(continued from previous page)

```
In [136]: crit.dtype
Out[136]: dtype('O')
```

Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

```
In [137]: reindexed = s.reindex(list(range(8))).fillna(0)

In [138]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-138-0dac417a4890> in <module>
----> 1 reindexed[crit]

/pandas-release/pandas/pandas/core/series.py in __getitem__(self, key)
    905             key = list(key)
    906
--> 907         if com.is_bool_indexer(key):
    908             key = check_bool_indexer(self.index, key)
    909

/pandas-release/pandas/pandas/core/common.py in is_bool_indexer(key)
    134                 na_msg = "Cannot mask with non-boolean array containing NA /␣
↪NaN values"
    135                 if isna(key).any():
--> 136                     raise ValueError(na_msg)
    137                 return False
    138             return True

ValueError: Cannot mask with non-boolean array containing NA / NaN values
```

However, these can be filled in using `fillna()` and it will work fine:

```
In [139]: reindexed[crit.fillna(False)]
Out[139]:
0    0.126504
2    0.696198
4    0.697416
6    0.601516
7    0.003659
dtype: float64

In [140]: reindexed[crit.fillna(True)]
Out[140]:
0    0.126504
1    0.000000
2    0.696198
3    0.000000
4    0.697416
5    0.000000
6    0.601516
7    0.003659
dtype: float64
```

Pandas provides a nullable integer dtype, but you must explicitly request it when creating the series or column. Notice that we use a capital "I" in the `dtype="Int64"`.

```
In [141]: s = pd.Series([0, 1, np.nan, 3, 4], dtype="Int64")

In [142]: s
Out[142]:
0       0
1       1
2    <NA>
3       3
4       4
dtype: Int64
```

See *Nullable integer data type* for more.

2.7.13 Experimental NA scalar to denote missing values

Warning: Experimental: the behaviour of `pd.NA` can still change without warning.

New in version 1.0.0.

Starting from pandas 1.0, an experimental `pd.NA` value (singleton) is available to represent scalar missing values. At this moment, it is used in the nullable *integer*, boolean and *dedicated string* data types as the missing value indicator.

The goal of `pd.NA` is provide a "missing" indicator that can be used consistently across data types (instead of `np.nan`, `None` or `pd.NaT` depending on the data type).

For example, when having missing values in a Series with the nullable integer dtype, it will use `pd.NA`:

```
In [143]: s = pd.Series([1, 2, None], dtype="Int64")

In [144]: s
Out[144]:
0       1
1       2
2    <NA>
dtype: Int64

In [145]: s[2]
Out[145]: <NA>

In [146]: s[2] is pd.NA
Out[146]: True
```

Currently, pandas does not yet use those data types by default (when creating a DataFrame or Series, or when reading in data), so you need to specify the dtype explicitly. An easy way to convert to those dtypes is explained *here*.

Propagation in arithmetic and comparison operations

In general, missing values *propagate* in operations involving `pd.NA`. When one of the operands is unknown, the outcome of the operation is also unknown.

For example, `pd.NA` propagates in arithmetic operations, similarly to `np.nan`:

```
In [147]: pd.NA + 1
Out[147]: <NA>

In [148]: "a" * pd.NA
Out[148]: <NA>
```

There are a few special cases when the result is known, even when one of the operands is `NA`.

```
In [149]: pd.NA ** 0
Out[149]: 1

In [150]: 1 ** pd.NA
Out[150]: 1
```

In equality and comparison operations, `pd.NA` also propagates. This deviates from the behaviour of `np.nan`, where comparisons with `np.nan` always return `False`.

```
In [151]: pd.NA == 1
Out[151]: <NA>

In [152]: pd.NA == pd.NA
Out[152]: <NA>

In [153]: pd.NA < 2.5
Out[153]: <NA>
```

To check if a value is equal to `pd.NA`, the *`isna()`* function can be used:

```
In [154]: pd.isna(pd.NA)
Out[154]: True
```

An exception on this basic propagation rule are *reductions* (such as the mean or the minimum), where pandas defaults to skipping missing values. See *above* for more.

