(continued from previous page)

```
Categories (2, object): [a, b]

In [158]: str_cat.str.contains("a")
Out[158]:
0     True
1     True
2    False
3    False
dtype: bool

In [159]: date_s = pd.Series(pd.date_range('1/1/2015', periods=5))

In [160]: date_cat = date_s.astype('category')

In [161]: date_cat
Out[161]:
0   2015-01-01
1   2015-01-02
2   2015-01-03
3   2015-01-04
4   2015-01-05
dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-04, 2015-
↪01-05]

In [162]: date_cat.dt.day
Out[162]:
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Note: The returned `Series` (or `DataFrame`) is of the same type as if you used the `.str.<method>` / `.dt.<method>` on a `Series` of that type (and not of type `category`!).

That means, that the returned values from methods and properties on the accessors of a `Series` and the returned values from methods and properties on the accessors of this `Series` transformed to one of type *category* will be equal:

```
In [163]: ret_s = str_s.str.contains("a")

In [164]: ret_cat = str_cat.str.contains("a")

In [165]: ret_s.dtype == ret_cat.dtype
Out[165]: True

In [166]: ret_s == ret_cat
Out[166]:
0    True
1    True
2    True
3    True
dtype: bool
```

Note: The work is done on the `categories` and then a new `Series` is constructed. This has some performance implication if you have a `Series` of type string, where lots of elements are repeated (i.e. the number of unique elements in the `Series` is a lot smaller than the length of the `Series`). In this case it can be faster to convert the original `Series` to one of type `category` and use `.str.<method>` or `.dt.<property>` on that.

Setting

Setting values in a categorical column (or `Series`) works as long as the value is included in the *categories*:

```
In [167]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [168]: cats = pd.Categorical(["a", "a", "a", "a", "a", "a", "a"],
   .....:                       categories=["a", "b"])
   .....:

In [169]: values = [1, 1, 1, 1, 1, 1, 1]

In [170]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [171]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

In [172]: df
Out[172]:
  cats  values
h    a       1
i    a       1
j    b       2
k    b       2
l    a       1
m    a       1
n    a       1

In [173]: try:
   .....:     df.iloc[2:4, :] = [["c", 3], ["c", 3]]
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
ValueError: Cannot setitem on a Categorical with a new category, set the categories␣
↪first
```

Setting values by assigning categorical data will also check that the *categories* match:

```
In [174]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"], categories=["a", "b"])

In [175]: df
Out[175]:
  cats  values
h    a       1
i    a       1
j    a       2
k    a       2
l    a       1
m    a       1
n    a       1
```

(continues on next page)

(continued from previous page)

```
In [176]: try:
   .....:     df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"],
   .....:                                              categories=["a", "b", "c"])
   .....: except ValueError as e:
   .....:     print("ValueError:", str(e))
   .....:
ValueError: Cannot set a Categorical with another, without identical categories
```

Assigning a `Categorical` to parts of a column of other types will use the values:

```
In [177]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a", "a"]})

In [178]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [179]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [180]: df
Out[180]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a

In [181]: df.dtypes
Out[181]:
a    object
b    object
dtype: object
```

Merging / Concatenation

By default, combining `Series` or `DataFrames` which contain the same categories results in `category` dtype, otherwise results will depend on the dtype of the underlying categories. Merges that result in non-categorical dtypes will likely have higher memory usage. Use `.astype` or `union_categoricals` to ensure `category` results.

