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.

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

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)

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.

CategoricalDtype

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`.

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
```

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
```

Update D4 Accessor: DataFrame is now integrated with the categorical properties. Due to DataFrame being a multi-dimensional object, `df.cat.categories` and `df.cat.ordered` becomes a column-wise operation. `df.cat.categories` will give a dictionary of the category's index while `df.cat.ordered` will give the columns that are categorical and are ordered.

```
In [63]: df = pd.DataFrame()

In [64]: df['s1'] = pd.Series(pd.Categorical(["a", "b", "c", "a"], categories=["c", "b", "a"], ordered=True))

In [65]: df['s2'] = pd.Series(pd.Categorical(["a", "b", "c", "a"], categories=["c", "b", "a"]))

In [66]: df.cat.categories
Out[66]:
{'s1': Index(['c', 'b', 'a'], dtype='object'),
 's2': Index(['c', 'b', 'a'], dtype='object')}

In [67]: df.cat.ordered
Out[67]:
  s1
0  a
```

```
1  b
2  c
3  a
```

In addition, we provided two extra properties `df.cat.all` and `df.cat.unordered` for categorical column extraction. These properties are intuitive to create cleaner code and abstract away the tedious task of order/unordered categorical column extraction.

```
In [68]: df.cat.all
Out[68]:
  s1 s2
0  a  a
1  b  b
2  c  c
3  a  a

In [69]: df.cat.unordered
Out[69]:
  s2
0  a
1  b
2  c
3  a
```

Note

The new accessor `df.cat` also delegates the `categorical` methods such as `rename_categories`. You will see examples later on of how they are used in `Series`. For `df.cat`, the operation is delegated to all the categorical columns. If you would like to only apply this to a specific subset, you can always use the existing `filter` and `loc` methods. Methods such as `remove_unused_categories` are direct beneficiaries of this addition as the user no longer needs to write manual loops to apply them one by one.

This delegation is limited in the sense that there is yet to be an error suppressing. If one column fails then all fail. In the future, a suppression parameter might need to be added or new methods being added to the delegation will need to be aware of this issue.

```
# Example usage
In [70]: unused_df = pd.DataFrame()

In [71]: unused_df['s1'] = pd.Series(pd.Categorical(["a", "b", "a"], categories=["a", "b", "c", "d"]))

In [72]: unused_df['s2'] = pd.Series(pd.Categorical(["a", "b", "a"], categories=["a", "b", "c", "d"]))

In [73]: clean_df = unused_df.cat.remove_unused_categories()

In [74]: clean_df['s1']
Out[74]:
0     a
1     b
2     a
Name: s1, dtype: category
Categories (2, object): ['a', 'b']

In [75]: clean_df['s2']
Out[75]:
0     a
1     b
2     a
Name: s2, dtype: category
Categories (2, object): ['a', 'b']
```

END Update D4 Accessor

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 [76]: s = pd.Series(list("babc")).astype(CategoricalDtype(list("abcd")))

In [77]: s
Out[77]:
0    b
1    a
2    b
3    c
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

# categories
In [78]: s.cat.categories
Out[78]: Index(['a', 'b', 'c', 'd'], dtype='object')

# uniques
In [79]: s.unique()
Out[79]:
['b', 'a', 'c']
Categories (4, object): ['a', 'b', 'c', 'd']
```

Renaming categories

Renaming categories is done by using the `rename_categories()` method:

```
In [80]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [81]: s
Out[81]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): ['a', 'b', 'c']

In [82]: new_categories = ["Group %s" % g for g in s.cat.categories]

In [83]: s = s.cat.rename_categories(new_categories)

In [84]: s
Out[84]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (3, object): ['Group a', 'Group b', 'Group c']

# You can also pass a dict-like object to map the renaming
In [85]: s = s.cat.rename_categories({1: "x", 2: "y", 3: "z"})

