This class does not inherit from 'abc.ABCMeta' for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

Attributes

kind	A character code (one of 'biufcmMOSUV'), default 'O'
na_value	Default NA value to use for this type.
name	A string identifying the data type.
names	Ordered list of field names, or None if there are no fields.
type	The scalar type for the array, e.g.

pandas.api.extensions.ExtensionDtype.kind

property ExtensionDtype.**kind**
A character code (one of 'biufcmMOSUV'), default 'O'

This should match the NumPy dtype used when the array is converted to an ndarray, which is probably 'O' for object if the extension type cannot be represented as a built-in NumPy type.

See also:

numpy.dtype.kind

pandas.api.extensions.ExtensionDtype.na_value

property ExtensionDtype.**na_value**
Default NA value to use for this type.

This is used in e.g. ExtensionArray.take. This should be the user-facing "boxed" version of the NA value, not the physical NA value for storage. e.g. for JSONArray, this is an empty dictionary.

pandas.api.extensions.ExtensionDtype.name

property ExtensionDtype.**name**
A string identifying the data type.

Will be used for display in, e.g. `Series.dtype`

pandas.api.extensions.ExtensionDtype.names

property ExtensionDtype.**names**
Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

pandas.api.extensions.ExtensionDtype.type

property ExtensionDtype.**type**
The scalar type for the array, e.g. `int`

It's expected `ExtensionArray[item]` returns an instance of `ExtensionDtype.type` for scalar `item`, assuming that value is valid (not NA). NA values do not need to be instances of *type*.

Methods

construct_array_type()	Return the array type associated with this dtype.
construct_from_string(string)	Construct this type from a string.
is_dtype(dtype)	Check if we match 'dtype'.

pandas.api.extensions.ExtensionDtype.construct_array_type

classmethod ExtensionDtype.**construct_array_type**()
Return the array type associated with this dtype.

Returns

type

pandas.api.extensions.ExtensionDtype.construct_from_string

classmethod ExtensionDtype.**construct_from_string**(*string: str*)
Construct this type from a string.

This is useful mainly for data types that accept parameters. For example, a period dtype accepts a frequency parameter that can be set as `period[H]` (where H means hourly frequency).

By default, in the abstract class, just the name of the type is expected. But subclasses can overwrite this method to accept parameters.

Parameters

string [str] The name of the type, for example `category`.

Returns

ExtensionDtype Instance of the dtype.

Raises

TypeError If a class cannot be constructed from this 'string'.

Examples

For extension dtypes with arguments the following may be an adequate implementation.

```
>>> @classmethod
... def construct_from_string(cls, string):
...     pattern = re.compile(r"^my_type\[(?P<arg_name>.+)\]$")
...     match = pattern.match(string)
...     if match:
...         return cls(**match.groupdict())
...     else:
...         raise TypeError(f"Cannot construct a '{cls.__name__}' from
...             " "'{string}'")
```

pandas.api.extensions.ExtensionDtype.is_dtype

classmethod ExtensionDtype.**is_dtype**(*dtype*) → bool

Check if we match 'dtype'.

Parameters

dtype [object] The object to check.

Returns

is_dtype [bool]

Notes

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

api.extensions.ExtensionArray()	Abstract base class for custom 1-D array types.
arrays.PandasArray(values, copy)	PandasArray], A pandas ExtensionArray for NumPy data.

3.16.6 pandas.api.extensions.ExtensionArray

class pandas.api.extensions.**ExtensionArray**

Abstract base class for custom 1-D array types.

pandas will recognize instances of this class as proper arrays with a custom type and will not attempt to coerce them to objects. They may be stored directly inside a `DataFrame` or `Series`.

New in version 0.23.0.

Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- _from_sequence
- _from_factorized
- __getitem__
- __len__
- dtype
- nbytes
- isna
- take
- copy
- _concat_same_type

A default repr displaying the type, (truncated) data, length, and dtype is provided. It can be customized or replaced by by overriding:

- __repr__ : A default repr for the ExtensionArray.
- _formatter : Print scalars inside a Series or DataFrame.