```
In [182]: from pandas.api.types import union_categoricals

# same categories
In [183]: s1 = pd.Series(['a', 'b'], dtype='category')

In [184]: s2 = pd.Series(['a', 'b', 'a'], dtype='category')

In [185]: pd.concat([s1, s2])
Out[185]:
0    a
1    b
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]
```

(continues on next page)

(continued from previous page)

```
# different categories
In [186]: s3 = pd.Series(['b', 'c'], dtype='category')

In [187]: pd.concat([s1, s3])
Out[187]:
0    a
1    b
0    b
1    c
dtype: object

# Output dtype is inferred based on categories values
In [188]: int_cats = pd.Series([1, 2], dtype="category")

In [189]: float_cats = pd.Series([3.0, 4.0], dtype="category")

In [190]: pd.concat([int_cats, float_cats])
Out[190]:
0    1.0
1    2.0
0    3.0
1    4.0
dtype: float64

In [191]: pd.concat([s1, s3]).astype('category')
Out[191]:
0    a
1    b
0    b
1    c
dtype: category
Categories (3, object): [a, b, c]

In [192]: union_categoricals([s1.array, s3.array])
Out[192]:
[a, b, b, c]
Categories (3, object): [a, b, c]
```

The following table summarizes the results of merging `Categoricals`:

arg1	arg2	identical	result
category	category	True	category
category (object)	category (object)	False	object (dtype is inferred)
category (int)	category (float)	False	float (dtype is inferred)

See also the section on *merge dtypes* for notes about preserving merge dtypes and performance.

Unioning

If you want to combine categoricals that do not necessarily have the same categories, the *union_categoricals()* function will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
In [193]: from pandas.api.types import union_categoricals

In [194]: a = pd.Categorical(["b", "c"])

In [195]: b = pd.Categorical(["a", "b"])

In [196]: union_categoricals([a, b])
Out[196]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the data. If you want the categories to be lexsorted, use `sort_categories=True` argument.

```
In [197]: union_categoricals([a, b], sort_categories=True)
Out[197]:
[b, c, a, b]
Categories (3, object): [a, b, c]
```

`union_categoricals` also works with the "easy" case of combining two categoricals of the same categories and order information (e.g. what you could also `append` for).

```
In [198]: a = pd.Categorical(["a", "b"], ordered=True)

In [199]: b = pd.Categorical(["a", "b", "a"], ordered=True)

In [200]: union_categoricals([a, b])
Out[200]:
[a, b, a, b, a]
Categories (2, object): [a < b]
```

The below raises `TypeError` because the categories are ordered and not identical.

```
In [1]: a = pd.Categorical(["a", "b"], ordered=True)
In [2]: b = pd.Categorical(["a", "b", "c"], ordered=True)
In [3]: union_categoricals([a, b])
Out[3]:
TypeError: to union ordered Categoricals, all categories must be the same
```

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```
In [201]: a = pd.Categorical(["a", "b", "c"], ordered=True)

In [202]: b = pd.Categorical(["c", "b", "a"], ordered=True)

In [203]: union_categoricals([a, b], ignore_order=True)
Out[203]:
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

union_categoricals() also works with a `CategoricalIndex`, or `Series` containing categorical data, but note that the resulting array will always be a plain `Categorical`:

```
In [204]: a = pd.Series(["b", "c"], dtype='category')

In [205]: b = pd.Series(["a", "b"], dtype='category')

In [206]: union_categoricals([a, b])
Out[206]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

Note: `union_categoricals` may recode the integer codes for categories when combining categoricals. This is likely what you want, but if you are relying on the exact numbering of the categories, be aware.

```
In [207]: c1 = pd.Categorical(["b", "c"])

In [208]: c2 = pd.Categorical(["a", "b"])

In [209]: c1
Out[209]:
[b, c]
Categories (2, object): [b, c]

# "b" is coded to 0
In [210]: c1.codes
Out[210]: array([0, 1], dtype=int8)

In [211]: c2
Out[211]:
[a, b]
Categories (2, object): [a, b]

# "b" is coded to 1
In [212]: c2.codes
Out[212]: array([0, 1], dtype=int8)

In [213]: c = union_categoricals([c1, c2])

In [214]: c
Out[214]:
[b, c, a, b]
Categories (3, object): [b, c, a]

# "b" is coded to 0 throughout, same as c1, different from c2
In [215]: c.codes
Out[215]: array([0, 1, 2, 0], dtype=int8)
```

2.8.9 Getting data in/out

You can write data that contains `category` dtypes to a `HDFStore`. See *here* for an example and caveats.

