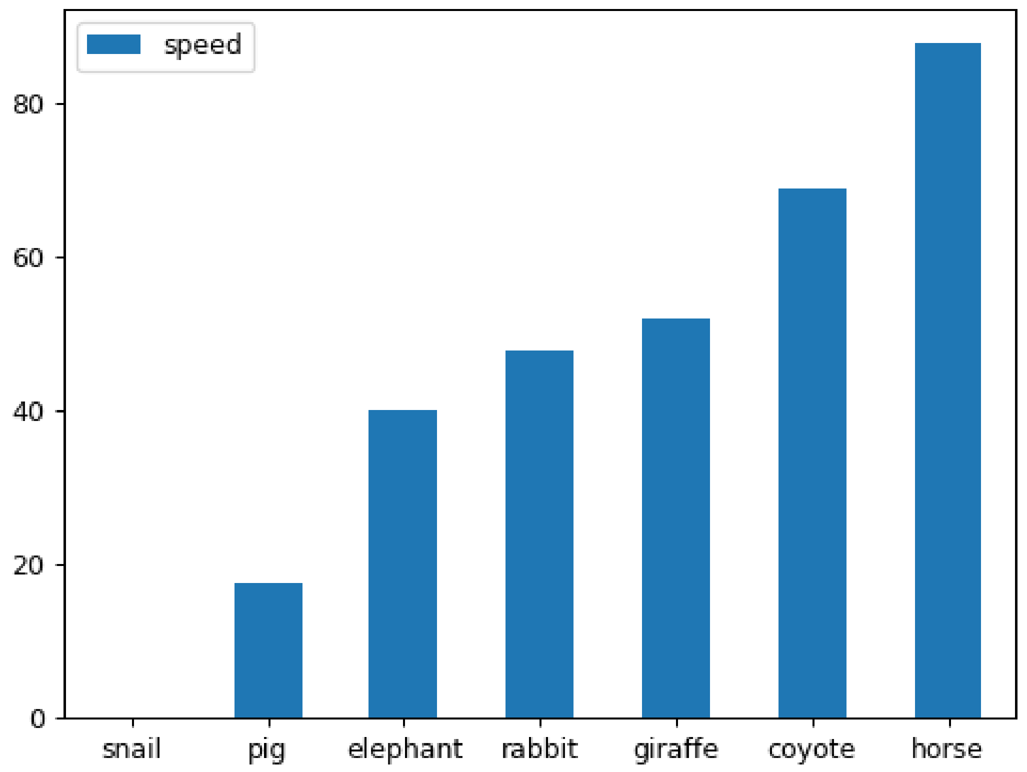
Plot a single column.

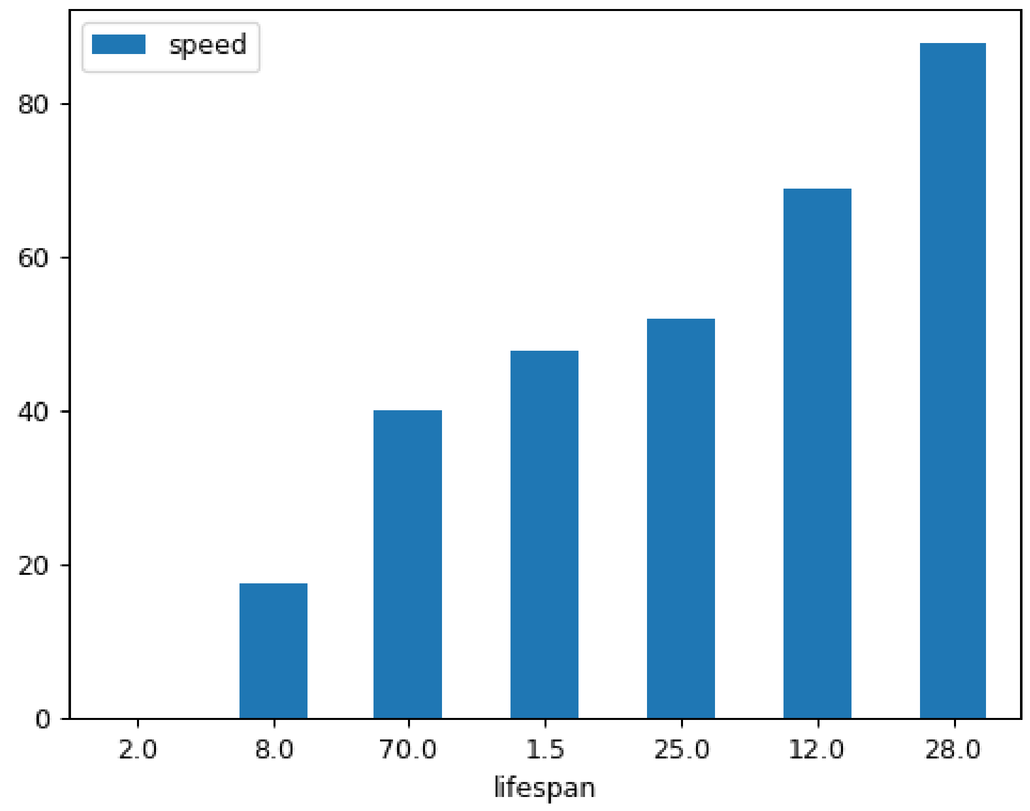
```
>>> ax = df.plot.bar(y='speed', rot=0)
```