Some methods require casting the ExtensionArray to an ndarray of Python objects with `self.astype(object)`, which may be expensive. When performance is a concern, we highly recommend overriding the following methods:

- fillna
- dropna
- unique
- factorize / _values_for_factorize
- argsort / _values_for_argsort
- searchsorted

The remaining methods implemented on this class should be performant, as they only compose abstract methods. Still, a more efficient implementation may be available, and these methods can be overridden.

One can implement methods to handle array reductions.

- _reduce

One can implement methods to handle parsing from strings that will be used in methods such as `pandas.io.parsers.read_csv`.

- _from_sequence_of_strings

This class does not inherit from 'abc.ABCMeta' for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

ExtensionArrays are limited to 1 dimension.

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 address may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists. Pandas makes no assumptions on how the data are stored, just that it can be converted to a NumPy array. The ExtensionArray interface does not impose any rules on how this data is stored. However, currently, the backing data cannot be stored in attributes called `.values` or `._values` to ensure full compatibility with pandas internals. But other names as `.data`, `._data`, `._items`, ... can be freely used.

If implementing NumPy's `__array_ufunc__` interface, pandas expects that

1. You defer by returning `NotImplemented` when any Series are present in *inputs*. Pandas will extract the arrays and call the ufunc again.
2. You define a `_HANDLED_TYPES` tuple as an attribute on the class. Pandas inspect this to determine whether the ufunc is valid for the types present.

See *NumPy Universal Functions* for more.

Attributes

dtype	An instance of 'ExtensionDtype'.
nbytes	The number of bytes needed to store this object in memory.
ndim	Extension Arrays are only allowed to be 1-dimensional.
shape	Return a tuple of the array dimensions.

pandas.api.extensions.ExtensionArray.dtype

property ExtensionArray.**dtype**
An instance of 'ExtensionDtype'.

pandas.api.extensions.ExtensionArray.nbytes

property ExtensionArray.**nbytes**
The number of bytes needed to store this object in memory.

pandas.api.extensions.ExtensionArray.ndim

property ExtensionArray.**ndim**
Extension Arrays are only allowed to be 1-dimensional.

pandas.api.extensions.ExtensionArray.shape

property ExtensionArray.**shape**
Return a tuple of the array dimensions.

Methods

argsort(self, ascending, kind, *args, **kwargs)	Return the indices that would sort this array.
astype(self, dtype[, copy])	Cast to a NumPy array with 'dtype'.
copy(self)	Return a copy of the array.
dropna(self)	Return ExtensionArray without NA values.
factorize(self, na_sentinel)	Encode the extension array as an enumerated type.
fillna(self[, value, method, limit])	Fill NA/NaN values using the specified method.
isna(self)	A 1-D array indicating if each value is missing.
ravel(self[, order])	Return a flattened view on this array.
repeat(self, repeats[, axis])	Repeat elements of a ExtensionArray.
searchsorted(self, value[, side, sorter])	Find indices where elements should be inserted to maintain order.
shift(self, periods, fill_value)	Shift values by desired number.
take(self, indices, allow_fill, fill_value)	Take elements from an array.
unique(self)	Compute the ExtensionArray of unique values.
view(self[, dtype])	Return a view on the array.
_concat_same_type(to_concat)	Concatenate multiple array.

continues on next page

Table 429 – continued from previous page

_formatter(self, boxed)	Formatting function for scalar values.
_from_factorized(values, original)	Reconstruct an ExtensionArray after factorization.
_from_sequence(scalars[, dtype, copy])	Construct a new ExtensionArray from a sequence of scalars.
_from_sequence_of_strings(strings[, dtype, copy])	Construct a new ExtensionArray from a sequence of strings.
_ndarray_values	Internal pandas method for lossy conversion to a NumPy ndarray.
_reduce(self, name[, skipna])	Return a scalar result of performing the reduction operation.
_values_for_argsort(self)	Return values for sorting.
_values_for_factorize(self)	Return an array and missing value suitable for factorization.

pandas.api.extensions.ExtensionArray.argsort

ExtensionArray.**argsort**(*self*, *ascending: bool = True*, *kind: str = 'quicksort'*, **args*, ***kwargs*) → numpy.ndarray

Return the indices that would sort this array.

Parameters

ascending [bool, default True] Whether the indices should result in an ascending or descending sort.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm.