Logical operations

For logical operations, `pd.NA` follows the rules of the three-valued logic (or *Kleene logic*, similarly to R, SQL and Julia). This logic means to only propagate missing values when it is logically required.

For example, for the logical "or" operation (`|`), if one of the operands is `True`, we already know the result will be `True`, regardless of the other value (so regardless the missing value would be `True` or `False`). In this case, `pd.NA` does not propagate:

```
In [155]: True | False
Out[155]: True

In [156]: True | pd.NA
Out[156]: True
```

(continues on next page)

(continued from previous page)

```
In [157]: pd.NA | True
Out[157]: True
```

On the other hand, if one of the operands is `False`, the result depends on the value of the other operand. Therefore, in this case `pd.NA` propagates:

```
In [158]: False | True
Out[158]: True

In [159]: False | False
Out[159]: False

In [160]: False | pd.NA
Out[160]: <NA>
```

The behaviour of the logical "and" operation (`&`) can be derived using similar logic (where now `pd.NA` will not propagate if one of the operands is already `False`):

```
In [161]: False & True
Out[161]: False

In [162]: False & False
Out[162]: False

In [163]: False & pd.NA
Out[163]: False
```

```
In [164]: True & True
Out[164]: True

In [165]: True & False
Out[165]: False

In [166]: True & pd.NA
Out[166]: <NA>
```

`NA` in a boolean context

Since the actual value of an NA is unknown, it is ambiguous to convert NA to a boolean value. The following raises an error:

```
In [167]: bool(pd.NA)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-167-5477a57d5abb> in <module>
----> 1 bool(pd.NA)

/pandas-release/pandas/pandas/_libs/missing.pyx in pandas._libs.missing.NAType.__bool_
↪_()

TypeError: boolean value of NA is ambiguous
```

This also means that `pd.NA` cannot be used in a context where it is evaluated to a boolean, such as `if condition: ...` where `condition` can potentially be `pd.NA`. In such cases, *`isna()`* can be used to check for `pd.NA` or `condition` being `pd.NA` can be avoided, for example by filling missing values beforehand.

A similar situation occurs when using Series or DataFrame objects in `if` statements, see *Using if/truth statements with pandas*.

NumPy ufuncs

`pandas.NA` implements NumPy's `__array_ufunc__` protocol. Most ufuncs work with `NA`, and generally return `NA`:

```
In [168]: np.log(pd.NA)
Out[168]: <NA>

In [169]: np.add(pd.NA, 1)
Out[169]: <NA>
```

Warning: Currently, ufuncs involving an ndarray and `NA` will return an object-dtype filled with NA values.

```
In [170]: a = np.array([1, 2, 3])

In [171]: np.greater(a, pd.NA)
Out[171]: array([<NA>, <NA>, <NA>], dtype=object)
```

The return type here may change to return a different array type in the future.

See *DataFrame interoperability with NumPy functions* for more on ufuncs.

Conversion

If you have a DataFrame or Series using traditional types that have missing data represented using `np.nan`, there are convenience methods `convert_dtypes()` in Series and `convert_dtypes()` in DataFrame that can convert data to use the newer dtypes for integers, strings and booleans listed *here*. This is especially helpful after reading in data sets when letting the readers such as `read_csv()` and `read_excel()` infer default dtypes.

In this example, while the dtypes of all columns are changed, we show the results for the first 10 columns.

```
In [172]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [173]: bb[bb.columns[:10]].dtypes
Out[173]:
player    object
year       int64
stint      int64
team      object
lg        object
g          int64
ab         int64
r          int64
h          int64
X2b        int64
dtype: object
```

```
In [174]: bbn = bb.convert_dtypes()

In [175]: bbn[bbn.columns[:10]].dtypes
Out[175]:
```

(continues on next page)

(continued from previous page)

```
player    string
year       Int64
stint      Int64
team      string
lg        string
g          Int64
ab         Int64
r          Int64
h          Int64
X2b        Int64
dtype: object
```

2.8 Categorical data

This is an introduction to pandas categorical data type, including a short comparison with R's `factor`.