It is also possible to write data to and reading data from *Stata* format files. See *here* for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to *category* and assign the right categories and categories ordering.

```
In [216]: import io

In [217]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))

# rename the categories
In [218]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [219]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [220]: df = pd.DataFrame({"cats": s, "vals": [1, 2, 3, 4, 5, 6]})

In [221]: csv = io.StringIO()

In [222]: df.to_csv(csv)

In [223]: df2 = pd.read_csv(io.StringIO(csv.getvalue()))

In [224]: df2.dtypes
Out[224]:
Unnamed: 0     int64
cats          object
vals           int64
dtype: object

In [225]: df2["cats"]
Out[225]:
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: object

# Redo the category
In [226]: df2["cats"] = df2["cats"].astype("category")

In [227]: df2["cats"].cat.set_categories(["very bad", "bad", "medium",
   .....:                                  "good", "very good"],
   .....:                                 inplace=True)
   .....:

In [228]: df2.dtypes
Out[228]:
Unnamed: 0       int64
cats          category
vals             int64
```

(continues on next page)

(continued from previous page)

```
dtype: object

In [229]: df2["cats"]
Out[229]:
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

The same holds for writing to a SQL database with `to_sql`.

2.8.10 Missing data

pandas primarily uses the value *np.nan* to represent missing data. It is by default not included in computations. See the *Missing Data section*.

Missing values should **not** be included in the Categorical's `categories`, only in the `values`. Instead, it is understood that NaN is different, and is always a possibility. When working with the Categorical's `codes`, missing values will always have a code of `-1`.

```
In [230]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")

# only two categories
In [231]: s
Out[231]:
0      a
1      b
2    NaN
3      a
dtype: category
Categories (2, object): [a, b]

In [232]: s.cat.codes
Out[232]:
0    0
1    1
2   -1
3    0
dtype: int8
```

Methods for working with missing data, e.g. `isna()`, `fillna()`, `dropna()`, all work normally:

```
In [233]: s = pd.Series(["a", "b", np.nan], dtype="category")

In [234]: s
Out[234]:
0      a
1      b
2    NaN
dtype: category
Categories (2, object): [a, b]
```

(continues on next page)

(continued from previous page)

```
In [235]: pd.isna(s)
Out[235]:
0    False
1    False
2     True
dtype: bool

In [236]: s.fillna("a")
Out[236]:
0    a
1    b
2    a
dtype: category
Categories (2, object): [a, b]
```

2.8.11 Differences to R's *factor*

The following differences to R's factor functions can be observed:

- R's *levels* are named *categories*.
- R's *levels* are always of type string, while *categories* in pandas can be of any dtype.
- It's not possible to specify labels at creation time. Use `s.cat.rename_categories(new_labels)` afterwards.
- In contrast to R's *factor* function, using categorical data as the sole input to create a new categorical series will *not* remove unused categories but create a new categorical series which is equal to the passed in one!
- R allows for missing values to be included in its *levels* (pandas' *categories*). Pandas does not allow *NaN* categories, but missing values can still be in the *values*.

2.8.12 Gotchas

Memory usage

The memory usage of a `Categorical` is proportional to the number of categories plus the length of the data. In contrast, an `object` dtype is a constant times the length of the data.

```
In [237]: s = pd.Series(['foo', 'bar'] * 1000)

# object dtype
In [238]: s.nbytes
Out[238]: 16000

# category dtype
In [239]: s.astype('category').nbytes
Out[239]: 2016
```

Note: If the number of categories approaches the length of the data, the `Categorical` will use nearly the same or more memory than an equivalent `object` dtype representation.