***args, **kwargs:** passed through to `numpy.argsort()`.

Returns

ndarray Array of indices that sort `self`. If NaN values are contained, NaN values are placed at the end.

See also:

numpy.argsort Sorting implementation used internally.

pandas.api.extensions.ExtensionArray.astype

ExtensionArray.**astype**(*self*, *dtype*, *copy=True*)

Cast to a NumPy array with 'dtype'.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

copy [bool, default True] Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns

array [ndarray] NumPy ndarray with 'dtype' for its dtype.

pandas.api.extensions.ExtensionArray.copy

ExtensionArray.**copy**(*self*) → pandas.core.dtypes.generic.ABCExtensionArray

Return a copy of the array.

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray.dropna

ExtensionArray.**dropna**(*self*)

Return ExtensionArray without NA values.

Returns

valid [ExtensionArray]

pandas.api.extensions.ExtensionArray.factorize

ExtensionArray.**factorize**(*self*, *na_sentinel: int = - 1*) → Tuple[numpy.ndarray, pandas.core.dtypes.generic.ABCExtensionArray]

Encode the extension array as an enumerated type.

Parameters

na_sentinel [int, default -1] Value to use in the *codes* array to indicate missing values.

Returns

codes [ndarray] An integer NumPy array that's an indexer into the original ExtensionArray.

uniques [ExtensionArray] An ExtensionArray containing the unique values of *self*.

Note: uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in *self*.

See also:

factorize Top-level factorize method that dispatches here.

Notes

pandas.factorize() offers a *sort* keyword as well.

pandas.api.extensions.ExtensionArray.fillna

ExtensionArray.**fillna**(*self*, *value=None*, *method=None*, *limit=None*)

Fill NA/NaN values using the specified method.

Parameters

value [scalar, array-like] If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like 'value' can be given. It's expected that the array-like have the same length as 'self'.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap.

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns

ExtensionArray With NA/NaN filled.

pandas.api.extensions.ExtensionArray.isna

ExtensionArray.**isna**(*self*) → ~ArrayLike

A 1-D array indicating if each value is missing.

Returns

na_values [Union[np.ndarray, ExtensionArray]] In most cases, this should return a NumPy ndarray. For exceptional cases like `SparseArray`, where returning an ndarray would be expensive, an ExtensionArray may be returned.

Notes

If returning an ExtensionArray, then

- `na_values._is_boolean` should be True
- *na_values* should implement *ExtensionArray._reduce()*
- `na_values.any` and `na_values.all` should be implemented

pandas.api.extensions.ExtensionArray.ravel

ExtensionArray.**ravel**(*self*, *order='C'*) → pandas.core.dtypes.generic.ABCExtensionArray

Return a flattened view on this array.

Parameters

order [{None, 'C', 'F', 'A', 'K'}, default 'C']

Returns

ExtensionArray

Notes

- Because ExtensionArrays are 1D-only, this is a no-op.
- The "order" argument is ignored, is for compatibility with NumPy.

pandas.api.extensions.ExtensionArray.repeat

ExtensionArray.**repeat**(*self*, *repeats*, *axis=None*)

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

repeats [int or array of ints] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

axis [None] Must be `None`. Has no effect but is accepted for compatibility with numpy.

Returns

repeated_array [ExtensionArray] Newly created ExtensionArray with repeated elements.

See also:

`Series.repeat` Equivalent function for Series.

`Index.repeat` Equivalent function for Index.

`numpy.repeat` Similar method for `numpy.ndarray`.

`ExtensionArray.take` Take arbitrary positions.