Plot only selected categories for the DataFrame.

```
>>> ax = df.plot.bar(x='lifespan', rot=0)
```


speed
speed
0
20
40
60
80
lifespan
lifespan
0
20
40
60
snail
pig
elephant
rabbit
giraffe
coyote
horse


speed
0
20
40
60
80
snail
pig
elephant
rabbit
giraffe
coyote
horse


speed
0
20
40
60
80
2.0
8.0
70.0
1.5
25.0
12.0
28.0
lifespan

pandas.Series.plot.barh

`Series.plot.barh`(*self*, *x=None*, *y=None*, ***kwargs*)

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

Parameters

x [label or position, default DataFrame.index] Column to be used for categories.

y [label or position, default All numeric columns in dataframe] Columns to be plotted from the DataFrame.

****kwargs** Keyword arguments to pass on to `DataFrame.plot()`.

Returns

`matplotlib.axes.Axes` or numpy.ndarray of them

See also:

`DataFrame.plot.bar` Vertical bar plot.
`DataFrame.plot` Make plots of DataFrame using matplotlib.
`matplotlib.axes.Axes.bar` Plot a vertical bar plot using matplotlib.

Examples

Basic example

```
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> ax = df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh()
```

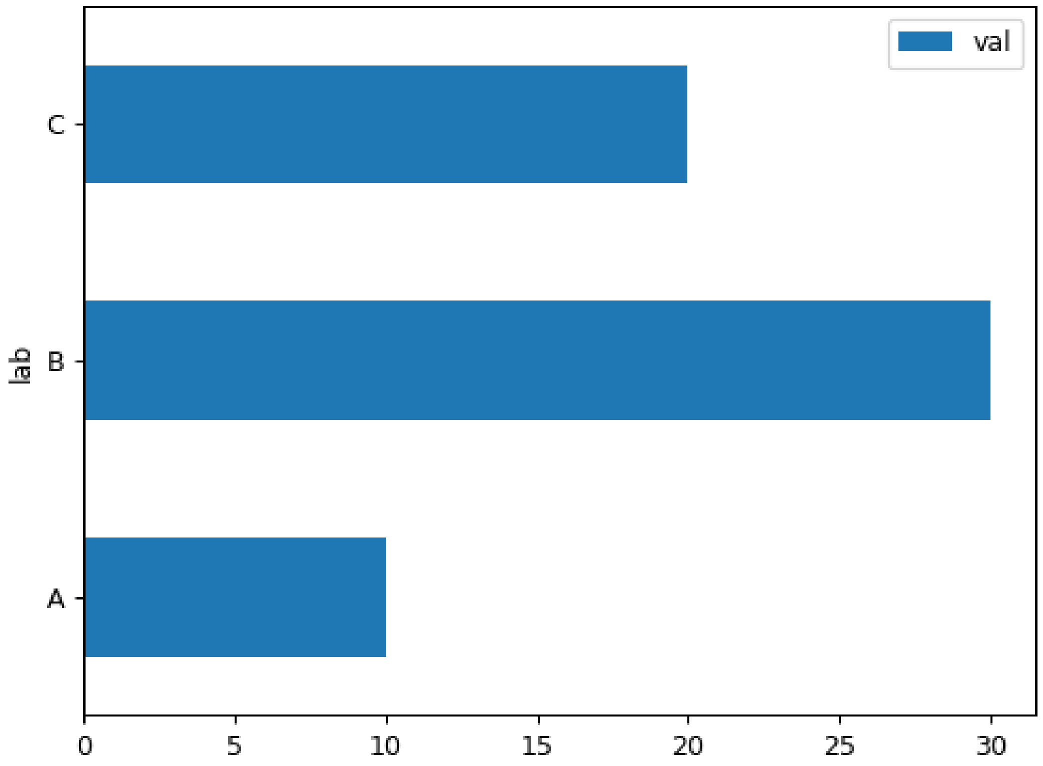
Plot a column of the DataFrame to a horizontal bar plot