```
In [240]: s = pd.Series(['foo%04d' % i for i in range(2000)])

# object dtype
In [241]: s.nbytes
Out[241]: 16000

# category dtype
In [242]: s.astype('category').nbytes
Out[242]: 20000
```

Categorical is not a *numpy* array

Currently, categorical data and the underlying `Categorical` is implemented as a Python object and not as a low-level NumPy array dtype. This leads to some problems.

NumPy itself doesn't know about the new *dtype*:

```
In [243]: try:
   .....:     np.dtype("category")
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type "category" not understood

In [244]: dtype = pd.Categorical(["a"]).dtype

In [245]: try:
   .....:     np.dtype(dtype)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type not understood
```

Dtype comparisons work:

```
In [246]: dtype == np.str_
Out[246]: False

In [247]: np.str_ == dtype
Out[247]: False
```

To check if a Series contains Categorical data, use `hasattr(s, 'cat')`:

```
In [248]: hasattr(pd.Series(['a'], dtype='category'), 'cat')
Out[248]: True

In [249]: hasattr(pd.Series(['a']), 'cat')
Out[249]: False
```

Using NumPy functions on a `Series` of type `category` should not work as *Categoricals* are not numeric data (even in the case that `.categories` is numeric).

```
In [250]: s = pd.Series(pd.Categorical([1, 2, 3, 4]))

In [251]: try:
```

(continues on next page)

(continued from previous page)

```
   .....:     np.sum(s)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categorical cannot perform the operation sum
```

Note: If such a function works, please file a bug at https://github.com/pandas-dev/pandas!

dtype in apply

Pandas currently does not preserve the dtype in apply functions: If you apply along rows you get a *Series* of `object` *dtype* (same as getting a row -> getting one element will return a basic type) and applying along columns will also convert to object. `NaN` values are unaffected. You can use `fillna` to handle missing values before applying a function.

```
In [252]: df = pd.DataFrame({"a": [1, 2, 3, 4],
   .....:                    "b": ["a", "b", "c", "d"],
   .....:                    "cats": pd.Categorical([1, 2, 3, 2])})
   .....:

In [253]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[253]:
0    <class 'int'>
1    <class 'int'>
2    <class 'int'>
3    <class 'int'>
dtype: object

In [254]: df.apply(lambda col: col.dtype, axis=0)
Out[254]:
a          int64
b         object
cats    category
dtype: object
```

Categorical index

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements. See the *advanced indexing docs* for a more detailed explanation.

Setting the index will create a `CategoricalIndex`:

```
In [255]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])

In [256]: strings = ["a", "b", "c", "d"]

In [257]: values = [4, 2, 3, 1]

In [258]: df = pd.DataFrame({"strings": strings, "values": values}, index=cats)

In [259]: df.index
```

(continues on next page)

(continued from previous page)

```
Out[259]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False,␣
↪dtype='category')

# This now sorts by the categories order
In [260]: df.sort_index()
Out[260]:
  strings  values
4       d       1
2       b       2
3       c       3
1       a       4
```

Side effects

Constructing a `Series` from a `Categorical` will not copy the input `Categorical`. This means that changes to the `Series` will in most cases change the original `Categorical`:

```
In [261]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [262]: s = pd.Series(cat, name="cat")

In [263]: cat
Out[263]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [264]: s.iloc[0:2] = 10

In [265]: cat
Out[265]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [266]: df = pd.DataFrame(s)

In [267]: df["cat"].cat.categories = [1, 2, 3, 4, 5]

In [268]: cat
Out[268]:
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]
```

Use `copy=True` to prevent such a behaviour or simply don't reuse `Categoricals`:

```
In [269]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [270]: s = pd.Series(cat, name="cat", copy=True)

In [271]: cat
Out[271]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [272]: s.iloc[0:2] = 10
```

(continues on next page)

(continued from previous page)

```
In [273]: cat
Out[273]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
```

Note: This also happens in some cases when you supply a NumPy array instead of a `Categorical`: using an int array (e.g. `np.array([1,2,3,4])`) will exhibit the same behavior, while using a string array (e.g. `np.array(["a","b","c","a"])`) will not.

2.9 Nullable integer data type

New in version 0.24.0.

Note: IntegerArray is currently experimental. Its API or implementation may change without warning.

Changed in version 1.0.0: Now uses `pandas.NA` as the missing value rather than `numpy.nan`.

In *Working with missing data*, we saw that pandas primarily uses `NaN` to represent missing data. Because `NaN` is a float, this forces an array of integers with any missing values to become floating point. In some cases, this may not matter much. But if your integer column is, say, an identifier, casting to float can be problematic. Some integers cannot even be represented as floating point numbers.

2.9.1 Construction

Pandas can represent integer data with possibly missing values using *`arrays.IntegerArray`*. This is an *extension types* implemented within pandas.