Categoricals are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (*categories*; *levels* in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

In contrast to statistical categorical variables, categorical data might have an order (e.g. 'strongly agree' vs 'agree' or 'first observation' vs. 'second observation'), but numerical operations (additions, divisions, . . .) are not possible.

All values of categorical data are either in *categories* or *np.nan*. Order is defined by the order of *categories*, not lexical order of the values. Internally, the data structure consists of a *categories* array and an integer array of *codes* which point to the real value in the *categories* array.

The categorical data type is useful in the following cases:

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory, see *here*.
- The lexical order of a variable is not the same as the logical order ("one", "two", "three"). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order, see *here*.
- As a signal to other Python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

See also the *API docs on categoricals*.

2.8.1 Object creation

Series creation

Categorical `Series` or columns in a `DataFrame` can be created in several ways:

By specifying `dtype="category"` when constructing a `Series`:

```
In [1]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

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

(continues on next page)

(continued from previous page)

```
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]
```

By converting an existing `Series` or column to a `category` dtype:

```
In [3]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [4]: df["B"] = df["A"].astype('category')

In [5]: df
Out[5]:
   A  B
0  a  a
1  b  b
2  c  c
3  a  a
```

By using special functions, such as *cut()*, which groups data into discrete bins. See the *example on tiling* in the docs.

```
In [6]: df = pd.DataFrame({'value': np.random.randint(0, 100, 20)})

In [7]: labels = ["{0} - {1}".format(i, i + 9) for i in range(0, 100, 10)]

In [8]: df['group'] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)

In [9]: df.head(10)
Out[9]:
   value    group
0     65  60 - 69
1     49  40 - 49
2     56  50 - 59
3     43  40 - 49
4     43  40 - 49
5     91  90 - 99
6     32  30 - 39
7     87  80 - 89
8     36  30 - 39
9      8    0 - 9
```

By passing a *pandas.Categorical* object to a `Series` or assigning it to a `DataFrame`.

```
In [10]: raw_cat = pd.Categorical(["a", "b", "c", "a"], categories=["b", "c", "d"],
   ....:                          ordered=False)
   ....:

In [11]: s = pd.Series(raw_cat)

In [12]: s
Out[12]:
0    NaN
1      b
2      c
3    NaN
```

(continues on next page)

(continued from previous page)

```
dtype: category
Categories (3, object): [b, c, d]

In [13]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [14]: df["B"] = raw_cat

In [15]: df
Out[15]:
   A    B
0  a  NaN
1  b    b
2  c    c
3  a  NaN
```

Categorical data has a specific `category` *dtype*:

```
In [16]: df.dtypes
Out[16]:
A      object
B    category
dtype: object
```

DataFrame creation

Similar to the previous section where a single column was converted to categorical, all columns in a `DataFrame` can be batch converted to categorical either during or after construction.

This can be done during construction by specifying `dtype="category"` in the `DataFrame` constructor:

```
In [17]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')}, dtype="category")

In [18]: df.dtypes
Out[18]:
A    category
B    category
dtype: object
```

Note that the categories present in each column differ; the conversion is done column by column, so only labels present in a given column are categories:

```
In [19]: df['A']
Out[19]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [20]: df['B']
Out[20]:
0    b
1    c
2    c
```

(continues on next page)

(continued from previous page)

```
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

New in version 0.23.0.

Analogously, all columns in an existing `DataFrame` can be batch converted using `DataFrame.astype()`:

```
In [21]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [22]: df_cat = df.astype('category')

In [23]: df_cat.dtypes
Out[23]:
A    category
B    category
dtype: object
```

This conversion is likewise done column by column:

```
In [24]: df_cat['A']
Out[24]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [25]: df_cat['B']
Out[25]:
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

Controlling behavior

In the examples above where we passed `dtype='category'`, we used the default behavior:

1. Categories are inferred from the data.
2. Categories are unordered.

To control those behaviors, instead of passing `'category'`, use an instance of `CategoricalDtype`.