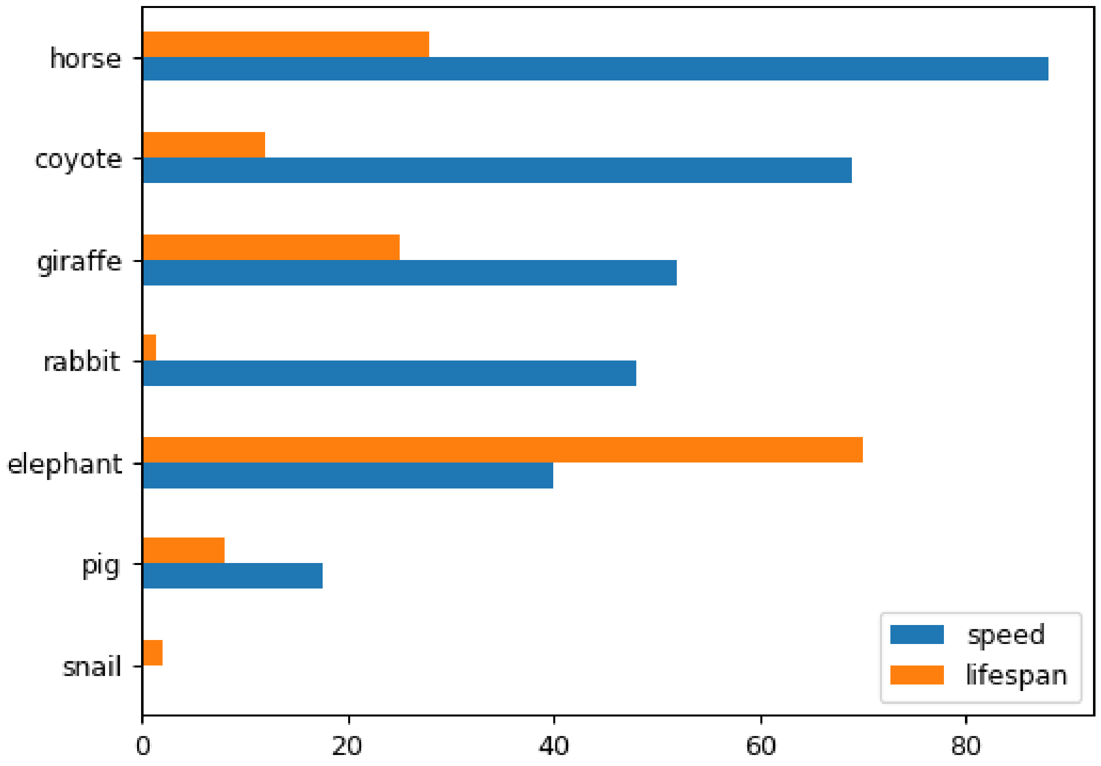
```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                    'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(y='speed')
```

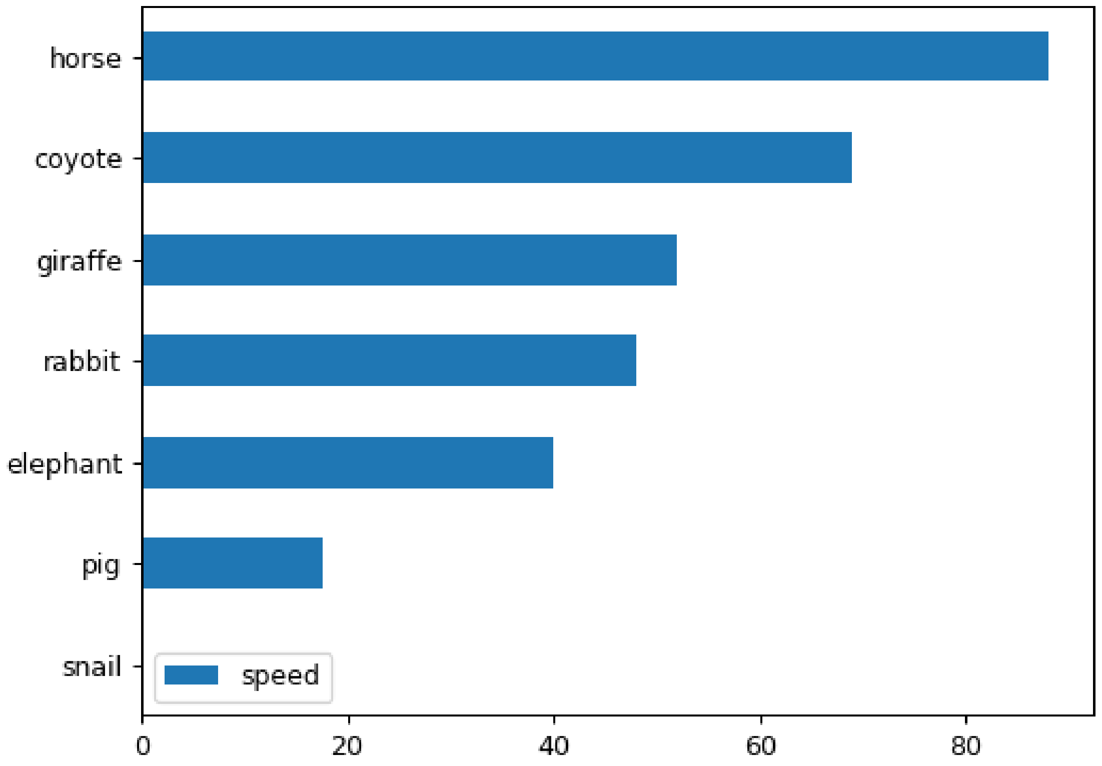
Plot DataFrame versus the desired column

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
```

(continues on next page)


val
C
B
A
lab
0
5
10
15
20
25
30


horse
coyote
giraffe
rabbit
elephant
pig
snail
speed
lifespan
0
20
40
60
80


horse
coyote
giraffe
rabbit
elephant
pig
snail
speed
0
20
40
60
80