```
In [1]: arr = pd.array([1, 2, None], dtype=pd.Int64Dtype())

In [2]: arr
Out[2]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

Or the string alias `"Int64"` (note the capital `"I"`, to differentiate from NumPy's `'int64'` dtype:

```
In [3]: pd.array([1, 2, np.nan], dtype="Int64")
Out[3]:
<IntegerArray>
[1, 2, <NA>]
Length: 3, dtype: Int64
```

All NA-like values are replaced with `pandas.NA`.

```
In [4]: pd.array([1, 2, np.nan, None, pd.NA], dtype="Int64")
Out[4]:
<IntegerArray>
[1, 2, <NA>, <NA>, <NA>]
Length: 5, dtype: Int64
```

This array can be stored in a `DataFrame` or `Series` like any NumPy array.

```
In [5]: pd.Series(arr)
Out[5]:
0       1
1       2
2    <NA>
dtype: Int64
```

You can also pass the list-like object to the `Series` constructor with the dtype.

Warning: Currently `pandas.array()` and `pandas.Series()` use different rules for dtype inference. `pandas.array()` will infer a nullable- integer dtype

```
In [6]: pd.array([1, None])
Out[6]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64

In [7]: pd.array([1, 2])
Out[7]:
<IntegerArray>
[1, 2]
Length: 2, dtype: Int64
```

For backwards-compatibility, `Series` infers these as either integer or float dtype

```
In [8]: pd.Series([1, None])
Out[8]:
0    1.0
1    NaN
dtype: float64

In [9]: pd.Series([1, 2])
Out[9]:
0    1
1    2
dtype: int64
```

We recommend explicitly providing the dtype to avoid confusion.

```
In [10]: pd.array([1, None], dtype="Int64")
Out[10]:
<IntegerArray>
[1, <NA>]
Length: 2, dtype: Int64

In [11]: pd.Series([1, None], dtype="Int64")
Out[11]:
0       1
1    <NA>
dtype: Int64
```

In the future, we may provide an option for `Series` to infer a nullable-integer dtype.

2.9.2 Operations

Operations involving an integer array will behave similar to NumPy arrays. Missing values will be propagated, and the data will be coerced to another dtype if needed.

```
In [12]: s = pd.Series([1, 2, None], dtype="Int64")

# arithmetic
In [13]: s + 1
Out[13]:
0       2
1       3
2    <NA>
dtype: Int64

# comparison
In [14]: s == 1
Out[14]:
0     True
1    False
2     <NA>
dtype: boolean

# indexing
In [15]: s.iloc[1:3]
Out[15]:
1       2
2    <NA>
dtype: Int64

# operate with other dtypes
In [16]: s + s.iloc[1:3].astype('Int8')
Out[16]:
0    <NA>
1       4
2    <NA>
dtype: Int64

# coerce when needed
In [17]: s + 0.01
Out[17]:
0    1.01
1    2.01
2     NaN
dtype: float64
```

These dtypes can operate as part of of `DataFrame`.

```
In [18]: df = pd.DataFrame({'A': s, 'B': [1, 1, 3], 'C': list('aab')})

In [19]: df
Out[19]:
      A  B  C
0     1  1  a
1     2  1  a
2  <NA>  3  b

In [20]: df.dtypes
```

(continues on next page)

(continued from previous page)

```
Out[20]:
A     Int64
B     int64
C    object
dtype: object
```