Examples

```
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

pandas.api.extensions.ExtensionArray.searchsorted

ExtensionArray.**searchsorted**(*self*, *value*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array *self* (a) such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Assuming that *self* is sorted:

side	returned index *i* satisfies
left	`self[i-1] < value <= self[i]`
right	`self[i-1] <= value < self[i]`

Parameters

value [array_like] Values to insert into *self*.

side [{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter [1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

Returns

array of ints Array of insertion points with the same shape as *value*.

See also:

numpy.searchsorted Similar method from NumPy.

pandas.api.extensions.ExtensionArray.shift

ExtensionArray.**shift**(*self*, *periods: int = 1*, *fill_value: object = None*) → pandas.core.dtypes.generic.ABCExtensionArray

Shift values by desired number.

Newly introduced missing values are filled with `self.dtype.na_value`.

New in version 0.24.0.

Parameters

periods [int, default 1] The number of periods to shift. Negative values are allowed for shifting backwards.

fill_value [object, optional] The scalar value to use for newly introduced missing values. The default is `self.dtype.na_value`.

New in version 0.24.0.

Returns

ExtensionArray Shifted.

Notes

If `self` is empty or `periods` is 0, a copy of `self` is returned.

If `periods > len(self)`, then an array of size len(self) is returned, with all values filled with `self.dtype.na_value`.

pandas.api.extensions.ExtensionArray.take

`ExtensionArray.`**`take`**(*self*, *indices: Sequence[int]*, *allow_fill: bool = False*, *fill_value: Any = None*) → pandas.core.dtypes.generic.ABCExtensionArray

Take elements from an array.

Parameters

indices [sequence of int] Indices to be taken.

allow_fill [bool, default False] How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other other negative values raise a `ValueError`.

fill_value [any, optional] Fill value to use for NA-indices when *allow_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing "boxed" scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns

ExtensionArray

Raises

IndexError When the indices are out of bounds for the array.

ValueError When *indices* contains negative values other than `-1` and *allow_fill* is True.

See also:

`numpy.take`

`api.extensions.take`

Notes

ExtensionArray.take is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill_value*.

Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
                  allow_fill=allow_fill)
    return self._from_sequence(result, dtype=self.dtype)
```

pandas.api.extensions.ExtensionArray.unique

ExtensionArray.**unique**(*self*)

Compute the ExtensionArray of unique values.

Returns

uniques [ExtensionArray]

pandas.api.extensions.ExtensionArray.view

ExtensionArray.**view**(*self*, *dtype=None*) → Union[pandas.core.dtypes.generic.ABCExtensionArray, numpy.ndarray]

Return a view on the array.

Parameters

dtype [str, np.dtype, or ExtensionDtype, optional] Default None.

Returns

ExtensionArray A view of the *ExtensionArray*.

pandas.api.extensions.ExtensionArray._concat_same_type

classmethod ExtensionArray.**_concat_same_type**(*to_concat: Sequence[pandas.core.dtypes.generic.ABCExtensionArray]*) → pandas.core.dtypes.generic.ABCExtensionArray

Concatenate multiple array.

Parameters

to_concat [sequence of this type]

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray._formatter

ExtensionArray.**_formatter**(*self*, *boxed: bool = False*) → Callable[[Any], Union[str, NoneType]]

Formatting function for scalar values.

This is used in the default '__repr__'. The returned formatting function receives instances of your scalar type.

Parameters

boxed [bool, default False] An indicated for whether or not your array is being printed within a Series, DataFrame, or Index (True), or just by itself (False). This may be useful if you want scalar values to appear differently within a Series versus on its own (e.g. quoted or not).

Returns

Callable[[Any], str] A callable that gets instances of the scalar type and returns a string. By default, `repr()` is used when `boxed=False` and `str()` is used when `boxed=True`.

pandas.api.extensions.ExtensionArray._from_factorized

classmethod ExtensionArray.**_from_factorized**(*values*, *original*)

Reconstruct an ExtensionArray after factorization.

Parameters

values [ndarray] An integer ndarray with the factorized values.

original [ExtensionArray] The original ExtensionArray that factorize was called on.

See also:

factorize

ExtensionArray.factorize

pandas.api.extensions.ExtensionArray._from_sequence

classmethod ExtensionArray.**_from_sequence**(*scalars*, *dtype=None*, *copy=False*)

Construct a new ExtensionArray from a sequence of scalars.

Parameters

scalars [Sequence] Each element will be an instance of the scalar type for this array, `cls.dtype.type`.

dtype [dtype, optional] Construct for this particular dtype. This should be a Dtype compatible with the ExtensionArray.

copy [bool, default False] If True, copy the underlying data.

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray._from_sequence_of_strings

classmethod ExtensionArray.**_from_sequence_of_strings**(*strings*, *dtype=None*, *copy=False*)

Construct a new ExtensionArray from a sequence of strings.