```
In [26]: from pandas.api.types import CategoricalDtype

In [27]: s = pd.Series(["a", "b", "c", "a"])

In [28]: cat_type = CategoricalDtype(categories=["b", "c", "d"],
   ....:                             ordered=True)
   ....:
```

(continues on next page)

(continued from previous page)

```
In [29]: s_cat = s.astype(cat_type)

In [30]: s_cat
Out[30]:
0    NaN
1      b
2      c
3    NaN
dtype: category
Categories (3, object): [b < c < d]
```

Similarly, a `CategoricalDtype` can be used with a `DataFrame` to ensure that categories are consistent among all columns.

```
In [31]: from pandas.api.types import CategoricalDtype

In [32]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [33]: cat_type = CategoricalDtype(categories=list('abcd'),
   ....:                             ordered=True)
   ....:

In [34]: df_cat = df.astype(cat_type)

In [35]: df_cat['A']
Out[35]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (4, object): [a < b < c < d]

In [36]: df_cat['B']
Out[36]:
0    b
1    c
2    c
3    d
Name: B, dtype: category
Categories (4, object): [a < b < c < d]
```

Note: To perform table-wise conversion, where all labels in the entire `DataFrame` are used as categories for each column, the `categories` parameter can be determined programmatically by `categories = pd.unique(df.to_numpy().ravel())`.

If you already have `codes` and `categories`, you can use the *from_codes()* constructor to save the factorize step during normal constructor mode:

```
In [37]: splitter = np.random.choice([0, 1], 5, p=[0.5, 0.5])

In [38]: s = pd.Series(pd.Categorical.from_codes(splitter,
   ....:                                         categories=["train", "test"]))
   ....:
```

Regaining original data

To get back to the original `Series` or NumPy array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:

```
In [39]: s = pd.Series(["a", "b", "c", "a"])

In [40]: s
Out[40]:
0    a
1    b
2    c
3    a
dtype: object

In [41]: s2 = s.astype('category')

In [42]: s2
Out[42]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [43]: s2.astype(str)
Out[43]:
0    a
1    b
2    c
3    a
dtype: object

In [44]: np.asarray(s2)
Out[44]: array(['a', 'b', 'c', 'a'], dtype=object)
```

Note: In contrast to R's *factor* function, categorical data is not converting input values to strings; categories will end up the same data type as the original values.

Note: In contrast to R's *factor* function, there is currently no way to assign/change labels at creation time. Use *categories* to change the categories after creation time.

2.8.2 CategoricalDtype

Changed in version 0.21.0.

A categorical's type is fully described by

1. `categories`: a sequence of unique values and no missing values
2. `ordered`: a boolean

This information can be stored in a `CategoricalDtype`. The `categories` argument is optional, which implies that the actual categories should be inferred from whatever is present in the data when the *pandas.Categorical* is created. The categories are assumed to be unordered by default.

```
In [45]: from pandas.api.types import CategoricalDtype

In [46]: CategoricalDtype(['a', 'b', 'c'])
Out[46]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)

In [47]: CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[47]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [48]: CategoricalDtype()
Out[48]: CategoricalDtype(categories=None, ordered=False)
```

A `CategoricalDtype` can be used in any place pandas expects a *dtype*. For example *pandas.read_csv()*, *pandas.DataFrame.astype()*, or in the `Series` constructor.

Note: As a convenience, you can use the string `'category'` in place of a `CategoricalDtype` when you want the default behavior of the categories being unordered, and equal to the set values present in the array. In other words, `dtype='category'` is equivalent to `dtype=CategoricalDtype()`.

Equality semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and order. When comparing two unordered categoricals, the order of the `categories` is not considered.