These dtypes can be merged & reshaped & casted.

```
In [21]: pd.concat([df[['A']], df[['B', 'C']]], axis=1).dtypes
Out[21]:
A     Int64
B     int64
C    object
dtype: object

In [22]: df['A'].astype(float)
Out[22]:
0    1.0
1    2.0
2    NaN
Name: A, dtype: float64
```

Reduction and groupby operations such as 'sum' work as well.

```
In [23]: df.sum()
Out[23]:
A      3
B      5
C    aab
dtype: object

In [24]: df.groupby('B').A.sum()
Out[24]:
B
1    3
3    0
Name: A, dtype: Int64
```

2.9.3 Scalar NA Value

arrays.IntegerArray uses `pandas.NA` as its scalar missing value. Slicing a single element that's missing will return `pandas.NA`

```
In [25]: a = pd.array([1, None], dtype="Int64")

In [26]: a[1]
Out[26]: <NA>
```

2.10 Nullable Boolean Data Type

New in version 1.0.0.

2.10.1 Indexing with NA values

pandas allows indexing with `NA` values in a boolean array, which are treated as `False`.

Changed in version 1.0.2.

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

In [2]: mask = pd.array([True, False, pd.NA], dtype="boolean")

In [3]: s[mask]
Out[3]:
0    1
dtype: int64
```

If you would prefer to keep the `NA` values you can manually fill them with `fillna(True)`.

```
In [4]: s[mask.fillna(True)]
Out[4]:
0    1
2    3
dtype: int64
```

2.10.2 Kleene Logical Operations

`arrays.BooleanArray` implements Kleene Logic (sometimes called three-value logic) for logical operations like `&` (and), `|` (or) and `^` (exclusive-or).

This table demonstrates the results for every combination. These operations are symmetrical, so flipping the left- and right-hand side makes no difference in the result.

Expression	Result
`True & True`	`True`
`True & False`	`False`
`True & NA`	`NA`
`False & False`	`False`
`False & NA`	`False`
`NA & NA`	`NA`
`True | True`	`True`
`True | False`	`True`
`True | NA`	`True`
`False | False`	`False`
`False | NA`	`NA`
`NA | NA`	`NA`
`True ^ True`	`False`
`True ^ False`	`True`
`True ^ NA`	`NA`
`False ^ False`	`False`
`False ^ NA`	`NA`
`NA ^ NA`	`NA`

When an `NA` is present in an operation, the output value is `NA` only if the result cannot be determined solely based on the other input. For example, `True | NA` is `True`, because both `True | True` and `True | False` are `True`. In that case, we don't actually need to consider the value of the `NA`.

On the other hand, `True & NA` is `NA`. The result depends on whether the `NA` really is `True` or `False`, since `True & True` is `True`, but `True & False` is `False`, so we can't determine the output.

This differs from how `np.nan` behaves in logical operations. Pandas treated `np.nan` is *always false in the output*.

In `or`

```
In [5]: pd.Series([True, False, np.nan], dtype="object") | True
Out[5]:
0     True
1     True
2    False
dtype: bool

In [6]: pd.Series([True, False, np.nan], dtype="boolean") | True
Out[6]:
0    True
1    True
2    True
dtype: boolean
```

In `and`

```
In [7]: pd.Series([True, False, np.nan], dtype="object") & True
Out[7]:
0     True
1    False
2    False
dtype: bool

In [8]: pd.Series([True, False, np.nan], dtype="boolean") & True
```

(continues on next page)

(continued from previous page)

```
Out[8]:
0     True
1    False
2     <NA>
dtype: boolean
```

{{ header }}

2.11 Visualization

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt

In [2]: plt.close('all')
```

We provide the basics in pandas to easily create decent looking plots. See the ecosystem section for visualization libraries that go beyond the basics documented here.

Note: All calls to `np.random` are seeded with 123456.

2.11.1 Basic plotting: `plot`

We will demonstrate the basics, see the *cookbook* for some advanced strategies.