New in version 0.24.0.

Parameters

strings [Sequence] Each element will be an instance of the scalar type for this array, `cls.dtype.type`.

dtype [dtype, optional] Construct for this particular dtype. This should be a Dtype compatible with the ExtensionArray.

copy [bool, default False] If True, copy the underlying data.

Returns

ExtensionArray

pandas.api.extensions.ExtensionArray._ndarray_values

property ExtensionArray.**_ndarray_values**

Internal pandas method for lossy conversion to a NumPy ndarray.

This method is not part of the pandas interface.

The expectation is that this is cheap to compute, and is primarily used for interacting with our indexers.

Returns

array [ndarray]

pandas.api.extensions.ExtensionArray._reduce

ExtensionArray.**_reduce**(*self*, *name*, *skipna=True*, ***kwargs*)

Return a scalar result of performing the reduction operation.

Parameters

name [str] Name of the function, supported values are: { any, all, min, max, sum, mean, median, prod, std, var, sem, kurt, skew }.

skipna [bool, default True] If True, skip NaN values.

****kwargs** Additional keyword arguments passed to the reduction function. Currently, *ddof* is the only supported kwarg.

Returns

scalar

Raises

TypeError [subclass does not define reductions]

pandas.api.extensions.ExtensionArray._values_for_argsort

ExtensionArray.**_values_for_argsort**(*self*) → numpy.ndarray

Return values for sorting.

Returns

ndarray The transformed values should maintain the ordering between values within the array.

See also:

ExtensionArray.argsort

pandas.api.extensions.ExtensionArray._values_for_factorize

ExtensionArray.**_values_for_factorize**(*self*) → Tuple[numpy.ndarray, Any]

Return an array and missing value suitable for factorization.

Returns

values [ndarray] An array suitable for factorization. This should maintain order and be a supported dtype (Float64, Int64, UInt64, String, Object). By default, the extension array is cast to object dtype.

na_value [object] The value in *values* to consider missing. This will be treated as NA in the factorization routines, so it will be coded as *na_sentinal* and not included in *uniques*. By default, `np.nan` is used.

Notes

The values returned by this method are also used in *pandas.util.hash_pandas_object()*.

3.16.7 pandas.arrays.PandasArray

class pandas.arrays.**PandasArray**(*values: Union[numpy.ndarray, PandasArray]*, *copy: bool = False*)

A pandas ExtensionArray for NumPy data.

New in version 0.24.0.

This is mostly for internal compatibility, and is not especially useful on its own.

Parameters

values [ndarray] The NumPy ndarray to wrap. Must be 1-dimensional.

copy [bool, default False] Whether to copy *values*.

Attributes

None	

Methods

None	

Additionally, we have some utility methods for ensuring your object behaves correctly.

api.indexers.check_array_indexer(array, indexer)	Check if *indexer* is a valid array indexer for *array*.

3.16.8 pandas.api.indexers.check_array_indexer

pandas.api.indexers.**check_array_indexer**(*array: ~ AnyArrayLike*, *indexer: Any*) → Any

Check if *indexer* is a valid array indexer for *array*.

For a boolean mask, *array* and *indexer* are checked to have the same length. The dtype is validated, and if it is an integer or boolean ExtensionArray, it is checked if there are missing values present, and it is converted to the appropriate numpy array. Other dtypes will raise an error.

Non-array indexers (integer, slice, Ellipsis, tuples, ..) are passed through as is.

New in version 1.0.0.

Parameters

array [array-like] The array that is being indexed (only used for the length).

indexer [array-like or list-like] The array-like that's used to index. List-like input that is not yet a numpy array or an ExtensionArray is converted to one. Other input types are passed through as is

Returns

numpy.ndarray The validated indexer as a numpy array that can be used to index.

Raises

IndexError When the lengths don't match.

ValueError When *indexer* cannot be converted to a numpy ndarray to index (e.g. presence of missing values).

See also:

`api.types.is_bool_dtype` Check if *key* is of boolean dtype.

Examples

When checking a boolean mask, a boolean ndarray is returned when the arguments are all valid.

```
>>> mask = pd.array([True, False])
>>> arr = pd.array([1, 2])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

An IndexError is raised when the lengths don't match.