In [86]: s
Out[86]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (3, object): ['Group a', 'Group b', 'Group c']
```

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 [87]: try:
   ....:     s = s.cat.rename_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 [88]: try:
   ....:     s = s.cat.rename_categories([1, 2, np.nan])
   ....: except ValueError as e:
   ....:     print("ValueError:", str(e))
   ....:
ValueError: Categorical categories cannot be null
```

Appending new categories

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

```
In [89]: s = s.cat.add_categories([4])

In [90]: s.cat.categories
Out[90]: Index(['Group a', 'Group b', 'Group c', 4], dtype='object')

In [91]: s
Out[91]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (4, object): ['Group a', 'Group b', 'Group c', 4]
```

Removing categories

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

```
In [92]: s = s.cat.remove_categories([4])

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

Removing unused categories

Removing unused categories can also be done:

```
In [94]: s = pd.Series(pd.Categorical(["a", "b", "a"], categories=["a", "b", "c", "d"]))

In [95]: s
Out[95]:
0    a
1    b
2    a
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

In [96]: s.cat.remove_unused_categories()
Out[96]:
0    a
1    b
2    a
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 [97]: s = pd.Series(["one", "two", "four", "-"], dtype="category")

In [98]: s
Out[98]:
0     one
1     two
2    four
3       -
dtype: category
Categories (4, object): ['-', 'four', 'one', 'two']

In [99]: s = s.cat.set_categories(["one", "two", "three", "four"])

In [100]: s
Out[100]:
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!

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

In [102]: s.sort_values(inplace=True)

In [103]: s = pd.Series(["a", "b", "c", "a"]).astype(CategoricalDtype(ordered=True))

In [104]: s.sort_values(inplace=True)

In [105]: s
Out[105]:
0    a
3    a
1    b
2    c
dtype: category
Categories (3, object): ['a' < 'b' < 'c']

In [106]: s.min(), s.max()
Out[106]: ('a', 'c')
```

Update D4 type checking: You can check whether an object is ordered using functions `.api.types.is_ordered_categorical_dtype()` and `.api.types.is_unordered_categorical_dtype()`. These functions will return booleans to indicate whether the dtype is ordered or not. For objects that are not in categorical dtype, both functions will return `False`.

```
In [107]: cat_dtype = pd.CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [108]: pd.api.types.is_ordered_categorical_dtype(cat_dtype)
Out[108]: True

In [109]: pd.api.types.is_unordered_categorical_dtype(cat_dtype)
Out[109]: False
```

```
In [110]: s = pd.Series(['a', 'b', 'c', 'd', 'e', 'f'])

In [111]: pd.api.types.is_ordered_categorical_dtype(s)
Out[111]: False

In [112]: pd.api.types.is_unordered_categorical_dtype(s)
Out[112]: False

In [113]: s = s.astype('category')

In [114]: pd.api.types.is_ordered_categorical_dtype(s)
Out[114]: False

In [115]: pd.api.types.is_unordered_categorical_dtype(s)
Out[115]: True
```

This interface abstracts away the inner details of verifying the dtypes when querying for categorical dtypes. It provides an alternative to users who do not want to get an entire DataFrame but only the ordering of the columns. Originally, they have to create their own function/lambda but with the new type-checking functions, they can do it in one line. Furthermore, this is also used to reduce duplicate code in the `df.cat` accessor's ordering properties.

```
# Original tedious solution
In [116]: categories = df.select_dtypes("category")

In [117]: categories[[col for col in categories.columns if categories[col].cat.ordered]]
Out[117]:
  s1
0  a
1  b
2  c
3  a

# New one liner with much cleaner code
In [118]: df.loc[:, df.apply(pd.api.types.is_ordered_categorical_dtype)]
Out[118]:
  s1
0  a
1  b
2  c
3  a
```

Note

`df.cat` does not use the apply approach. The implementation does not adopt it as apply is known for being slow, and it passes `None` into the lambdas which result in unwanted behavior.

END Update D4 type checking

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 [119]: s.cat.as_ordered()
Out[119]:
0    a
1    b
2    c
3    d
4    e
5    f
dtype: category
Categories (6, object): ['a' < 'b' < 'c' < 'd' < 'e' < 'f']

In [120]: s.cat.as_unordered()
Out[120]:
0    a
1    b
2    c
3    d
4    e
5    f
dtype: category
Categories (6, object): ['a', 'b', 'c', 'd', 'e', 'f']
```