```
In [49]: c1 = CategoricalDtype(['a', 'b', 'c'], ordered=False)

# Equal, since order is not considered when ordered=False
In [50]: c1 == CategoricalDtype(['b', 'c', 'a'], ordered=False)
Out[50]: True

# Unequal, since the second CategoricalDtype is ordered
In [51]: c1 == CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[51]: False
```

All instances of `CategoricalDtype` compare equal to the string `'category'`.

```
In [52]: c1 == 'category'
Out[52]: True
```

Warning: Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances `CategoricalDtype` compare equal to `'category'`, all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of `categories` or `ordered`.

2.8.3 Description

Using `describe()` on categorical data will produce similar output to a `Series` or `DataFrame` of type `string`.

```
In [53]: cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])

In [54]: df = pd.DataFrame({"cat": cat, "s": ["a", "c", "c", np.nan]})

In [55]: df.describe()
Out[55]:
       cat  s
count    3  3
unique   2  2
top      c  c
freq     2  2

In [56]: df["cat"].describe()
Out[56]:
count     3
unique    2
top       c
freq      2
Name: cat, dtype: object
```

2.8.4 Working with categories

Categorical data has a *categories* and a *ordered* property, which list their possible values and whether the ordering matters or not. These properties are exposed as `s.cat.categories` and `s.cat.ordered`. If you don't manually specify categories and ordering, they are inferred from the passed arguments.

```
In [57]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [58]: s.cat.categories
Out[58]: Index(['a', 'b', 'c'], dtype='object')

In [59]: s.cat.ordered
Out[59]: False
```

It's also possible to pass in the categories in a specific order:

```
In [60]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"],
   ....:                 categories=["c", "b", "a"]))
   ....:

In [61]: s.cat.categories
Out[61]: Index(['c', 'b', 'a'], dtype='object')

In [62]: s.cat.ordered
Out[62]: False
```

Note: New categorical data are **not** automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered `Categorical`.

Note: The result of *unique()* is not always the same as `Series.cat.categories`, because `Series.unique()` has a couple of guarantees, namely that it returns categories in the order of appearance, and it only includes values that are actually present.

```
In [63]: s = pd.Series(list('babc')).astype(CategoricalDtype(list('abcd')))

In [64]: s
Out[64]:
0    b
1    a
2    b
3    c
dtype: category
Categories (4, object): [a, b, c, d]

# categories
In [65]: s.cat.categories
Out[65]: Index(['a', 'b', 'c', 'd'], dtype='object')

# uniques
In [66]: s.unique()
Out[66]:
[b, a, c]
Categories (3, object): [b, a, c]
```

Renaming categories

Renaming categories is done by assigning new values to the `Series.cat.categories` property or by using the `rename_categories()` method:

```
In [67]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [68]: s
Out[68]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [69]: s.cat.categories = ["Group %s" % g for g in s.cat.categories]

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

(continues on next page)

(continued from previous page)

```
dtype: category
Categories (3, object): [Group a, Group b, Group c]

In [71]: s = s.cat.rename_categories([1, 2, 3])

In [72]: s
Out[72]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [1, 2, 3]

# You can also pass a dict-like object to map the renaming
In [73]: s = s.cat.rename_categories({1: 'x', 2: 'y', 3: 'z'})

In [74]: s
Out[74]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Note: In contrast to R's *factor*, categorical data can have categories of other types than string.

Note: Be aware that assigning new categories is an inplace operation, while most other operations under `Series.cat` per default return a new `Series` of dtype *category*.

Categories must be unique or a *ValueError* is raised:

```
In [75]: try:
   ....:     s.cat.categories = [1, 1, 1]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorical categories must be unique
```

Categories must also not be `NaN` or a *ValueError* is raised:

```
In [76]: try:
   ....:     s.cat.categories = [1, 2, np.nan]
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorial categories cannot be null
```

Appending new categories

Appending categories can be done by using the `add_categories()` method:

```
In [77]: s = s.cat.add_categories([4])