```
>>> mask = pd.array([True, False, True])
>>> pd.api.indexers.check_array_indexer(arr, mask)
Traceback (most recent call last):
...
IndexError: Boolean index has wrong length: 3 instead of 2.
```

NA values in a boolean array are treated as False.

```
>>> mask = pd.array([True, pd.NA])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

A numpy boolean mask will get passed through (if the length is correct):

```
>>> mask = np.array([True, False])
>>> pd.api.indexers.check_array_indexer(arr, mask)
array([ True, False])
```

Similarly for integer indexers, an integer ndarray is returned when it is a valid indexer, otherwise an error is (for integer indexers, a matching length is not required):

```
>>> indexer = pd.array([0, 2], dtype="Int64")
>>> arr = pd.array([1, 2, 3])
>>> pd.api.indexers.check_array_indexer(arr, indexer)
array([0, 2])
```

```
>>> indexer = pd.array([0, pd.NA], dtype="Int64")
>>> pd.api.indexers.check_array_indexer(arr, indexer)
Traceback (most recent call last):
...
ValueError: Cannot index with an integer indexer containing NA values
```

For non-integer/boolean dtypes, an appropriate error is raised:

```
>>> indexer = np.array([0., 2.], dtype="float64")
>>> pd.api.indexers.check_array_indexer(arr, indexer)
Traceback (most recent call last):
...
IndexError: arrays used as indices must be of integer or boolean type
```

The sentinel `pandas.api.extensions.no_default` is used as the default value in some methods. Use an `is` comparison to check if the user provides a non-default value.

CHAPTER

FOUR

DEVELOPMENT

4.1 Contributing to pandas

Table of contents:

4.1.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are brand new to pandas or open-source development, we recommend going through the GitHub "issues" tab to find issues that interest you. There are a number of issues listed under Docs and good first issue where you could start out. Once you've found an interesting issue, you can return here to get your development environment setup.

When you start working on an issue, it's a good idea to assign the issue to yourself, so nobody else duplicates the work on it. GitHub restricts assigning issues to maintainers of the project only. In most projects, and until recently in pandas, contributors added a comment letting others know they are working on an issue. While this is ok, you need to check each issue individually, and it's not possible to find the unassigned ones.

For this reason, we implemented a workaround consisting of adding a comment with the exact text *take*. When you do it, a GitHub action will automatically assign you the issue (this will take seconds, and may require refreshint the page to see it). By doing this, it's possible to filter the list of issues and find only the unassigned ones.

So, a good way to find an issue to start contributing to pandas is to check the list of unassigned good first issues and assign yourself one you like by writing a comment with the exact text *take*.

If for whatever reason you are not able to continue working with the issue, please try to unassign it, so other people know it's available again. You can check the list of assigned issues, since people may not be working in them anymore. If you want to work on one that is assigned, feel free to kindly ask the current assignee if you can take it (please allow at least a week of inactivity before considering work in the issue discontinued).

Feel free to ask questions on the mailing list or on Gitter.

4.1.2 Bug reports and enhancement requests

Bug reports are an important part of making *pandas* more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing. See this stackoverflow article and this blogpost for tips on writing a good bug report.

Trying the bug-producing code out on the *master* branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self-contained Python snippet reproducing the problem. You can format the code nicely by using GitHub Flavored Markdown:

```
```python
>>> from pandas import DataFrame
>>> df = DataFrame(...)
...
```
```

2. Include the full version string of *pandas* and its dependencies. You can use the built-in function:

```
>>> import pandas as pd
>>> pd.show_versions()
```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the *pandas* community and be open to comments/ideas from others.

4.1.3 Working with the code

Now that you have an issue you want to fix, enhancement to add, or documentation to improve, you need to learn how to work with GitHub and the *pandas* code base.