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

In [122]: s = s.cat.set_categories([2, 3, 1], ordered=True)

In [123]: s
Out[123]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [124]: s.sort_values(inplace=True)

In [125]: s
Out[125]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [126]: s.min(), s.max()
Out[126]: (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 [127]: s = pd.Series([1, 2, 3, 1], dtype="category")

In [128]: s = s.cat.reorder_categories([2, 3, 1], ordered=True)

In [129]: s
Out[129]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [130]: s.sort_values(inplace=True)

In [131]: s
Out[131]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [132]: s.min(), s.max()
Out[132]: (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 [133]: dfs = pd.DataFrame(
   .....:     {
   .....:         "A": pd.Categorical(
   .....:             list("bbeebbaa"),
   .....:             categories=["e", "a", "b"],
   .....:             ordered=True,
   .....:         ),
   .....:         "B": [1, 2, 1, 2, 2, 1, 2, 1],
   .....:     }
   .....: )
   .....:

In [134]: dfs.sort_values(by=["A", "B"])
Out[134]:
   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 [135]: dfs["A"] = dfs["A"].cat.reorder_categories(["a", "b", "e"])

In [136]: dfs.sort_values(by=["A", "B"])
Out[136]:
   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
```

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

In [138]: cat_base = pd.Series([2, 2, 2]).astype(CategoricalDtype([3, 2, 1], ordered=True))

In [139]: cat_base2 = pd.Series([2, 2, 2]).astype(CategoricalDtype(ordered=True))

In [140]: cat
```

```
Out[140]:
0    1
1    2
2    3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [141]: cat_base
Out[141]:
0    2
1    2
2    2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [142]: cat_base2
Out[142]:
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 [143]: cat > cat_base
Out[143]:
0     True
1    False
2    False
dtype: bool

In [144]: cat > 2
Out[144]:
0     True
1    False
2    False
dtype: bool
```

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

```
In [145]: cat == cat_base
Out[145]:
0    False
1     True
2    False
dtype: bool

In [146]: cat == np.array([1, 2, 3])
Out[146]:
0    True
1    True
2    True
dtype: bool

In [147]: cat == 2
Out[147]:
0    False
1     True
2    False
dtype: bool
```

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

```
In [148]: try:
   .....:     cat > cat_base2
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Categoricals can only be compared if 'categories' are the same.
```

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 [149]: base = np.array([1, 2, 3])

In [150]: 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 [151]: np.asarray(cat) > base
Out[151]: array([False, False, False])
```

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

```
In [152]: c1 = pd.Categorical(["a", "b"], categories=["a", "b"], ordered=False)

In [153]: c2 = pd.Categorical(["a", "b"], categories=["b", "a"], ordered=False)

In [154]: c1 == c2
Out[154]: array([ True,  True])
```

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

In [156]: s.value_counts()
Out[156]:
c    2
a    1
b    1
d    0
dtype: int64
```

`DataFrame` methods like `DataFrame.sum()` also show "unused" categories.

```
In [157]: columns = pd.Categorical(
   .....:     ["One", "One", "Two"], categories=["One", "Two", "Three"], ordered=True
   .....: )
   .....:

In [158]: df = pd.DataFrame(
   .....:     data=[[1, 2, 3], [4, 5, 6]],
   .....:     columns=pd.MultiIndex.from_arrays([["A", "B", "B"], columns]),
   .....: )
   .....:

In [159]: df.groupby(axis=1, level=1).sum()
Out[159]:
   One  Two  Three
0    3    3      0
1    9    6      0
```