In [78]: s.cat.categories
Out[78]: Index(['x', 'y', 'z', 4], dtype='object')

In [79]: s
Out[79]:
0    x
1    y
2    z
3    x
dtype: category
Categories (4, object): [x, y, z, 4]
```

Removing categories

Removing categories can be done by using the `remove_categories()` method. Values which are removed are replaced by `np.nan`.:

```
In [80]: s = s.cat.remove_categories([4])

In [81]: s
Out[81]:
0    x
1    y
2    z
3    x
dtype: category
Categories (3, object): [x, y, z]
```

Removing unused categories

Removing unused categories can also be done:

```
In [82]: s = pd.Series(pd.Categorical(["a", "b", "a"],
   ....:                 categories=["a", "b", "c", "d"]))
   ....:

In [83]: s
Out[83]:
0    a
1    b
2    a
dtype: category
Categories (4, object): [a, b, c, d]

In [84]: s.cat.remove_unused_categories()
Out[84]:
0    a
1    b
2    a
```

(continues on next page)

(continued from previous page)

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

Setting categories

If you want to do remove and add new categories in one step (which has some speed advantage), or simply set the categories to a predefined scale, use `set_categories()`.

```
In [85]: s = pd.Series(["one", "two", "four", "-"], dtype="category")

In [86]: s
Out[86]:
0     one
1     two
2    four
3       -
dtype: category
Categories (4, object): [-, four, one, two]

In [87]: s = s.cat.set_categories(["one", "two", "three", "four"])

In [88]: s
Out[88]:
0     one
1     two
2    four
3     NaN
dtype: category
Categories (4, object): [one, two, three, four]
```

Note: Be aware that `Categorical.set_categories()` cannot know whether some category is omitted intentionally or because it is misspelled or (under Python3) due to a type difference (e.g., NumPy S1 dtype and Python strings). This can result in surprising behaviour!

2.8.5 Sorting and order

If categorical data is ordered (`s.cat.ordered == True`), then the order of the categories has a meaning and certain operations are possible. If the categorical is unordered, `.min()/.max()` will raise a `TypeError`.

```
In [89]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], ordered=False))

In [90]: s.sort_values(inplace=True)

In [91]: s = pd.Series(["a", "b", "c", "a"]).astype(
   ....:     CategoricalDtype(ordered=True)
   ....: )
   ....:

In [92]: s.sort_values(inplace=True)

In [93]: s
```

(continues on next page)

(continued from previous page)

```
Out[93]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [94]: s.min(), s.max()
Out[94]: ('a', 'c')
```

You can set categorical data to be ordered by using `as_ordered()` or unordered by using `as_unordered()`. These will by default return a *new* object.

```
In [95]: s.cat.as_ordered()
Out[95]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a < b < c]

In [96]: s.cat.as_unordered()
Out[96]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): [a, b, c]
```

Sorting will use the order defined by categories, not any lexical order present on the data type. This is even true for strings and numeric data:

```
In [97]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [98]: s = s.cat.set_categories([2, 3, 1], ordered=True)

In [99]: s
Out[99]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [100]: s.sort_values(inplace=True)

In [101]: s
Out[101]:
1    2
2    3
0    1
3    1
```

(continues on next page)

(continued from previous page)

```
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [102]: s.min(), s.max()
Out[102]: (2, 1)
```

Reordering

Reordering the categories is possible via the `Categorical.reorder_categories()` and the `Categorical.set_categories()` methods. For `Categorical.reorder_categories()`, all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [103]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [104]: s = s.cat.reorder_categories([2, 3, 1], ordered=True)

In [105]: s
Out[105]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [106]: s.sort_values(inplace=True)

In [107]: s
Out[107]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [108]: s.min(), s.max()
Out[108]: (2, 1)
```

Note: Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the `Series`, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the `Series` are changed.