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`:

```
In [3]: ts = pd.Series(np.random.randn(1000),
   ...:                index=pd.date_range('1/1/2000', periods=1000))
   ...:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-00eeb137fb11> in <module>
----> 1 ts = pd.Series(np.random.randn(1000),
      2                index=pd.date_range('1/1/2000', periods=1000))

NameError: name 'pd' is not defined

In [4]: ts = ts.cumsum()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-a7771f529bde> in <module>
----> 1 ts = ts.cumsum()

NameError: name 'ts' is not defined

In [5]: ts.plot()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-8a34b37f0ce9> in <module>
----> 1 ts.plot()
```

(continues on next page)

(continued from previous page)

```
NameError: name 'ts' is not defined
```

If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above.

On DataFrame, `plot()` is a convenience to plot all of the columns with labels:

```
In [6]: df = pd.DataFrame(np.random.randn(1000, 4),
   ...:                   index=ts.index, columns=list('ABCD'))
   ...:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-ae243d2a43b5> in <module>
----> 1 df = pd.DataFrame(np.random.randn(1000, 4),
      2                   index=ts.index, columns=list('ABCD'))

NameError: name 'pd' is not defined

In [7]: df = df.cumsum()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-08208d45ae16> in <module>
----> 1 df = df.cumsum()
```

(continues on next page)

(continued from previous page)

```
NameError: name 'df' is not defined

In [8]: plt.figure();

In [9]: df.plot();
```

You can plot one column versus another using the *x* and *y* keywords in `plot()`:

```
In [10]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-2bf7f8097da6> in <module>
----> 1 df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()

NameError: name 'pd' is not defined

In [11]: df3['A'] = pd.Series(list(range(len(df))))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-d039f08f00c4> in <module>
----> 1 df3['A'] = pd.Series(list(range(len(df))))

NameError: name 'pd' is not defined
```

(continues on next page)

(continued from previous page)

```
In [12]: df3.plot(x='A', y='B')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-6b33533f2e7d> in <module>
----> 1 df3.plot(x='A', y='B')

NameError: name 'df3' is not defined
```

Note: For more formatting and styling options, see *formatting* below.

2.11.2 Other plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to `plot()`, and include:

- *'bar'* or *'barh'* for bar plots
- *'hist'* for histogram
- *'box'* for boxplot
- *'kde'* or *'density'* for density plots

- *'area'* for area plots
- *'scatter'* for scatter plots
- *'hexbin'* for hexagonal bin plots
- *'pie'* for pie plots

For example, a bar plot can be created the following way:

```
In [13]: plt.figure();

In [14]: df.iloc[5].plot(kind='bar');
```

You can also create these other plots using the methods `DataFrame.plot.<kind>` instead of providing the `kind` keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

```
In [15]: df = pd.DataFrame()

In [16]: df.plot.<TAB>  # noqa: E225, E999
df.plot.area     df.plot.barh     df.plot.density  df.plot.hist     df.plot.line    ␣
↪df.plot.scatter
df.plot.bar      df.plot.box      df.plot.hexbin   df.plot.kde      df.plot.pie
```

In addition to these `kind` s, there are the *DataFrame.hist()*, and *DataFrame.boxplot()* methods, which use a separate interface.

Finally, there are several *plotting functions* in `pandas.plotting` that take a `Series` or `DataFrame` as an argument. These include:

- *Scatter Matrix*
- *Andrews Curves*
- *Parallel Coordinates*
- *Lag Plot*
- *Autocorrelation Plot*
- *Bootstrap Plot*
- *RadViz*

Plots may also be adorned with *errorbars* or *tables*.

Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

```
In [17]: plt.figure();

In [18]: df.iloc[5].plot.bar()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-d1a2cddc601a> in <module>
----> 1 df.iloc[5].plot.bar()

NameError: name 'df' is not defined

In [19]: plt.axhline(0, color='k');
```

Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

```
In [20]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-6133adb252fc> in <module>
----> 1 df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

NameError: name 'pd' is not defined

In [21]: df2.plot.bar();
```