Groupby will also show "unused" categories:

```
In [160]: cats = pd.Categorical(
   .....:     ["a", "b", "b", "b", "c", "c", "c"], categories=["a", "b", "c", "d"]
   .....: )
   .....:

In [161]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 2, 3, 4, 5]})

In [162]: df.groupby("cats").mean()
Out[162]:
      values
cats
a        1.0
b        2.0
c        4.0
d        NaN

In [163]: cats2 = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [164]: df2 = pd.DataFrame(
   .....:     {
   .....:         "cats": cats2,
   .....:         "B": ["c", "d", "c", "d"],
   .....:         "values": [1, 2, 3, 4],
   .....:     }
   .....: )
   .....:

In [165]: df2.groupby(["cats", "B"]).mean()
Out[165]:
        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 [166]: raw_cat = pd.Categorical(["a", "a", "b", "b"], categories=["a", "b", "c"])

In [167]: df = pd.DataFrame({"A": raw_cat, "B": ["c", "d", "c", "d"], "values": [1, 2, 3, 4]})

In [168]: pd.pivot_table(df, values="values", index=["A", "B"])
Out[168]:
     values
A B
a c       1
  d       2
b c       3
  d       4
```

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

In [170]: cats = pd.Series(["a", "b", "b", "b", "c", "c", "c"], dtype="category", index=idx)

In [171]: values = [1, 2, 2, 2, 3, 4, 5]

In [172]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [173]: df.iloc[2:4, :]
Out[173]:
  cats  values
j    b       2
k    b       2

In [174]: df.iloc[2:4, :].dtypes
Out[174]:
cats      category
values       int64
dtype: object

In [175]: df.loc["h":"j", "cats"]
Out[175]:
h    a
i    b
j    b
Name: cats, dtype: category
Categories (3, object): ['a', 'b', 'c']

In [176]: df[df["cats"] == "b"]
Out[176]:
  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 [177]: df.loc["h", :]
Out[177]:
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 [178]: df.iat[0, 0]
Out[178]: 'a'

In [179]: df["cats"] = df["cats"].cat.rename_categories(["x", "y", "z"])

In [180]: df.at["h", "cats"]  # returns a string
Out[180]: '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 [181]: df.loc[["h"], "cats"]
Out[181]:
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 [182]: str_s = pd.Series(list("aabb"))

In [183]: str_cat = str_s.astype("category")

In [184]: str_cat
Out[184]:
0    a
1    a
2    b
3    b
dtype: category
Categories (2, object): ['a', 'b']

In [185]: str_cat.str.contains("a")
Out[185]:
0     True
1     True
2    False
3    False
dtype: bool

In [186]: date_s = pd.Series(pd.date_range("1/1/2015", periods=5))

In [187]: date_cat = date_s.astype("category")

In [188]: date_cat
Out[188]:
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 [189]: date_cat.dt.day
Out[189]:
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 [190]: ret_s = str_s.str.contains("a")

In [191]: ret_cat = str_cat.str.contains("a")

In [192]: ret_s.dtype == ret_cat.dtype
Out[192]: True

In [193]: ret_s == ret_cat
Out[193]:
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 [194]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

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

In [196]: values = [1, 1, 1, 1, 1, 1, 1]

In [197]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [198]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

In [199]: df
Out[199]:
  cats  values
h    a       1
i    a       1
j    b       2
k    b       2
l    a       1
m    a       1
n    a       1

In [200]: try:
   .....:     df.iloc[2:4, :] = [["c", 3], ["c", 3]]
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: 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 [201]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"], categories=["a", "b"])

In [202]: df
```

```
Out[202]:
  cats  values
h    a       1
i    a       1
j    a       2
k    a       2
l    a       1
m    a       1
n    a       1

In [203]: try:
   .....:     df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"], categories=["a", "b", "c"])
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: 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 [204]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a", "a"]})

In [205]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [206]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [207]: df
Out[207]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a

In [208]: df.dtypes
Out[208]:
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 [209]: from pandas.api.types import union_categoricals

# same categories
In [210]: s1 = pd.Series(["a", "b"], dtype="category")

In [211]: s2 = pd.Series(["a", "b", "a"], dtype="category")