Version control, Git, and GitHub

To the new user, working with Git is one of the more daunting aspects of contributing to *pandas*. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- the GitHub help pages.
- the NumPy's documentation.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the pandas project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/pandas.git pandas-yourname
cd pandas-yourname
git remote add upstream https://github.com/pandas-dev/pandas.git
```

This creates the directory *pandas-yourname* and connects your repository to the upstream (main project) *pandas* repository.

Creating a development environment

To test out code changes, you'll need to build pandas from source, which requires a C compiler and Python environment. If you're making documentation changes, you can skip to *Contributing to the documentation* but you won't be able to build the documentation locally before pushing your changes.

Installing a C compiler

Pandas uses C extensions (mostly written using Cython) to speed up certain operations. To install pandas from source, you need to compile these C extensions, which means you need a C compiler. This process depends on which platform you're using.

Windows

You will need Build Tools for Visual Studio 2017.

> **Warning:** You DO NOT need to install Visual Studio 2019. You only need "Build Tools for Visual Studio 2019" found by scrolling down to "All downloads" -> "Tools for Visual Studio 2019".

Mac OS

Information about compiler installation can be found here: https://devguide.python.org/setup/#macos

Unix

Some Linux distributions will come with a pre-installed C compiler. To find out which compilers (and versions) are installed on your system:

```
# for Debian/Ubuntu:
dpkg --list | grep compiler
# for Red Hat/RHEL/CentOS/Fedora:
yum list installed | grep -i --color compiler
```

GCC (GNU Compiler Collection), is a widely used compiler, which supports C and a number of other languages. If GCC is listed as an installed compiler nothing more is required. If no C compiler is installed (or you wish to install a newer version) you can install a compiler (GCC in the example code below) with:

```
# for recent Debian/Ubuntu:
sudo apt install build-essential
# for Red Had/RHEL/CentOS/Fedora
yum groupinstall "Development Tools"
```

For other Linux distributions, consult your favourite search engine for compiler installation instructions.

Let us know if you have any difficulties by opening an issue or reaching out on Gitter.

Creating a Python environment

Now that you have a C compiler, create an isolated pandas development environment:

- Install either Anaconda or miniconda
- Make sure your conda is up to date (`conda update conda`)
- Make sure that you have *cloned the repository*
- `cd` to the *pandas* source directory

We'll now kick off a three-step process:

1. Install the build dependencies
2. Build and install pandas
3. Install the optional dependencies

```
# Create and activate the build environment
conda env create -f environment.yml
conda activate pandas-dev

# or with older versions of Anaconda:
source activate pandas-dev

# Build and install pandas
python setup.py build_ext --inplace -j 4
python -m pip install -e . --no-build-isolation --no-use-pep517
```

At this point you should be able to import pandas from your locally built version:

```
$ python  # start an interpreter
>>> import pandas
>>> print(pandas.__version__)
0.22.0.dev0+29.g4ad6d4d74
```

This will create the new environment, and not touch any of your existing environments, nor any existing Python installation.

To view your environments:

```
conda info -e
```

To return to your root environment:

```
conda deactivate
```

See the full conda docs here.

Creating a Python environment (pip)

If you aren't using conda for your development environment, follow these instructions. You'll need to have at least Python 3.6.1 installed on your system.

Unix/Mac OS

```
# Create a virtual environment
# Use an ENV_DIR of your choice. We'll use ~/virtualenvs/pandas-dev
# Any parent directories should already exist
python3 -m venv ~/virtualenvs/pandas-dev

# Activate the virtualenv
. ~/virtualenvs/pandas-dev/bin/activate

# Install the build dependencies
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext --inplace -j 0
python -m pip install -e . --no-build-isolation --no-use-pep517
```

Windows

Below is a brief overview on how to set-up a virtual environment with Powershell under Windows. For details please refer to the official virtualenv user guide

Use an ENV_DIR of your choice. We'll use ~virtualenvspandas-dev where '~' is the folder pointed to by either $env:USERPROFILE (Powershell) or %USERPROFILE% (cmd.exe) environment variable. Any parent directories should already exist.

```
# Create a virtual environment
python -m venv $env:USERPROFILE\virtualenvs\pandas-dev

# Activate the virtualenv. Use activate.bat for cmd.exe
~\virtualenvs\pandas-dev\Scripts\Activate.ps1

# Install the build dependencies
python -m pip install -r requirements-dev.txt

# Build and install pandas
python setup.py build_ext --inplace -j 0
python -m pip install -e . --no-build-isolation --no-use-pep517
```

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```