Note: If the `Categorical` is not ordered, *`Series.min()`* and *`Series.max()`* will raise `TypeError`. Numeric operations like `+`, `-`, `*`, `/` and operations based on them (e.g. *`Series.median()`*, which would need to compute the mean between two values if the length of an array is even) do not work and raise a `TypeError`.

Multi column sorting

A categorical dtyped column will participate in a multi-column sort in a similar manner to other columns. The ordering of the categorical is determined by the `categories` of that column.

```
In [109]: dfs = pd.DataFrame({'A': pd.Categorical(list('bbeebbaa'),
   .....:                                         categories=['e', 'a', 'b'],
   .....:                                         ordered=True),
   .....:                     'B': [1, 2, 1, 2, 2, 1, 2, 1]})
   .....:

In [110]: dfs.sort_values(by=['A', 'B'])
Out[110]:
   A  B
2  e  1
3  e  2
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
```

Reordering the `categories` changes a future sort.

```
In [111]: dfs['A'] = dfs['A'].cat.reorder_categories(['a', 'b', 'e'])

In [112]: dfs.sort_values(by=['A', 'B'])
Out[112]:
   A  B
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
2  e  1
3  e  2
```

2.8.6 Comparisons

Comparing categorical data with other objects is possible in three cases:

- Comparing equality (`==` and `!=`) to a list-like object (list, Series, array, . . .) of the same length as the categorical data.
- All comparisons (`==`, `!=`, `>`, `>=`, `<`, and `<=`) of categorical data to another categorical Series, when `ordered==True` and the *categories* are the same.
- All comparisons of a categorical data to a scalar.

All other comparisons, especially "non-equality" comparisons of two categoricals with different categories or a categorical with any list-like object, will raise a `TypeError`.

Note: Any "non-equality" comparisons of categorical data with a `Series`, `np.array`, `list` or categorical data with different categories or ordering will raise a `TypeError` because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [113]: cat = pd.Series([1, 2, 3]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
   .....:

In [114]: cat_base = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype([3, 2, 1], ordered=True)
   .....: )
   .....:

In [115]: cat_base2 = pd.Series([2, 2, 2]).astype(
   .....:     CategoricalDtype(ordered=True)
   .....: )
   .....:

In [116]: cat
Out[116]:
0    1
1    2
2    3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [117]: cat_base
Out[117]:
0    2
1    2
2    2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [118]: cat_base2
Out[118]:
0    2
1    2
2    2
dtype: category
Categories (1, int64): [2]
```

Comparing to a categorical with the same categories and ordering or to a scalar works:

```
In [119]: cat > cat_base
Out[119]:
0     True
1    False
2    False
dtype: bool

In [120]: cat > 2
Out[120]:
0     True
1    False
2    False
dtype: bool
```

Equality comparisons work with any list-like object of same length and scalars:

```
In [121]: cat == cat_base
Out[121]:
0    False
1     True
2    False
dtype: bool

In [122]: cat == np.array([1, 2, 3])
Out[122]:
0    True
1    True
2    True
dtype: bool

In [123]: cat == 2
Out[123]:
0    False
1     True
2    False
dtype: bool
```

This doesn't work because the categories are not the same:

```
In [124]: try:
   .....:     cat > cat_base2
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categoricals can only be compared if 'categories' are the same. Categories␣
↪are different lengths
```

If you want to do a "non-equality" comparison of a categorical series with a list-like object which is not categorical data, you need to be explicit and convert the categorical data back to the original values:

```
In [125]: base = np.array([1, 2, 3])

In [126]: try:
   .....:     cat > base
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Cannot compare a Categorical for op __gt__ with type <class 'numpy.ndarray
↪'>.
If you want to compare values, use 'np.asarray(cat) <op> other'.