In [212]: pd.concat([s1, s2])
Out[212]:
0    a
1    b
0    a
1    b
2    a
dtype: category
Categories (2, object): ['a', 'b']

# different categories
In [213]: s3 = pd.Series(["b", "c"], dtype="category")

In [214]: pd.concat([s1, s3])
Out[214]:
0    a
1    b
0    b
1    c
dtype: object

# Output dtype is inferred based on categories values
In [215]: int_cats = pd.Series([1, 2], dtype="category")

In [216]: float_cats = pd.Series([3.0, 4.0], dtype="category")

In [217]: pd.concat([int_cats, float_cats])
Out[217]:
```

```
0    1.0
1    2.0
0    3.0
1    4.0
dtype: float64

In [218]: pd.concat([s1, s3]).astype("category")
Out[218]:
0    a
1    b
0    b
1    c
dtype: category
Categories (3, object): ['a', 'b', 'c']

In [219]: union_categoricals([s1.array, s3.array])
Out[219]:
['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 [220]: from pandas.api.types import union_categoricals

In [221]: a = pd.Categorical(["b", "c"])

In [222]: b = pd.Categorical(["a", "b"])

In [223]: union_categoricals([a, b])
Out[223]:
['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 [224]: union_categoricals([a, b], sort_categories=True)
Out[224]:
['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 [225]: a = pd.Categorical(["a", "b"], ordered=True)

In [226]: b = pd.Categorical(["a", "b", "a"], ordered=True)

In [227]: union_categoricals([a, b])
Out[227]:
['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 [228]: a = pd.Categorical(["a", "b", "c"], ordered=True)

In [229]: b = pd.Categorical(["c", "b", "a"], ordered=True)

In [230]: union_categoricals([a, b], ignore_order=True)
Out[230]:
['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 [231]: a = pd.Series(["b", "c"], dtype="category")

In [232]: b = pd.Series(["a", "b"], dtype="category")

In [233]: union_categoricals([a, b])
Out[233]:
['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 [234]: c1 = pd.Categorical(["b", "c"])

In [235]: c2 = pd.Categorical(["a", "b"])

In [236]: c1
Out[236]:
['b', 'c']
Categories (2, object): ['b', 'c']

# "b" is coded to 0
In [237]: c1.codes
Out[237]: array([0, 1], dtype=int8)

In [238]: c2
Out[238]:
['a', 'b']
Categories (2, object): ['a', 'b']

# "b" is coded to 1
In [239]: c2.codes
Out[239]: array([0, 1], dtype=int8)

In [240]: c = union_categoricals([c1, c2])

In [241]: c
Out[241]:
['b', 'c', 'a', 'b']
Categories (3, object): ['b', 'c', 'a']

# "b" is coded to 0 throughout, same as c1, different from c2
In [242]: c.codes
Out[242]: array([0, 1, 2, 0], dtype=int8)
```

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 [243]: import io

In [244]: s = pd.Series(pd.Categorical(["a", "b", "b", "a", "a", "d"]))

# rename the categories
In [245]: s = s.cat.rename_categories(["very good", "good", "bad"])

# reorder the categories and add missing categories
In [246]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [247]: df = pd.DataFrame({"cats": s, "vals": [1, 2, 3, 4, 5, 6]})

In [248]: csv = io.StringIO()

In [249]: df.to_csv(csv)

In [250]: df2 = pd.read_csv(io.StringIO(csv.getvalue()))

In [251]: df2.dtypes
Out[251]:
Unnamed: 0     int64
cats          object
vals           int64
dtype: object

In [252]: df2["cats"]
Out[252]:
0    very good
1         good
2         good
3    very good
4    very good
5          bad
Name: cats, dtype: object

# Redo the category
In [253]: df2["cats"] = df2["cats"].astype("category")

In [254]: df2["cats"].cat.set_categories(
   .....:     ["very bad", "bad", "medium", "good", "very good"], inplace=True
   .....: )
   .....:

In [255]: df2.dtypes
Out[255]:
Unnamed: 0       int64
cats          category
vals             int64
dtype: object

In [256]: df2["cats"]
Out[256]:
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`.

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

# only two categories
```

```
Out[258]:
0      a
1      b
2    NaN
3      a
dtype: category
Categories (2, object): ['a', 'b']

In [259]: s.cat.codes
Out[259]:
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 [260]: s = pd.Series(["a", "b", np.nan], dtype="category")

In [261]: s
Out[261]:
0      a
1      b
2    NaN
dtype: category
Categories (2, object): ['a', 'b']

In [262]: pd.isna(s)
Out[262]:
0    False
1    False
2     True
dtype: bool

In [263]: s.fillna("a")
Out[263]:
0    a
1    b
2    a
dtype: category
Categories (2, object): ['a', 'b']
```

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`.

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 [264]: s = pd.Series(["foo", "bar"] * 1000)

# object dtype
In [265]: s.nbytes
Out[265]: 16000

# category dtype
In [266]: s.astype("category").nbytes
In [266]: s.astype("category").nbytes
Out[266]: 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 [267]: s = pd.Series(["foo%04d" % i for i in range(2000)])

# object dtype
In [268]: s.nbytes
Out[268]: 16000

# category dtype
In [269]: s.astype("category").nbytes
Out[269]: 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 [270]: try:
   .....:     np.dtype("category")
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: data type 'category' not understood

In [271]: dtype = pd.Categorical(["a"]).dtype

In [272]: try:
   .....:     np.dtype(dtype)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: Cannot interpret 'CategoricalDtype(categories=['a'], ordered=False)' as a data type
```

Dtype comparisons work:

```
In [273]: dtype == np.str_
Out[273]: False

In [274]: np.str_ == dtype
Out[274]: False
```

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

```
In [275]: hasattr(pd.Series(["a"], dtype="category"), "cat")
Out[275]: True

In [276]: hasattr(pd.Series(["a"]), "cat")
Out[276]: 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 [277]: s = pd.Series(pd.Categorical([1, 2, 3, 4]))

In [278]: try:
   .....:     np.sum(s)
   .....: except TypeError as e:
   .....:     print("TypeError:", str(e))
   .....:
TypeError: 'Categorical' with dtype category does not support reduction '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 [279]: df = pd.DataFrame(
   .....:     {
   .....:         "a": [1, 2, 3, 4],
   .....:         "b": ["a", "b", "c", "d"],
   .....:         "cats": pd.Categorical([1, 2, 3, 2]),
   .....:     }
   .....: )
   .....:

In [280]: df.apply(lambda row: type(row["cats"]), axis=1)
Out[280]:
0    <class 'int'>
1    <class 'int'>
2    <class 'int'>
3    <class 'int'>
dtype: object

In [281]: df.apply(lambda col: col.dtype, axis=0)
Out[281]:
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 [282]: cats = pd.Categorical([1, 2, 3, 4], categories=[4, 2, 3, 1])

In [283]: strings = ["a", "b", "c", "d"]

In [284]: values = [4, 2, 3, 1]

In [285]: df = pd.DataFrame({"strings": strings, "values": values}, index=cats)

In [286]: df.index
Out[286]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False, dtype='category')

# This now sorts by the categories order
In [287]: df.sort_index()
Out[287]:
  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 [288]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [289]: s = pd.Series(cat, name="cat")
In [290]: cat
Out[290]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [291]: s.iloc[0:2] = 10

In [292]: cat
Out[292]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [293]: df = pd.DataFrame(s)

In [294]: df["cat"].cat.categories = [1, 2, 3, 4, 5]

In [295]: cat
Out[295]:
[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 [296]: cat = pd.Categorical([1, 2, 3, 10], categories=[1, 2, 3, 4, 10])

In [297]: s = pd.Series(cat, name="cat", copy=True)

In [298]: cat
Out[298]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [299]: s.iloc[0:2] = 10

In [300]: cat
Out[300]:
[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.

Previous
pandas documentation