In [127]: np.asarray(cat) > base
Out[127]: array([False, False, False])
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```
In [128]: c1 = pd.Categorical(['a', 'b'], categories=['a', 'b'], ordered=False)

In [129]: c2 = pd.Categorical(['a', 'b'], categories=['b', 'a'], ordered=False)

In [130]: c1 == c2
Out[130]: array([ True,  True])
```

2.8.7 Operations

Apart from *Series.min()*, *Series.max()* and *Series.mode()*, the following operations are possible with categorical data:

Series methods like *Series.value_counts()* will use all categories, even if some categories are not present in the data:

```
In [131]: s = pd.Series(pd.Categorical(["a", "b", "c", "c"],
   .....:                 categories=["c", "a", "b", "d"]))
   .....:

In [132]: s.value_counts()
Out[132]:
c    2
b    1
a    1
d    0
dtype: int64
```

Groupby will also show "unused" categories:

```
In [133]: cats = pd.Categorical(["a", "b", "b", "b", "c", "c", "c"],
   .....:                        categories=["a", "b", "c", "d"])
   .....:

In [134]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 2, 3, 4, 5]})

In [135]: df.groupby("cats").mean()
Out[135]:
      values
cats
a        1.0
b        2.0
c        4.0
d        NaN

In [136]: cats2 = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [137]: df2 = pd.DataFrame({"cats": cats2,
   .....:                     "B": ["c", "d", "c", "d"],
   .....:                     "values": [1, 2, 3, 4]})
   .....:

In [138]: df2.groupby(["cats", "B"]).mean()
Out[138]:
        values
cats B
a    c     1.0
     d     2.0
b    c     3.0
     d     4.0
c    c     NaN
     d     NaN
```

Pivot tables:

```
In [139]: raw_cat = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [140]: df = pd.DataFrame({"A": raw_cat,
   .....:                    "B": ["c", "d", "c", "d"],
   .....:                    "values": [1, 2, 3, 4]})
   .....:

In [141]: pd.pivot_table(df, values='values', index=['A', 'B'])
Out[141]:
     values
A B
a c       1
  d       2
b c       3
  d       4
```

2.8.8 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.at`, and `.iat`, work as normal. The only difference is the return type (for getting) and that only values already in *categories* can be assigned.

Getting

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.

```
In [142]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [143]: cats = pd.Series(["a", "b", "b", "b", "c", "c", "c"],
   .....:                  dtype="category", index=idx)
   .....:

In [144]: values = [1, 2, 2, 2, 3, 4, 5]

In [145]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [146]: df.iloc[2:4, :]
Out[146]:
  cats  values
j    b       2
k    b       2

In [147]: df.iloc[2:4, :].dtypes
Out[147]:
cats      category
values       int64
dtype: object

In [148]: df.loc["h":"j", "cats"]
Out[148]:
h    a
i    b
j    b
Name: cats, dtype: category
Categories (3, object): [a, b, c]
```

(continues on next page)

(continued from previous page)

```
In [149]: df[df["cats"] == "b"]
Out[149]:
  cats  values
i    b       2
j    b       2
k    b       2
```

An example where the category type is not preserved is if you take one single row: the resulting `Series` is of dtype `object`:

```
# get the complete "h" row as a Series
In [150]: df.loc["h", :]
Out[150]:
cats      a
values    1
Name: h, dtype: object
```

Returning a single item from categorical data will also return the value, not a categorical of length "1".

```
In [151]: df.iat[0, 0]
Out[151]: 'a'

In [152]: df["cats"].cat.categories = ["x", "y", "z"]

In [153]: df.at["h", "cats"]  # returns a string
Out[153]: 'x'
```

Note: The is in contrast to R's *factor* function, where `factor(c(1,2,3))[1]` returns a single value *factor*.

To get a single value `Series` of type `category`, you pass in a list with a single value:

```
In [154]: df.loc[["h"], "cats"]
Out[154]:
h    x
Name: cats, dtype: category
Categories (3, object): [x, y, z]
```

String and datetime accessors

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```
In [155]: str_s = pd.Series(list('aabb'))

In [156]: str_cat = str_s.astype('category')

In [157]: str_cat
Out[157]:
0    a
1    a
2    b
3    b
dtype: category
```

(continues on next page)